NVIDIA IndeX

Programmer's Manual

19 July 2024
Version 2020.2.2

NVIDIA IndeX – Programmer's Manual

Document build number rev376353

Contents

1 System overview

NVIDIA IndeX is a visualization platform for the access, processing, rendering, and visual annotation of large volume data sets. Integrated into applications through a C++ API, you link library files that are loaded at run-time to your application.

This document covers the basic programming principles of IndeX and how to compile and run IndeX programs.

1.1 Software components

The NVIDIA IndeX release contains the following parts:

- Two shared library files for the Linux platform:
  - `libdice`, the Distributed Computing Environment (DiCE) layer
  - `libnvindex`, the IndeX library
- The set of C++ header files that declare the components of the IndeX API.
- The NVIDIA IndeX Programmer's Manual (this document)
- The API documentation for IndeX and its support libraries
- Installation instructions

In the documentation, the IndeX and DiCE software libraries are referenced together as the "IndeX library." The application programming interface for these two libraries is called the "IndeX API."

2 Basic programming concepts

The follow sections describe the syntactic conventions, file system organization, and basic programming concepts of NVIDIA IndeX.

2.1 Naming conventions

The IndeX library is written in C++. It use the namespace `nv::index` for identifiers, and the `NVIDIA_INDEX_` prefix for macros.

Multiple words are concatenated with the underscore character (_) to form identifiers. Function names are all spelled in lowercase; type and class names start with one initial uppercase letter.

2.2 Main C++ header files

The C++ header file iindex.h in the IndeX API directory `nv/index` contains the base functionality for initializing and accessing the IndeX library. More specific components of the library also have their respective header files in `nv/index`.

2.3 Interfaces

The IndeX API follows current 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.

The global function `nv::index::nv_index_factory`() returns the main interface `nv::index::IIndex` that allows access to the whole library. From this interface other interfaces of the library can be accessed with the IIndex::get_api_component member function.

2.4 Reference counting

Interfaces are reference-counted dynamic resources that need to be released when no longer needed. Whenever a function returns a pointer to `mi::base::IInterface` or a class that uses it as a base class, the corresponding reference counter has already been increased by 1. That is, you can use the interface pointer without first determining if 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.

2.5 Handle class

To assist in memory management of class instances, the IndeX API provides the handle class `mi::base::Handle`. This handle class maintains pointer semantics 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; the 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::tr1::shared_ptr<T>`[1] or `boost::shared_ptr<T>`.[2]

2.6 Resources

As is typical in all resource-heavy applications, you should aim for minimal resource usage by releasing interface pointers as soon as you no longer need the resources to which they provide access. When a handle class instance goes out of scope, its destructor method releases its resources. By introducing a nested scope (surrounding statements in a { and } pair), resources acquired with the scope will automatically be released at its end.

2.7 Strings

The interface `mi::IString` represents strings. However, some methods return constant strings as a pointer to `const char` for simplicity. These strings are managed by the IndeX library 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.

1. `https://en.cppreference.com/w/cpp/memory/shared_ptr`
2. `https://www.boost.org/doc/libs/1_58_0/libs/smart_ptr/shared_ptr.htm`

3 Compiling and running programs

Source code
`example_shared.h` (page 322)

The NVIDIA IndeX software distribution contains example programs that present important background concepts and demonstrate the use of the classes and methods of the IndeX API. These example programs are described in the remainder of the Programmer's Manual. Each example program is preceded by a description of the features it demonstrates, and includes links from the names of classes and methods to the relevant API documentation.

3.1 How to compile a program

The NVIDIA IndeX software distribution contains a set of example programs. It is recommended that you first compile them to test your installation of IndeX. You can also use these programs as a starting point for your own development.

The release also contains a `Makefile` to support building the examples on the Linux platform. See the `README` text for installation instructions and possible customization steps.

You can compile the examples by hand following the steps below. (Environment variable assignment uses the Bash syntax.)

1. Define the environment variable `NVINDEX_ROOT` to be the installation root of NVIDIA IndeX:

```
export NVINDEX_ROOT=your-index-installation-directory
```

2. Set `NVINDEX_EXAMPLE` to be the IndeX example directory:

```
export NVINDEX_EXAMPLE=$NVINDEX_ROOT/src/index_examples
```

3. Use the DiCE and IndeX include files in your source:

```
#include <mi/dice.h>
#include <nv/index/iindex.h>
```

4. Load the IndeX library dynamically and access the API entry point:

```
mi::neuraylib::INeuray* dice_interface = load_and_get_idice("libdice↦
    .so");
nv::index::IIndex nvindex_interface = load_and_get_iindex("↦
    libnvindex.so");
```

See example_shared.h for the definition of these convenience methods. Alternatively, if you want to use the `mi::base::Handle` class, you can load the libraries in this way:

```
mi::base::Handle<mi::neuraylib::INeuray>
    dice_interface(load_and_get_idice("libdice.so"));

mi::base::Handle<nv::index::IIndex>
    nvindex_interface(load_and_get_iindex("libnvindex.so"));
```

As a starting point for you own project, you can use the provided `Makefile` in the `$exttt{NVINDEX_EXAMPLE}` directory. (In that case, the build location is assumed to be `$exttt{NVINDEX_EXAMPLE}`.)

5. Compile the program with the appropriate include-path argument that specifies the location of the IndeX and DiCE header files. A `g++` compiler call could look like this:

```
g++ -I$NVINDEX_ROOT/include -c my_example.cpp -o my_example.o -↦
   pthread
```

6. Link the program with the necessary libraries, which provide `dlopen()`, threading, math and CUDA support: A `g++` linker call could look like this:

```
g++ -o my_example my_example.o -ldl -pthread -lm -lcudart
```

7. Make the IndeX and DiCE libraries as well as their dependencies known to your system so that they are found when starting the example program. You can achieve this by placing the shared library in an appropriate location (such as `/usr/lib`) or by setting the environment variable `LD_LIBRARY_PATH`.

```
export LD_LIBRARY_PATH=/your/library/path
```

If you use the same directory structure as in the IndeX software distribution, `$NVINDEX_ROOT/src/setup.sh` can define the necessary `LD_LIBRARY_PATH`.

3.2 How to run the example programs

Once compiled, the example programs need to link to the IndeX library at runtime, so you will need to define the library's location using the environment variable `LD_LIBRARY_PATH`. This can be done using the `$NVINDEX_ROOT/src/setup.sh` bash script.

Many example programs run without command-line options. For example, to run the program defined by `start_index/index_start.cpp`, make `start_index` the current directory and enter:

```
./index_start
```

Many of the example programs also have a command line option for usage information to describe the options available for the program. To see the usage information, enter the command with the option `-h`. For example, to see the options for the `configuration` example program, enter:

```
./configuration -h
```

4 Initializing and configuring IndeX

4.1 Startup and shutdown

All programs using NVIDIA IndeX must perform the same initialization and shutdown procedures. The example program in this section loads the IndeX library, making available the classes and functions of the IndeX API. It then prints version information and unloads the library. (See "Compiling and running programs" (page 4) for a description of how the example programs can be compiled.)

4.1.1 Main API access point

The function `nv::index::nv_index_factory()` is the only public access point to all algorithms and data structures in the IndeX API. This factory function returns a pointer to an instance of the main `nv::index::IIndex` interface. The interface instance is used to configure, initialize, and shut down access to the IndeX library. All components of the library are also acquired through the instance. Only one interface instance can be used in an application, so the function `nv::index::nv_index_factory()` may only be called once per process.

Before you are able to call the factory function, you need to load the IndeX library file `libnvindex.so` and access the factory function. To simplify this task, a convenience function load_and_get_iindex() is defined in the example code.

Note that it is not required to use load_and_get_iindex(). Especially in larger application you might want to write your own code to load the shared library and locate its factory function. In such cases, you need to call `nv::index::nv_index_factory()` directly. For simplicity, the examples will use the convenience function load_and_get_iindex() instead.

4.1.2 Starting and shutting down NVIDIA IndeX

The `nv::index::IIndex` interface is used to start and shut down NVIDIA IndeX. The API can only be used after the libraries have been initialized and can no longer be used after shutdown. Startup does not happen during the `nv::index::nv_index_factory()` call because you might want to configure the behavior of the API, which needs to happen before startup (see "Configuration" (page 8) for details).

Finally, you have to shut down the API before terminating the application. At this point, you should have released all interface pointers except the pointer to the main `nv::index::IIndex` interface. If you are using the `mi::base::Handle` class, make sure that all handle instances have gone out of scope.

4.2 Configuration

Source code
`configuration.cpp` (page 122)

4.2.1 Error handling

Errors are reported in the following ways:

By returning an integer value
Some methods indicate their success or failure by an integer return value (as shown in the previous example (page 7)). The general rule is that `0` indicates success, and all other values indicate failure.

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

In this and the following examples we use a helper macro called check_success() to check for errors. If the condition is false, the macro prints an error message and exits. In production code, errors should be handled more thoroughly, but such handling has been omitted in these examples for brevity.

4.2.2 Configuration of the IndeX API

The behavior of IndeX is configured through several interfaces which can be obtained from `nv::index::IIndex::get_api_component`. (See the function's documentation for a list of these interfaces.)

You can customize the logging behavior of IndeX by providing your own logging object. This object must implement the `mi::base::ILogger` interface. A very minimal implementation that prints all messages to `stderr` is presented in this example. A similar implementation is used by default if you do not provide your own implementation.

4.3 Creating a session

Source code
`session.cpp` (page 482)

4.3.1 The session

An observer (an application user) of volume data in IndeX maintains an interaction with the application called a *session*. This session maintains the necessary information about the camera and the scene for that observer. An IndeX application must therefore create at least one session.

4.3.2 Accessing API components

You can use the functionality of the IndeX library through objects that are made available with the IndeX API interface function `nv::index::IIndex::get_api_component`.

4.3.3 Creating an IndeX transaction

One important component of IndeX is the cluster-wide distributed database. The database model is transaction-based. When you want to change a database element, you need a `mi::neuraylib::IDice_transaction`. You can store serializable database elements in the DiCE database through `mi::neuraylib::IDice_transaction::store`. These elements are visible from other hosts after calling `mi::neuraylib::IDice_transaction::commit`.

4.3.4 Creating the IIndex_session and ISession

Most of the functions of the IndeX library originate from `nv::index::IIndex_session`. An `nv::index::IIndex_session` object can create (currently one) `nv::index::ISession` for the user. Through the session, the user can then access the scene.

5 Creating elements

5.1 Sparse volume data

Source code
`cluster_rendering_mainhost.cpp` (page 104)

5.1.1 Overview of sparse volume data creation

To visualize sparse volume data using NVIDIA IndeX, the data must be converted from its native format to the sparse volume representation in the IndeX API. This conversion is implemented by a class called a *sparse volume generator*. The sparse volume generator must be a subclass of `nv::index::IDistributed_data_import_callback`. An instance of this class is passed as an argument to the API call `nv::index::IScene::create_sparse_volume`.

NVIDIA IndeX provides a parallel and on-demand solution for creating sparse volume data. In this context, "sparse volume data creation" can mean:

- Loading sparse volume data from data files
- Computing synthetic data (based on a sparse volume generation algorithm)
- The result of operations on existing sparse volume data (copy, filter, etc.)

This data creation depends on a user-defined `nv::index::IDistributed_data_import_callback`. This callback defines the method used to import sparse volume data, for example, loaded from a file or generated procedurally as the implementation of a sparse volume generation algorithm.

Sparse volume data is created in two steps:

1. Create a sparse volume generator through the IndeX API. This initializes a sparse volume scene element, but actual data is not created at this point.
2. When the IndeX library first accesses the sparse volume data, the generator is run as a callback. The callback will fill the data on demand.

The sparse volume generator callback permits the application to create the data in an appropriate manner. Unnecessary sections of data (for example, regions not visible from the current camera position) are not generated, saving time and memory. The IndeX library automatically detects the necessary parts of the data, executing data creation in parallel.

After a sparse volume scene element is created, it must be made part of the scene to visualize it. Sparse volume data, triangle mesh data, and any IDistributed_data_import_callback created data are considered *large data*. Large data as part of the hierarchical scene description requires a `nv::index::IStatic_scene_group` node.

Figure 5.1 (page 11) shows the output image of this example code. This example sets up the IndeX library, creates a small sparse volume data (20x20x20) in the scene, applies a colormap, and renders one frame to an image file.

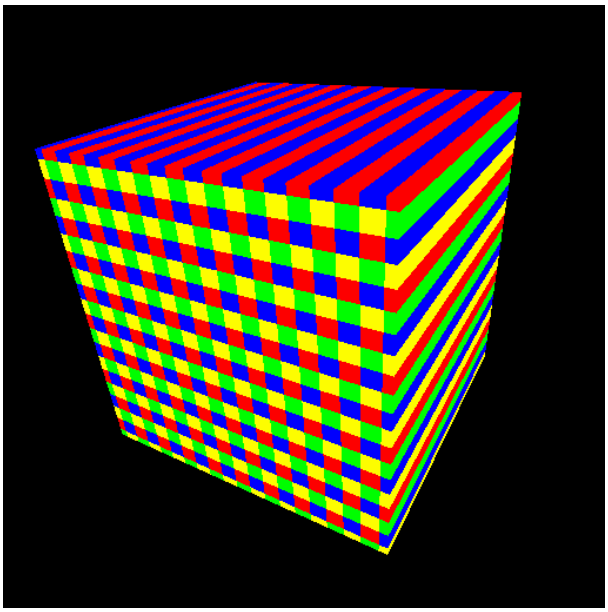

Fig. 5.1 - Synthetically generated small volume (20x20x20) rendered from the example code

5.1.2 Synthetic sparse volume data generator

When the application calls the `nv::index::IScene::create_sparse_volume` API method, the IndeX library acquires the bounding box of the sparse volume, transformation information, a generator, and a DiCE transaction. This API call creates an instance of `nv::index::ISparse_volume_scene_element`, which is a scene element. But the data itself may have not been created at this point. When the application invokes IndeX library to render the scene, IndeX library computes the visibility of each scene element, and if the data creation is needed, IndeX library calls the user-defined generator callback to produce the data.

The callback mechanism insures that the application does not need to manage data creation. For sparse volume data, you first create a `nv::index::ISparse_volume_scene_element` scene element. However, the volume data itself is not created until needed. For example, if the data is not in the region of interest, it will not be loaded. When the region of interest becomes larger, the necessary volume data will be loaded automatically. This mechanism is hidden from the application.

An example of implementation of `nv::index::IDistributed_data_import_callback` is provided by application_layer's plugin, base_importer.

5.1.3 Adding a volume element to the scene

After a volume scene element is created, it is inserted into the hierarchical scene description so it can be visualized. The session in this example contains a scene defined by a root node of type `nv::index::IStatic_scene_group`. Because it is a static group node, the volume element can be directly appended to it.

A sparse volume is defined as large data, requiring the `nv::index::IStatic_scene_group` node. The example code creates such a `nv::index::IStatic_scene_group` node and the sparse volume is appended to it. This node is then appended to the scene root.

This example demonstrates the simple case of a single volume. More typically, managing several volumes would be better implemented using a hierarchical scene description node, for example, with `nv::index::IScene_group`.

5.2 Heightfield data

Source code
`create_synthetic_heightfield.cpp` (page 246)

5.2.1 Overview of heightfield data creation

Heightfield data is visualized in NVIDIA IndeX by defining a *heightfield generator* that converts the data from its native format to the format required by the *heightfield scene element*.

A generator is a subclass of `nv::index::IDistributed_data_import_callback`. An instance of this class is passed as an argument to the API call `nv::index::IScene::create_regular_heightfield`.

NVIDIA IndeX provides a parallel and on-demand solution for creating heightfield data. Here heightfield data creation can be implemented by:

- Loading a heightfield data set from file
- Computing synthetic data (based on a heightfield generation algorithm)
- The result of operations on existing heightfield data (copy, filter, etc.)

Data generation depends on a user-defined `nv::index::IDistributed_data_import_callback` that implements file loading or synthetic data creation.

The heightfield data creation should be performed as follows:

1. Submit a heightfield generator through the IndeX API. This creates a heightfield scene element, but its actual data is not created.
2. When the IndeX library first accesses the heightfield data, the data generator callback is run, filling the data on demand.

Like the synthetic volume example (page 10), the heightfield generator callback permits the application to create the data in an appropriate manner. Data that does not need to be visualized (for example, data not visible from the current camera position) are not created to save time and memory. The library automatically detects the necessary parts of the data and the data creation process is executed in parallel.

After the heightfield scene element is created, it is inserted into the scene for visualization. Heightfield data, sparse volume subset data, and triangle mesh data are considered as *large data*. When large data is part of a hierarchical scene description, the `nv::index::IStatic_scene_group` node must be used.

Figure 5.2 (page 13) below shows the output image created by running this example. The code sets up the IndeX library, creates a minimal (20x20) heightfield data set in the scene, then render a single frame to an image file.

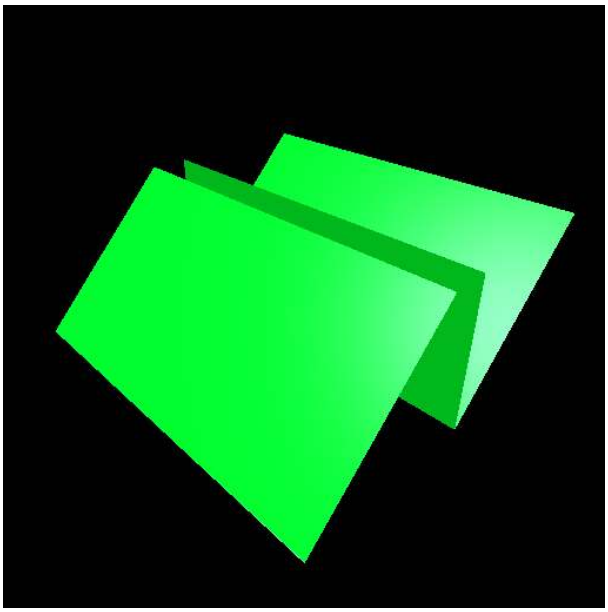

Fig. 5.2 - Synthetically generated minimal heightfield (20x20) rendered from the example code

5.2.2 Synthetic heightfield data generator

The generator concept is the same as for the volume generator described in section "Sparse volume data" (page 10).

This example has an implementation of Synthetic_heightfield_generator. This generator's `nv::index::IDistributed_data_import_callback::create` method will be called on demand when the data are needed for rendering.

5.2.3 Adding a heightfield element to the scene

The use of heightfield data in a hierarchically constructed scene is structurally similar to the case of volume data.

After a heightfield scene element is created, it is inserted into the hierarchical scene description so it can be visualized. The session in this example contains a scene defined by a root node of type `nv::index::IStatic_scene_group`. Because it is a static group node, the heightfield element can be directly appended to it.

A heightfield is defined as large data, requiring the `nv::index::IStatic_scene_group` node. The example code creates such a `nv::index::IStatic_scene_group` node and the heightfield is appended to it. This node is then appended to the scene root.

This example demonstrates the simple case of a single heightfield. More typically, managing several heightfields would be better implemented using a hierarchical scene description node, for example, with `nv::index::IScene_group`.

5.2.4 Embedded geometry in heightfields

Source code
`embedded_heightfield_geometry.cpp` (page 306)

5.2.4.1 Overview

The heightfield renderer supports rendering of connecting lines and isolated points that visualize the heightfield structure at locations where not enough details are provided to generate a triangular surface. These lines and points are not visible by default, but are enabled by applying an `nv::index::IHeightfield_geometry_settings` attribute to the heightfield scene element.

5.2.4.2 Color modes for embedded geometry

The IHeightfield_geometry_settings attribute defines which types of embedded geometry should be rendered and how they should be colored. The following color modes are supported, controlled by method `nv::index::IHeightfield_geometry_settings::set_color_mode` using values from the Color_mode enum.

- A **constant color** is used for rendering the lines and points.
- In **material color** mode, the color for rendering the lines and points results from modulating the surface material (applied to the heightfield structure by means of the scene description) with the constant color.
- The **volume texturing** mode refers to the built-in heightfield texturing technique that takes as input an associated volume and a colormap and maps the amplitude values that the heightfield intersect onto the heightfield's surface using the colormap. The color mode ensures that the lines and points and the heightfield surface are colored in the same way.
- The **compute texturing** mode refers to the compute integration implemented for the heightfield rendering using compute techniques such as `nv::index::IDistributed_compute_technique`. This color mode ensures that the computed color values also map to the lines and points.

5.2.4.3 Example program description

The example program loads an synthetic spiral-shaped heightfield from an image file and an additional mask file, so that the heightfield will contain holes and therefore isolated points and connecting lines will exist. All the different rendering modes are demonstrated, which can be controlled using the `-geometry` and `-texture` command-line arguments.

In addition to a standard scene setup with a regular heightfield, light and material, two `nv::index::IHeightfield_geometry_settings` attributes are created and inserted into the scene description before the `nv::index::IRegular_heightfield`. Optionally, a computed texture can be applied to the heightfield (specified with `-texture mandelbrot`), and a synthetic volume that is used in volume-texturing mode (specified with `-geometry texture`).

The images below show rendering output in fixed color mode (a field of the Color_mode enum), where two attributes with different colors are created and each of those applies to

either points or lines using
`nv::index::IHeightfield_geometry_settings::set_type_mask`.

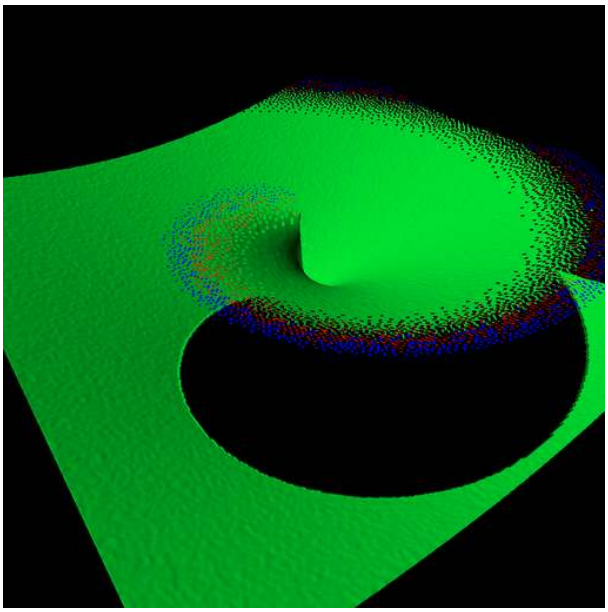

Fig. 5.3 - A spiral heightfield (green) with embedded isolated points (blue) and connecting lines (red)

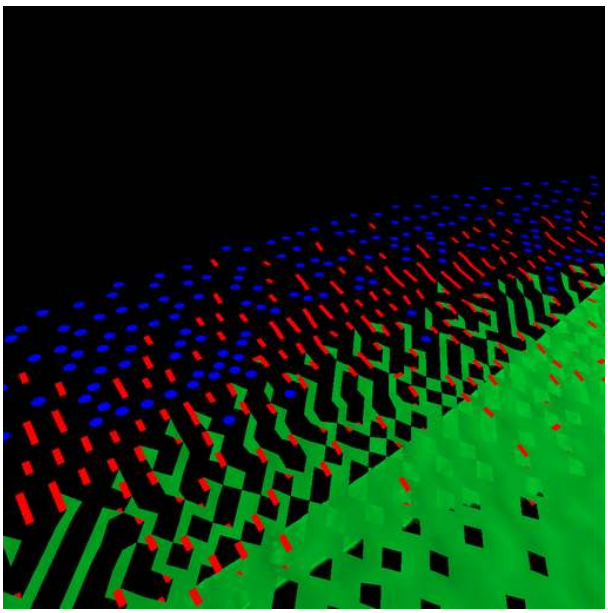

Fig. 5.4 - Detail of the points and lines embedded in the heightfield

5.3 Planes

Source code

`create_plane.cpp` (page 202)

5.3.1 Overview of plane shape creation

A *plane shape* is a pickable scene element that has a planar geometric definition with an associated textured image. The plane shape is defined by its position, normal vector, up vector, and extent. The texture image is mapped to plane shape for display.

The IndeX API recognize a user-defined class that implements the `nv::index::IPlane` interface.

After an application creates a `nv::index::IPlane` object, a call to `mi::neuraylib::IDice_transaction::store` is made to save the object to the database. The transaction must then be committed. If the plane shape is visible from the camera, they will be rendered by the render command. A plane shape is affected by the current region of interest. If a plane shape is located outside of the region of interest, it is clipped.

The plane shape can be displayed with a texturing technique by assigning an attribute of `nv::index::IDistributed_compute_technique`. If the plane shape returns a texturing technique value of `mi::neuraylib::NULL_TAG`, then the plane shape will not be rendered. The IndeX library distributes the plane shape rendering jobs over the cluster so that each job may be responsible for rendering a portion of the plane shape. This area is given by IndeX as a subregion bounding box, and the user-defined texturing technique must be aware of this bounding box to generate texture image colors. At that point, mapping these color values on the plane shape is the responsibility of IndeX.

5.3.2 The plane shape

You can find an example implementation of `nv::index::IPlane` in the example directory Mandelbrot_plane. This example shows the axis-aligned plane with rendered Mandelbrot set as seen in Figure 5.5.

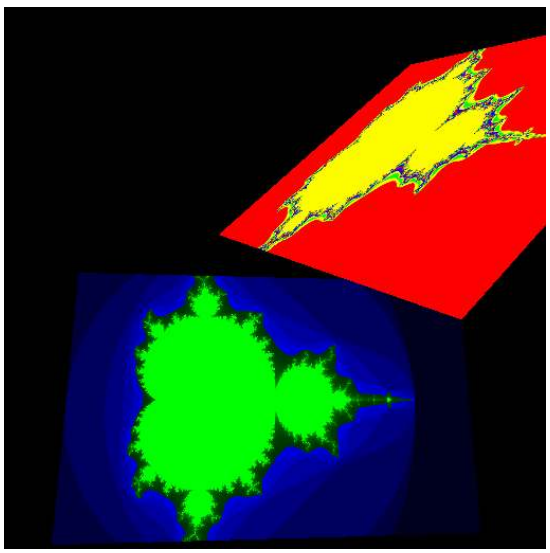

Fig. 5.5 - A plane shape rendering of a procedurally generated Mandelbrot set

The plane shape is specified by its position, normal, up vector, and its spatial extent. How to generate color values on the plane shape is specified by `nv::index::IDistributed_compute_technique` in the next section.

5.3.3 The texturing technique ITexture_buffer_generation_technique

A user needs to specify how to generate the color values on a plane shape through the interface `nv::index::IDistributed_compute_destination_buffer`.

In the simplest case, the user can return one color with single tile resolution. A plane shape of a single color will be displayed.

In this example, we show a procedural texture defined by an implementation of the Mandelbrot set. The IDistributed_compute_technique for this mapping is provided by the application_layer's data_analysis_and_processing component. The texture image is created in a parallel and distributed manner. The technique gets subregion bounding box information to compute the part of the Mandelbrot set. The user must aware of these subregion.

5.4 Polygons

Source code

`create_polygons.cpp` (page 222)

5.4.1 Overview of polygon shape creation

The `nv::index::IPolygon` shape is a pickable 2-D scene element that has a 2-D polygon shape which has three or more sides. The polygon shape is defined by vertex positions, a fill color, a fill style, and 3D center position.

The vertices of the polygon are defined by 2D pixel space coordinates. After an application creates an `nv::index::IPolygon` object, a call to `mi::neuraylib::IDice_transaction::store` can be made to save the object to the database. The transaction must then be committed.

If the polygon shape is visible from the camera, it will be rendered by the render command. A polygon shape is a raster shape, defined by 2D coordinates, so it is not affected by the region of interest (which only affects 3D shapes).

Fig. 5.6 - Various polygon shapes rendered in a scene.

5.5 Line path

Source code

`create_path.cpp` (page 183)

5.5.1 Overview of a line path shape creation

A *line path shape*, defined by class `nv::index::IPath_3D`, is a scene element that describes a connected series of line segments. A separate color can be specified for each segment.

To create a line path element in a scene:

1. Create a group node using the `nv::index::ITransformed_scene_group` class.
2. Create a light and materials and append to the group node.
3. Create an instance of `nv::index::IPath_3D` and append it to the group node created in the first step.
4. Append the group node to the scene.

Figure 5.7 (page 19) shows the output image of this example code.

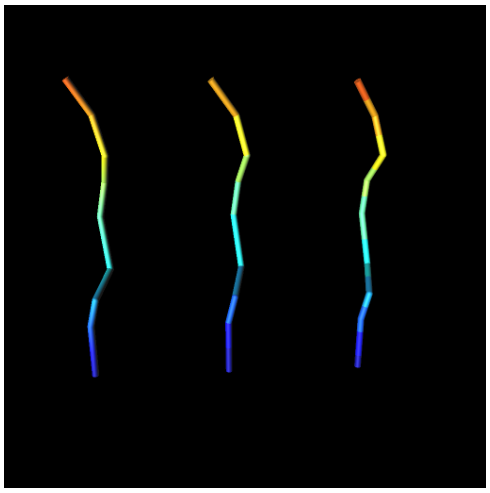

Fig. 5.7 - Line path shape examples

After an application creates an `nv::index::IPath_3D` object, a call to `mi::neuraylib::IDice_transaction::store` is made to save the object to the database. The transaction must then be committed. If the line set shape is visible from the camera, it will be rendered by the render command. A line path shape is affected by the region of interest. If the line path shape is located outside of the region of interest, it will be clipped.

> **Performance tip:** The line path shape is currently tested against the octree. If the path shape is far from the current visible octree nodes, this test becomes expensive. For example, the performance test will be expensive for volume data in a range of [1024, 1024, 1024] with an octree subcube size of [512, 512, 512]. If you put a line path at (100000, 0, 0)-(100010, 0, 0), the performance will be affected even though the line path is not visible.

The example code generates a PPM image file[1] as the rendering result.

5.6 Point sets

Source code

`create_point_set.cpp` (page 213)

5.6.1 Overview of point set shape creation

A *point set shape*, implemented by class `nv::index::IPoint_set`, is a pickable scene element specified by a position, a radius, and a color. scene element.

To create a point set element in a scene:

- Create a group node using the `nv::index::ITransformed_scene_group` class.
- Create a material using class `nv::index::IPhong_gl`. Append the material to the group node.
- Create an instance of `nv::index::IPoint_set` and append it to the group node.
- Append the group node to the scene.

1. `https://netpbm.sourceforge.net/doc/ppm.html`

The color of an `nv::index::IPoint_set` is defined by a material.

In the hierarchical scene description, an appended material affects the appearance of subsequently appended shapes. The color of the 3D points in the `nv::index::IPoint_set` is based on the color defined for each point, in combination with the appearance defined by the material and its dependencies (for example, the calculation of the highlight or the location of light sources).

Figure 5.8 shows the output image of the example code in this section. The top rows demonstrate ray-traced rendering mode. The bottom rows demonstrate raster rendering mode.

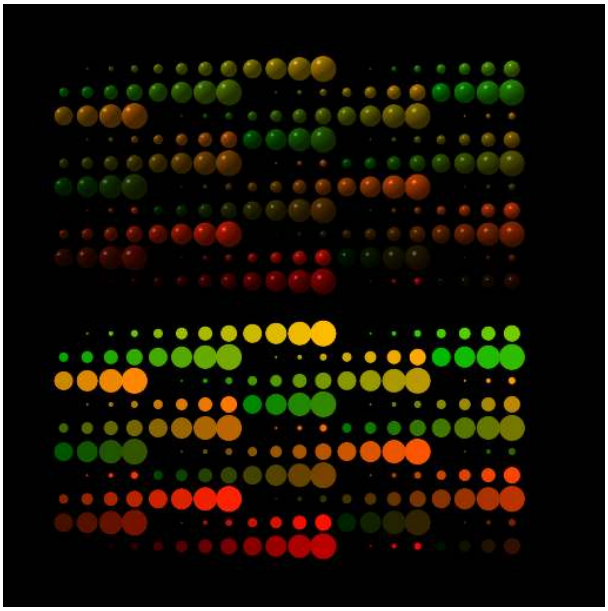

Fig. 5.8 - Point set shape example with various radii and various colors

The API of the IndeX library recognizes a user-defined class that implements the `nv::index::IPoint_set` interface.

After an application creates a `nv::index::IPoint_set` object, a call to `mi::neuraylib::IDice_transaction::store` is made to save the object to the database. The transaction must then be committed. If the point set shape is visible from the camera, it will be rendered by the render command. A point set shape is affected by the region of interest. If the point set shape is located outside of the region of interest, it will be clipped.

> **Performance tip:** The point set shape is currently tested against the octree. If the path shape is far from the current visible octree nodes, this test becomes expensive. For example, the performance test will be expensive for volume data in a range of [1024, 1024, 1024] with an octree subcube size of [512, 512, 512]. If you put a point set at (100000, 0, 0)-(100010, 0, 0), the performance will be affected even though the point set is not visible.

The example code generates a ppm file as the rendering result.

5.6.2 Point sets: user-defined attributes

Source code

`create_attribute_point_set.cpp` (page 148)

5.6.2.1 Overview of a user-defined attribute point set shape creation

Creation of point sets is demonstrated in section "Point sets" (page 19). This example shows how to implement a custom class derived from `nv::index::IPoint_set`.

Sometimes it is useful for the color and radius computation to belong to the point set class. This implementation example integrates the color and radius computation into a new class that is derived from `nv::index::IPoint_set`, called Attribute_point_set. The result is the same as the "Point sets" (page 19) example, but the code allows the developer to think in a more object-oriented way.

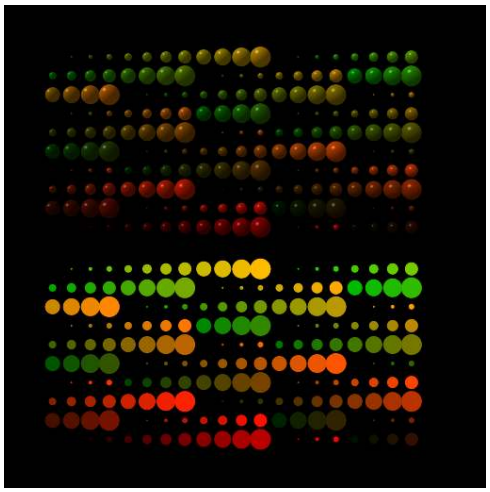

Fig. 5.9 - Point set shape example with various radii and colors

5.6.2.2 Attribute point set

Class Attribute_point_set contains points in which each color and radius varies according to an attribute. This example implementation uses two mapping functions. An attribute can be any scalar value, for instance, an amplitude value.

`mi::math::Color_struct get_color_from_attribute(mi::Float32 attribute)`
A map function from an attribute scalar value to a color of the point.

`mi::Float32 get_radius_from_attribute(mi::Float32 attribute)`
A map function from an attribute scalar value to a radius of the point.

Figure 5.9 (page 21) shows a rendering result using the Attribute_point_set class containing various colors and radii according to the attribute values. The example code generates a PPM image file as the rendering result.

5.7 Line sets

Source code
`create_line_set.cpp` (page 175)

5.7.1 Overview of line set shape creation

A *line set shape*, implemented by class `nv::index::ILine_set`, is a pickable scene element that has a line or cylinder shape and is specified by position, radius, color, and line style.

To create a line set shape in a scene:

1. Create a group node using the `nv::index::ITransformed_scene_group` class.
2. Create an instance of `nv::index::ILine_set` and append it to the group node.
3. Append the group node to the scene.

An `nv::index::ILine_set` is not affected by a material, therefore we have no material in this example. The color of the line is based on the per-segment color.

Figure 5.10 shows the output image of this example code.

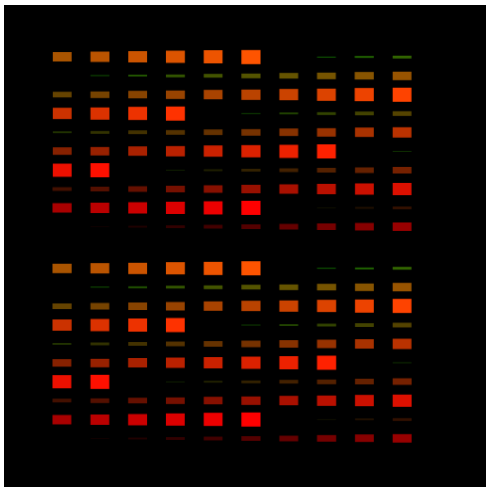

Fig. 5.10 - Line set shape example with various width and various colors

The API of the IndeX library recognizes a user defined class that implements the Elie_set interface.

After an application creates a `nv::index::ILine_set` object, a call to `mi::neuraylib::IDice_transaction::store` is made to save the object to the database. The transaction must then be committed. If the line set shape is visible from the camera, it will be rendered by the render command. A line set shape is affected by the region of interest. If the line set shape is located outside of the region of interest, it will be clipped.

> **Performance tip:** The line set shape is currently tested against the octree. If the path shape is far from the current visible octree nodes, this test becomes expensive. For example, the performance test will be expensive for volume data in a range of [1024, 1024, 1024] with an octree subcube size of [512, 512, 512]. If you put a line set at (100000, 0, 0)-(100010, 0, 0), the performance will be affected even though the line set is not visible.

The example code generates a PPM image file as the rendering result.

5.7.2 Line sets: user-defined attributes

Source code

`create_attribute_line_set.cpp` (page 140)

5.7.2.1 Creating a custom line set class

Creation of line sets is demonstrated in section "Line sets" (page 22). This example shows how to implement a custom class derived from `nv::index::ILine_set`.

Sometimes it is useful for the color and width computation to belong to the line set class. This implementation example integrates the color and width computation into a new class that is derived from `nv::index::ILine_set`, called Attribute_line_set. The result is the same as the "Line sets" (page 22) example, but the code allows the developer to think in a more object-oriented way.

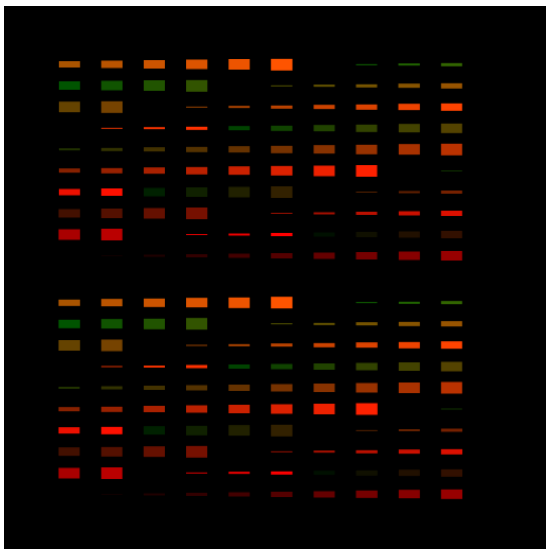

Fig. 5.11 - Line set shape example with various widths and colors

5.7.2.2 Attribute line segment set

Class Attribute_line_set contains lines in which each color and width varies according to an attribute. This example implementation uses two mapping functions. An attribute can be any scalar value, for instance, an amplitude value.

`mi::math::Color_struct get_color_from_attribute(mi::Float32 attribute)`
A map function from an attribute scalar value to a color of the line segment.

`mi::Float32 get_width_from_attribute(mi::Float32 attribute)`
A map function from an attribute scalar value to a width of the line segment.

Figure 5.11 (page 23) shows a rendering result using the Attribute_line_set class containing various colors and widths according to the attribute values.

The example code generates a PPM image file as the rendering result.

5.8 Stylized points and lines

Source code
`create_stylized_points_and_lines.cpp` (page 235)

5.8.1 Overview of stylized points and lines.

Point and line set shapes represent scene elements and can either be rendered in object space using a ray-tracer or in raster space using a rasterizer. The rasterizer of NVIDIA IndeX support a stylized rendering of points and lines. Points can be drawn as circles, squares, or triangles, or as a shaded sphere. Lines can be drawn using solid, dotted or dashed line styles. The size of points and lines are specified in raster space.

The typical process for creating instances of `nv::index::IPoint_set` and `nv::index::ILine_set` is described in sections "Point sets" (page 19) and "Line sets" (page 22), respectively.

Figure 5.12 shows an output image generated using the example code linked above. The top rows demonstrate rasterized lines using the dashed line style and colors. The bottom rows illustrate the rasterized points at various radii and colors.

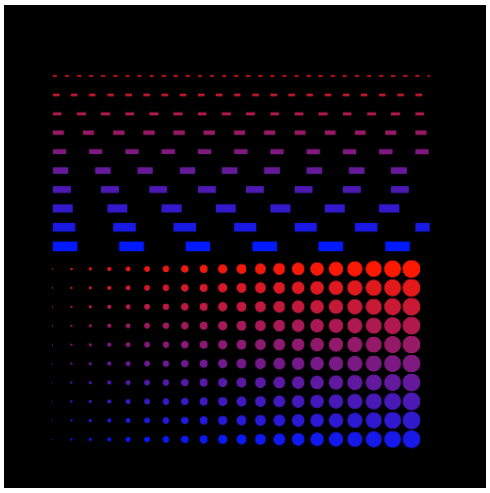

Fig. 5.12 - Points and lines rendered with various width and various colors

The API of the IndeX library recognizes a user-defined class that implements the `nv::index::IPoint_set` and the `nv::index::ILine_set` interfaces.

After an application creates instance of the classes IPoint_set and ILine_set and then stores them in the distributed database using the `mi::neuraylib::IDice_transaction::store` method.

The points and lines are rendered using the rasterizer. The NVIDIA IndeX rendering system ensures that the rasterized points and lines are then correctly integrated with the remaining scene elements, such as volumetric data or height maps.

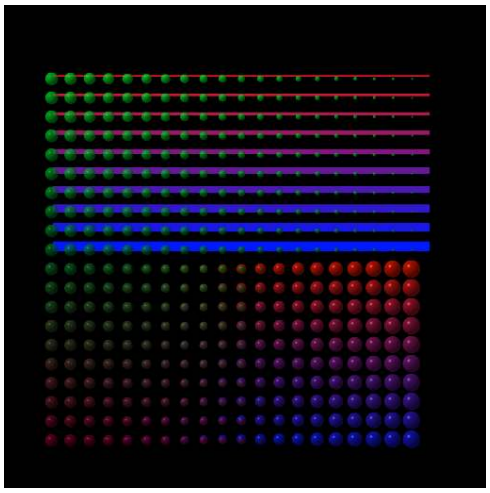

Fig. 5.13 - Semi-transparent points rendered as shaded spheres/circles and together with the opaque points and lines.

5.8.2 Point set

The point set provides means to store an array of vertices that represent the center of each point, an array of per-point color values, and an array of radius values. According to the point style each point of the point set will be rendered as a flat and filled circle, square, or triangle, or as a shaded sphere with extent in depth.

Point styles are defined by the enum `nv::index::IPoint_set::Point_style`. The following point styles are available:

```
FLAT_CIRCLE
SHADED_CIRCLE
FLAT_SQUARE
FLAT_TRIANGLE
SHADED_FLAT_CIRCLE
```

Figure 5.12 (page 24) shows a point set rendered as flat circles. Figure 5.13 shows an additional point set rendered as shaded circles.

5.8.3 Line set

A line set represent a collection of line segments without connectivity. The line set provides means to store an array of vertices that represent the start and end positions of each of the

many line segments, and an array of per segment color values and widths. Each line segment of the line set will be rendered in the specified line style.

Line styles are defined by the enum `nv::index::ILine_set::Line_style`. The following line styles are available:

```
LINE_STYLE_SOLID
LINE_STYLE_DASHED
LINE_STYLE_DOTTED
LINE_STYLE_CENTER
LINE_STYLE_HIDDEN
LINE_STYLE_PHANTOM
LINE_STYLE_DASHDOT
LINE_STYLE_BORDER
LINE_STYLE_DIVIDE
```

Figure 5.12 (page 24) shows dashed lines based on the `LINE_STYLE_DASHED` style. Figure 5.13 (page 25) shows solid lines based on the `LINE_STYLE_SOLID` style.

> **Note:** Line sets with connectivity such as line strips or line loops will be supported in the future.

The example code generates a PPM image file to illustrate point and line styles.

5.9 Circles and ellipsoids

Source code
`create_circles_and_ellipses.cpp` (page 156)

5.9.1 Overview of circle and ellipsoid shape creation

IndeX provides circle and ellipsoid shapes with the classes `nv::index::ICircle` and `nv::index::IEllipse`. Both are raster space objects and pickable. However since they are defined in raster space, the 3D region of interest doesn't affect their display. Figure 5.14 (page 27) shows the output of the API example code.

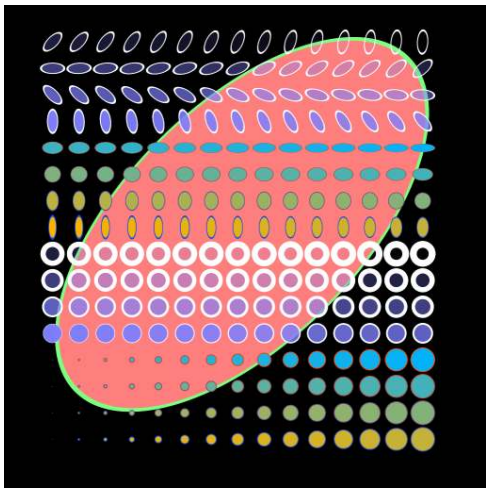

Fig. 5.14 - Various circle and ellipse shapes are rendered in a scene

5.9.2 Circle shape creation

`nv::index::ICircle` is a 2D raster object with an outline. To create an `nv::index::ICircle`, its center position and radius must be specified. The circle's appearance also depends on how the circle is outlined and filled, for which both style and color can be specified.

The lower section of Figure 5.14 (page 27) shows various circle shape examples.

5.9.3 Ellipse shape creation

`nv::index::IEllipse` is a 2D raster object with an outline. To create an `nv::index::IEllipse`, its center position, horizontal radius, vertical radius, and rotation angle in radians must be specified. Like the `nv::index::ICircle`, an `nv::index::IEllipse` may also specify the style and color of how it is outlined and filled.

The upper part of Figure 5.14 (page 27) shows various ellipse shapes.

5.10 Annotation labels

Source code

`create_annotations.cpp` (page 127)

5.10.1 Overview of annotation label shape creation

IndeX has two types of annotation string label shapes: `nv::index::ILabel_2D` and `nv::index::ILabel_3D`. The two label types define a planar space on which a string of characters can be mapped in order to annotate other scene elements. Both label types are pickable scene elements.

Figure 5.15 shows the output of API example code.

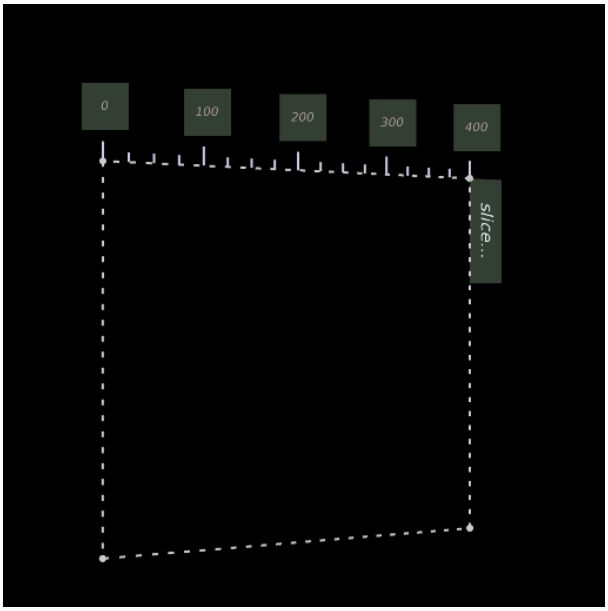

Fig. 5.15 - Annotation string label shapes rendered in a scene.

The two label types provide textual annotation; for annotation using images, see section "Icons" (page 30).

5.10.2 ILabel_2D and ILabel_3D dimension

Figure 5.16 shows the layout and size attributes of `nv::index::ILabel_2D` and `nv::index::ILabel_3D`. For `nv::index::ILabel_2D`, the width and height values specify screen-space pixels. For `nv::index::ILabel_3D`, the width and height specify the size of the label in object space. If a negative width is specified, IndeX computes the width value that best fits the annotation text. The padding places the annotation text in the total label area as shown in Figure 5.16.

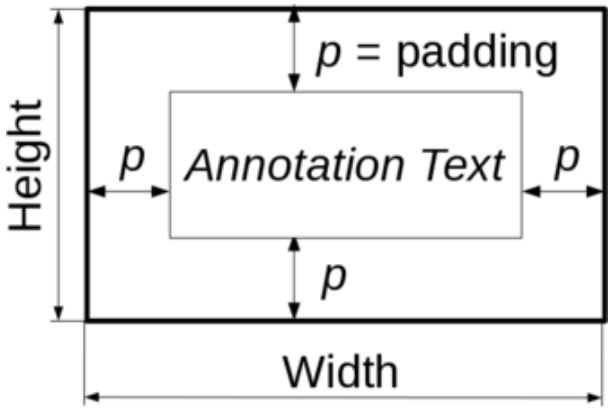

Fig. 5.16 - ILabel_2D and ILabel_3D dimensions

5.10.3 IFont: A font attribute of ILabel

To display a text in the label, IndeX uses the font attribute class `nv::index::IFont`. The font filename is specified for an `nv::index::IFont` instance using the `nv::index::IFont::set_file_name` method. When the font file is not found for the given path, the method returns `false`. The resolution of the font defined by the `nv::index::IFont::set_font_resolution` method defines the size of the internal bitmap used by the font for text display, and therefore affects the quality of the rendered text. The recommended resolution value is between 16 and 1024.

5.10.4 A two-dimensional label

`nv::index::ILabel_2D` is a 2D raster object, a plane on which strings of characters are mapped in the plane's 2D space. The labels of coordinates (0, 100, 200, 300, 400) at the top of Figure 5.16 are `nv::index::ILabel_2D` instances. The label geometry is specified by the `nv::index::ILabel_2D::set_geometry` method. The arguments of `nv::index::ILabel_2D::set_geometry` are: the lower left corner position of the label, a vector pointing to the right for the width direction, a vector pointing up for the height direction, and the height and width of the label.

5.10.5 A three-dimensional label

`nv::index::ILabel_3D` is a 3D object, a plane defined in three dimensions on which strings of characters are mapped.

In Figure 5.16, the label on the right ("slice...") is an instance of `nv::index::ILabel_3D`. The label geometry is specified by the `nv::index::ILabel_3D::set_geometry` method. The arguments of `nv::index::ILabel_3D::set_geometry` are: lower-left corner position of the label, a vector pointing to the right for the width direction, a vector pointing up for the height direction, and the height and width of the label.

5.11 Icons

Source code
`create_icons.cpp` (page 166)

5.11.1 Overview of icon shape creation

IndeX has two types of annotation image icon shapes: `nv::index::IIcon_2D` and `nv::index::IIcon_3D`. The two icon types define a planar space on which a string of characters can be mapped in order to annotate other scene elements. Both icon types are pickable scene elements.

Figure 5.17 shows the output of the example code that demonstrates using icons in a scene.

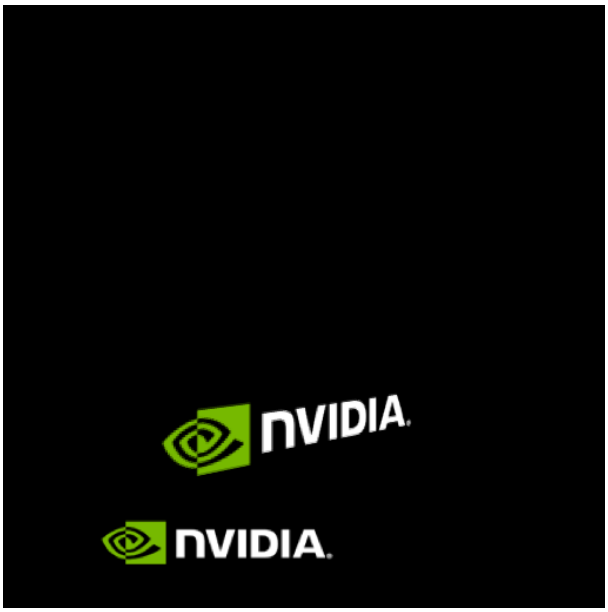

Fig. 5.17 - Annotation image icon shapes rendered in a scene.

The two icon types provide annotation through mapping images into the scene; for annotation using text, see section "Annotation labels" (page 28).

5.11.2 IIcon_2D and IIcon_3D dimensions

For `nv::index::IIcon_2D`, the width and height specify the size of the icon in screen-space pixels. For `nv::index::IIcon_3D`, the width and height specify the size of the icon in object space. If a negative width is specified, IndeX computes the width value that best fits the image in the icon.

5.11.3 ITexture: A texture for IIcon

To display an image in the icon, IndeX uses the texture attribute class, `nv::index::ITexture`. `nv::index::ITexture` gets an image buffer containing the texture using the `nv::index::ITexture::set_pixel_data` method. The application must ensure that the arguments passed to `nv::index::ITexture::set_pixel_data` (that specify the width, height, and format of the input pixel data) are correct. The `nv::index::ITexture` attribute must be defined before the icon scene element that wishes to use it as an argument.

5.11.4 A two-dimensional icon

`nv::index::IIcon_2D` is a 2D raster object, a plane on an image is mapped. The icon geometry is specified by the `nv::index::IIcon_2D::set_geometry` method. The arguments of `nv::index::IIcon_2D::set_geometry` are: the lower left corner position of the icon, a vector pointing to the right for the width direction, a vector pointing up for the height direction, and the height and width of the icon.

5.11.5 A three-dimensional icon

`nv::index::IIcon_3D` is a 3D object, a plane defined in three dimensions on which an image is mapped. The icon geometry is specified by the `nv::index::IIcon_3D::set_geometry` method. The arguments of `nv::index::IIcon_3D::set_geometry` are: lower-left corner position of the icon, a vector pointing to the right for the width direction, a vector pointing up for the height direction, and the height and width of the icon.

6 Rendering

6.1 Rendering a scene

Source code
`render_frame.cpp` (page 451)

6.1.1 Overview of rendering

The process of *rendering* in IndeX involves the creation and organization of *scene elements* (synthetic cameras, data and annotational objects, appearance definitions) with the various mechanisms provided by IndeX for computation, data management and process parallelism.

A minimal sequence for rendering would require that the application execute the following:

1. Create a session. (page 8)
2. Initialize the application rendering context.
3. Set up a camera.
4. Create a scene.
5. Synchronize the IndeX session and the application session.
6. Set up a canvas.
7. Issue a render call.

Typically, an application will first define the rendering context. Scene elements are then created — camera, a scene, and lights — that are required for rendering. Output of the rendered scene requires a structure that is analogous to film in traditional photography. In the IndeX library, the output object is called the *canvas*.

For simplicity, the example code in this section only defines a scene transformation matrix without any scene elements. Any rendering of a scene without scene elements will result in a black image. However, this minimal example can serve as a basic framework for scene construction and is used by many of the examples in this manual.

6.1.2 Initialize application rendering context

In the example source code for this section, render_frame.cpp, we first initialize our application rendering context. The application rendering context is basic infrastructure for rendering, including:

- DiCE component initialization for cluster wide database access.
- Initialize IndeX's session component (hierarchical scene description root etc.), rendering component (data structure for rendering), and cluster management component.

After these initialization, IndeX is ready to create and render a scene.

6.1.3 Camera set up

In the example source file render_frame.cpp, a call to the `nv::index::ISession::create_camera` method creates an `nv::index::ICamera` node in the database and returns a `mi::neuraylib::Tag` that is used to access the camera. The tag and the current `mi::neuraylib::IDice_transaction` are the arguments to function `setup_camera`, which defines the camera's geometric attributes: position, orientation, aperture, aspect ratio, focal length, and minimum and maximum clipping planes.

6.1.4 Scene definition

At a minimum, the following must defined:

1. Scene transform matrix
2. Region of interest of the scene space
3. Active camera

The function `setup_render_frame_scene` in render_frame.cpp is an example implementation of these definitions.

6.1.5 Synchronize the IndeX session and the application session

Once the scene has been set up, the scene data must be synchronized with the IndeX session.

6.1.6 Setup canvas

In this example, the canvas is provided by the canvas_infrastructure application_layer module, `nv::index::app::canvas_infrastructure::IIndex_image_file_canvas`. This is the renderer's *image target*.

6.1.7 Issue a rendering call

After the scene has been created and synchronized with the session, rendering is executed with the `nv::index::IIndex_rendering::render`() method. The rendered result image is contained by the canvas provided by the application_layer's canvas_infrastructure component. `nv::index::app::canvas_infrastructure::IIndex_image_file_canvas`.

6.2 Cluster rendering

Source code

`cluster_rendering_mainhost.cpp` (page 104)
`cluster_rendering_remotehost.cpp` (page 116)

6.2.1 Overview of cluster rendering

NVIDIA IndeX is capable of *cluster rendering* in which a single main host coordinates rendering with multiple remote hosts, the *cluster*. A typical role of the main host is interacting with the user and controlling the entire cluster. The remote hosts process sub-rendering tasks and return the resulting image to the main host.

Using a multi-host cluster has two advantages:

- Handling larger data sizes. The IndeX library can use the total memory size of all the hosts in the cluster.
- Improving rendering performance.

For example, a single host can have 24 GB of main memory and 12 GB of graphics memory. However, the IndeX library can employ multiple hosts as a rendering cluster. Therefore, a two-host cluster could have as much as 48 GB of main memory and 24 GB of GPU memory. The combined memory size allows IndeX to handle several hundred gigabtyes of volume data, with a performance improvement that scales linearly with additional hosts.

Rendering performance varies depending upon data set size and configuration. For example, if a data set is small and can be handled by a single GPU card, then the overhead of data distribution may eliminate the advantage of multiple host rendering.

Figure 6.1 shows the output image of the example code from this section, an artificially created volume of size 1020x1020x1020 rendered by two hosts (one main host and one remote host).

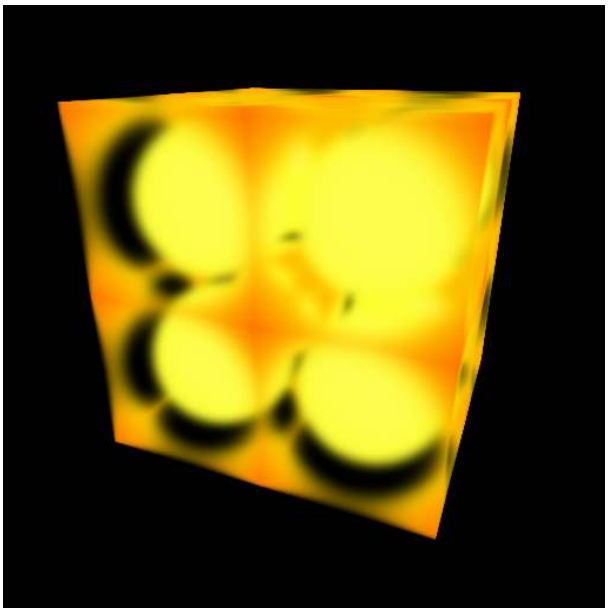

Fig. 6.1 - The output image of cluster rendering example

6.2.2 Network settings

In the example code, the rendering mode is “UDP multicast.” The example uses DiCE’s network communication layer that is based on TCP/UDP. With the same multicast address for the main host and the remote host, IndeX library automatically finds the remote host. Starting all remote hosts before starting the main host is preferred; starting a remote host after the main host has started, called *late joining*, requires additional data transfer between the hosts.

> **Note:** Before running the example code for this section, the network configuration should be checked to verify that UDP multicast is enabled.

This example does not support multiple network cards on a host. However, such support is available in DiCE, and support in the example is provided in a commented section of the

code. If hosts are connected to more than one network, the network to be used is specified by a call to `mi::neuraylib::INetwork_configuration::set_cluster_interface`.

6.2.3 Main host

The main host is called the *cluster controller* and is responsible for handling user requests such as camera movement, scene element creation, and other manipulations of the scene. In the example code, the main-host process checks the number of remote hosts in the cluster using UDP multicast, and employees them for rendering. This requires that the remote hosts processes are started first, followed by the cluster controller.

The user can specify multicast addresses to find the remote hosts and the number of frames to render using command line options. The `-h` option to the command-line executable of the example code will display the list of possible options.

Execution examples:

- Normal run:

```
run_example.sh example_cluster_rendering_mainhost
```

- Display help:

```
run_example.sh example_cluster_rendering_mainhost -h
```

6.2.4 Remote host

Processing on a remote host is controlled by the main host. The remote-host example uses a default multicast address. To specify a different address, use the `-dice::network::multicast_address` command-line option. To display all command-line options, use the `-h` option.

Execution example:

- Normal run:

```
run_example.sh example_cluster_rendering_remotehost
```

- Display help:

```
run_example.sh example_cluster_rendering_remotehost -h
```

6.2.5 Measuring cluster performance

Source code
`cluster_performance_mainhost.cpp` (page 88)

6.2.5.1 Overview of performance measurement

The IndeX library has several performance counters to measure which component spends how much time in order to discover performance bottleneck. For instance, if most of the

rendering time is spent uploading the volume data to GPU memory, then we can reconfigure the cluster that can hold the volume data in the GPU memory. This usually improves the performance significantly. These configurations depend upon data size and the user's resources.

To use this facility, the application first needs to enable performance measurements. These measurements are collected by IndeX and analyzed to produce the performance statistics. Performance measurement requires additional computation, but the overhead is low. Performance measurement is switched off by default.

6.2.5.2 Enabling performance monitoring

To enable performance monitoring, the user needs to edit `nv::index::IConfig_settings`. The method `nv::index::IConfig_settings::set_monitor_performance_values` can change the status as shown in the example code.

6.2.5.3 Querying performance values

Once performance monitoring is enabled, the render call `nv::index::IIndex_rendering::render()` returns an instance of `nv::index::IPerformance_values`. The user can query performance counter values using the `nv::index::IPerformance_values::get()` method.

There are two types of data acquisition methods for IPerformance_values: `nv::index::IPerformance_values::get()` and `nv::index::IPerformance_values::get_time()`.

Method `nv::index::IPerformance_values::get()` returns an integer value that describes how much data is transferred, or a float value that describes elapsed time. Currently, all the performance item's keys that contain "time_" or "frames_per_second" should be accessed by the `nv::index::IPerformance_values::get_time()` method.

6.2.5.4 Automatic number of spans control

A *span* is a processing unit of image composition in IndeX. The number of spans affects the number of parallel compositing jobs performed. When composition is parallelized, IndeX distributes the jobs over the cluster and the jobs are performed in parallel. This process involves a trade off between distribution overhead and the speed improvement of parallel processing. More jobs improve performance, but to a limit, given that job distribution itself also incurs overhead.

The same trade-off happens with composition span distribution. If the number of spans is too high, the job distribution overhead is dominant and no performance improvement is possible, but if the number of spans is too small, parallelism is not exploited.

To determine the optimal number of spans, IndeX employs a heuristic algorithm to determine the number of spans in the composition. An application can enable this feature through the `nv::index::IConfig_settings` as in this example code. The number of spans can also be directly set by the application if the application has better knowledge of the task and system performance when this function is disabled. The example has command-line options to control this feature.

6.2.5.5 Large display and performance

This example sets a large canvas size (2560x1440) compared to other examples (512x512). You can change the canvas resolution using the `nv::index::app::canvas_infrastructure::IIndex_canvas::set_resolution`() method as demonstrated in cluster_performance_mainhost.cpp. Since the large resolution rendering has a impact to rendering time, this performance example will show what performance value is important. These results can also provide a guideline for the optimal system configuration.

6.2.5.6 Tips for performance measurement

Screen size change
: See function `setup_camera` in cluster_performance_mainhost.cpp. Changing the screen size may also require that other parameters should be changed, for example the aspect ratio.

Volume data size change
: See function setup_main_host() in cluster_performance_mainhost.cpp. The volume size can be changed with the `volume_size` variable. The cluster must have enough GPU memory, otherwise uploading volume data can become the bottleneck.

Performance statistics data change
: See variable `p_key` in function cluster_performance_mainhost.cpp/main() in cluster_performance_mainhost.cpp. These keys define which performance value is retrieved. For a description of these keys, see `nv::index::IPerformance_values`.

First frame overhead
: The performance data of the first frame is typically misleading. Since the rendering triggers the volume data creation, the first frame statistics also include the time that creation required.

6.3 Multi-view rendering

The NVIDIA IndeX software is able to render multiple view-port in the final canvas buffer. Here we show a few examples using this multiple view-port rendering capability.

You can review the NVIDIA IndeX's multi-view rendering technique through the following examples. The examples introduce viewports and the multi-view rendering technique. In the multi-view rendering technique of IndeX, a "view" is not only a camera view, but also includes a "view" of the database.

6.3.1 Overview of multi-view rendering

NVIDIA IndeX is capable of *multi-view rendering* which renders a scene from multiple views with minor scene modifications per viewport. A *viewport* in NVIDIA IndeX represented by the `nv::index::IViewport` class. An `IViewport` instance has a DiCE scope, a position on a canvas, and a size on a canvas. A DiCE scope represents a DiCE database view. The position and size represents a viewport position and size on the final rendering buffer.

One typical use case shows a single scene from multiple cameras in a single rendering canvas. Figure 6.2 (page 38) shows such a rendering result. In this result we can see four viewports, but this is only one scene, i.e., one volume, a few labels and lines. Each viewport's difference

is camera set up. The multi-view rendering can be used to help to understand the scene. Figure 6.3 annotates the each viewport of the Figure 6.2.

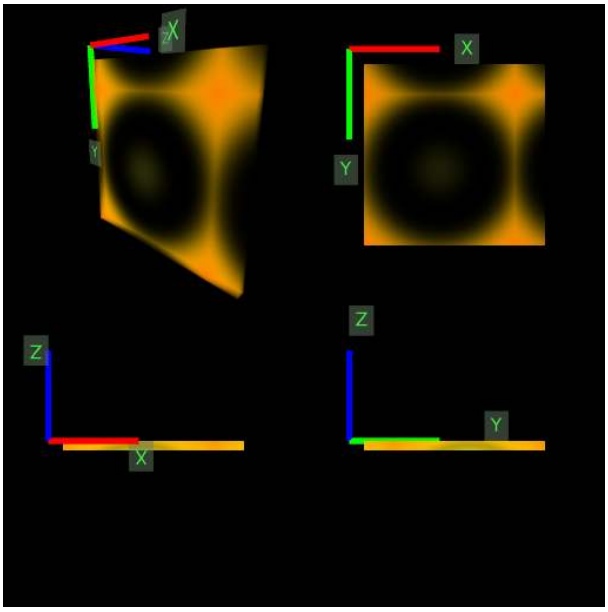

Fig. 6.2 - Example use case of multi-view rendering result

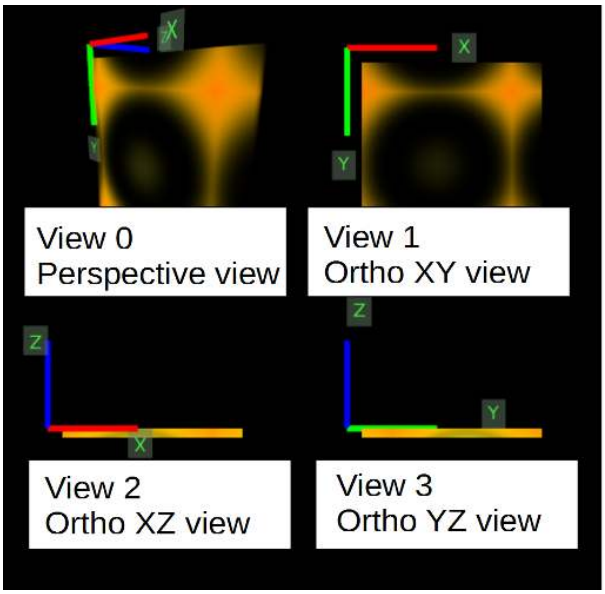

Fig. 6.3 - Views are annotated in a multi-view rendering result

6.3.2 Viewport

A viewport is one view from a camera on a canvas. Figure 6.4 shows the relationship between a camera (an eye), a viewport, and a canvas. What a camera sees is in the viewport, and the viewport is mapped on the canvas.

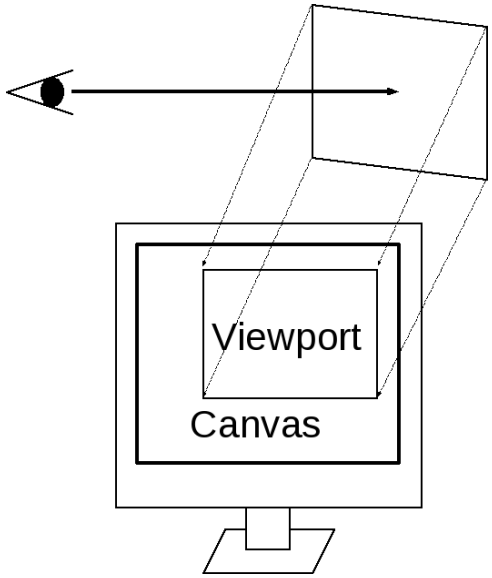

Fig. 6.4 - Viewport and canvas

In NVIDIA IndeX, this viewport is represented by `nv::index::IViewport` class. In Figure 6.4, the viewport only represents what the camera sees. However, the NVIDIA IndeX uses the DiCE database system, which supports different database views. Each NVIDIA IndeX `IViewport` is bound to a DiCE scope, thus the minor scene changes are possible for each viewport. (The details of "What is a minor scene change?" will be explained later.) For example, a usual NVIDIA IndeX scene has only one camera, and each viewport can differentiate its camera parameters. It is possible to have multiple cameras and using them for multi-view rendering, however, the DiCE database scope mechanism is the preferred method to render each viewport. In this way, the application doesn't need to switch or manage multiple cameras. The application can just change the scope binding to the viewport and can manipulate the camera parameters without knowing the other viewport's camera.

Figure 6.5 shows the geometry of IViewport. The coordinates of `IViewport` is based on the canvas (IIndex_canvas) pixel coordinates.

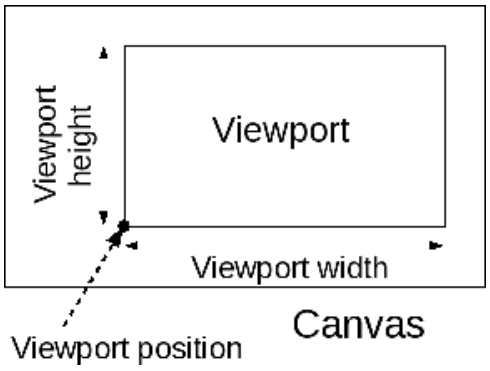

Fig. 6.5 - The geometry of IViewport

6.3.3 Multiple viewport arrangement

Figure 6.6 shows an example of multiple viewport arrangement. The viewports are set to the IViewport_list and send to NVIDIA IndeX via a render call. IViewport_list is a list of the IViewport. The order of the rendering of viewports is the same as the order of the IViewport_list holds viewports.

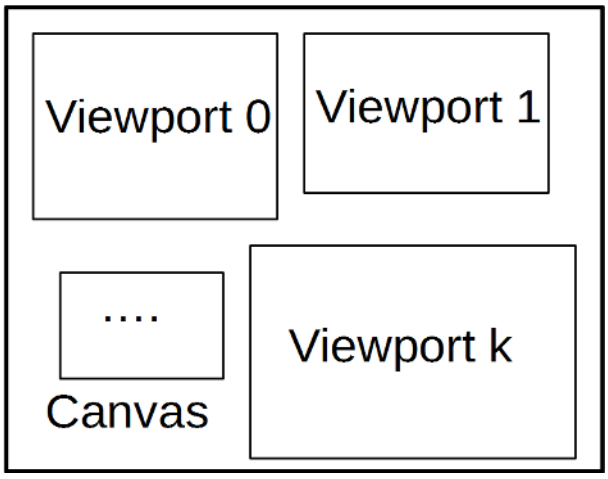

Fig. 6.6 - The arrangement of multiple viewports

The geometry of IViewport can be overlapped each other and also can be located partially outside the canvas. However, the application should take care the performance impact when the IViewport overlap. For instance, if multiple viewports completely overlap on the canvas, and only see the last viewport results, former viewport's rendering results are discarded. But still NVIDIA IndeX need to render the scene for all the viewports. The application sets how to arrange the viewports, so the application has the responsibility to avoid such cases.

6.3.4 Rendering into the viewport list

When the application has set up the viewports and localized the scene elements, the application calls the NVIDIA IndeX render call with a viewport list to render the scene with multiple viewports.

The NVIDIA IndeX renders each viewport. The order of rendering is the same order of viewports in the viewport list. The rendering results is sent back to the application span buffer.

6.3.5 Performance notice of multi-view rendering

The rendering of NVIDIA IndeX follows the viewports. Thus, the application should aware the viewport arrangement to get the optimal performance of NVIDIA IndeX. For example, if all the viewports are the same position and the same size, the rendering results are all overlap on the canvas. With a specific span buffer implementation, this produces the same result with the single viewport rendering, but, the rendering times became longer. A span buffer is implemented at the application side, thus, the application still can differentiate the rendering results, but the NVIDIA IndeX cannot know the detail of the application span buffer implementation. Therefore, NVIDIA IndeX only follows the viewport list that the application gave to the NVIDIA IndeX.

6.3.6 Picking of multi-view rendering

The NVIDIA IndeX supports picking operation in the context of multi-view rendering. There is a pick operation for multi-view rendering. The operation gets an viewport list and returns IScene_pick_results_list. This contains all the viewport pick test results.

See the multi_view_heightfield example code for the picking with multi-view.

6.3.7 Multi-view: volume

Source code

`multi_view_volume.cpp` (page 412)

6.3.7.1 Multi-view rendering with volume: overview

The example code in multi_view_volume.cpp shows how to use the NVIDIA IndeX multi-view capability. In this example, we have a volume and four labels in the scene. The basic flow of this example are following:

1. Initialize and start the IndeX service.
2. Create scene elements and set up the scene description hierarchy.
3. Create viewports and its scope. Append viewports to the viewport list. Arrange the viewports on the canvas.
4. Localize the scene elements with the scope associated with the viewport.
5. Render the scene with the viewport list.

You can find the details of initializing and starting the IndeX service (Step 1) in other examples (such as "Rendering a scene" (page 32)). Step 2 creates the scene elements and Step 3 creates all the viewports. Figure 6.7 shows the result of Step 3. You can see the scene set up of all the viewport are the same. Figure 6.8 (page 42) shows the viewport arrangements and its scope binding of this example.

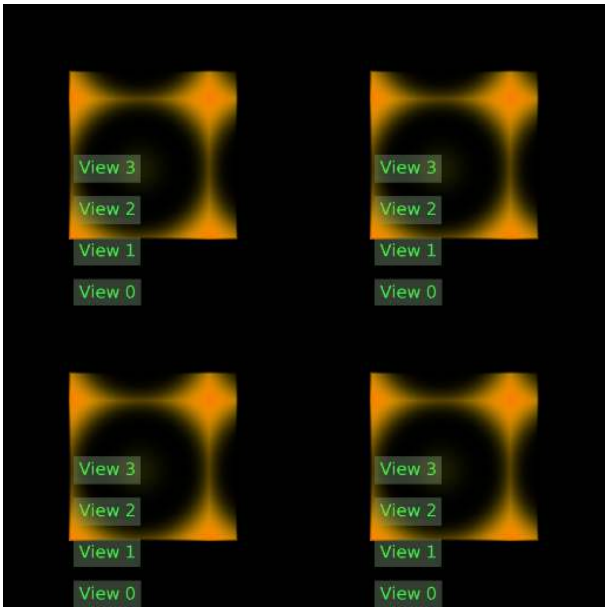

Fig. 6.7 - Basic scene set up for each viewport (after Step 3)

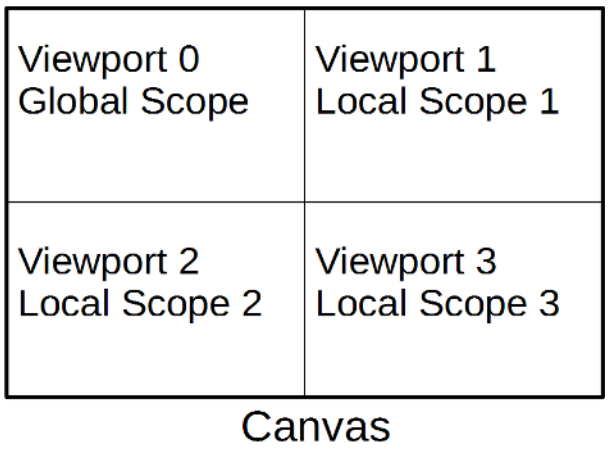

Fig. 6.8 - Viewport arrangement and scope binding

The localization of the scene elements are the important part of the multi-view rendering. (The details is in the function localize_scene_element() of the example code.) We use the DiCE scope mechanism to localize the scene element. This allows us to make a minor modification.

A minor modification is a scene element modification. Only non-distributed scene elements are allowed to do a minor modification. The distributed data are volume, heightfield, triangle mesh, and irregular volume in the current version of NVIDIA IndeX.

The camera and labels are not distributed data, thus we can modify them with the associated scope of each viewport. Figure 6.9 shows the multi-view rendering result of this example.

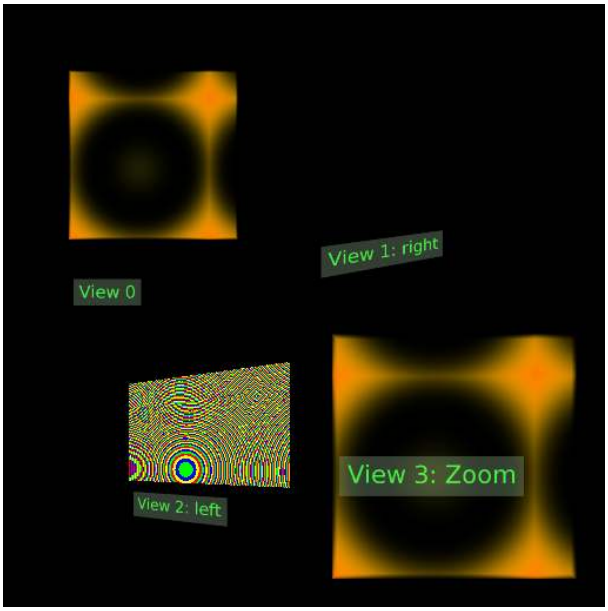

Fig. 6.9 - Multi-view rendering result of multi_view_volume example

Note the following:

- Each viewport enables only one label; the label text has been modified.
- Changes to camera parameters are dependent on the viewport.
- In viewport 1, the volume is disabled.

- The volume data (brick or voxels) is distributed. However, scene elements are not distributed data, which means that their state can be changed. For example, the ISparse_volume_scene_element instance, which manages the distributed data, is a scene element — not distributed data. Consequently, it is possible to change its state.
- As shown in viewport 2, colormap assignment is viewport dependent.

6.3.8 Multi-view: heightfield

Source code
`multi_view_heightfield.cpp` (page 332)

6.3.8.1 Example code: overview

The example code multi_view_heightfield.cpp shows how to use the NVIDIA IndeX multi-view capability. The basic structure of the example is similar with the multi_view_volume example. The multi_view_volume example documentation also describes multi-view rendering in general. If you are not familiar with multi-view rendering concept, please see Multi-view: volume (page 41).

In this example, we have a heightfield, two distribute compute technique, and lines for normal visualization. The basic steps of this example are the following (similar to the Multi-view: volume (page 41) case):

1. Initialize and start the IndeX service.
2. Create scene elements and set up the scene description hierarchy.
3. Create viewports and its scope. Append viewports to the viewport list. Arrange the viewports on the canvas.
4. Localize the scene elements with the scope associated with the viewport.
5. Render the scene with the viewport list.

You can find the details of initializing and starting the IndeX service (Step 1) in other examples (such as "Rendering a scene" (page 32)). Step 2 creates the scene elements and Step 3 creates all the viewports. Figure 6.10 (page 44) shows the result of Step 3. You can see the scene set up of all the viewport are the same. Figure 6.11 (page 44) shows the viewport arrangements and its scope binding of this example.

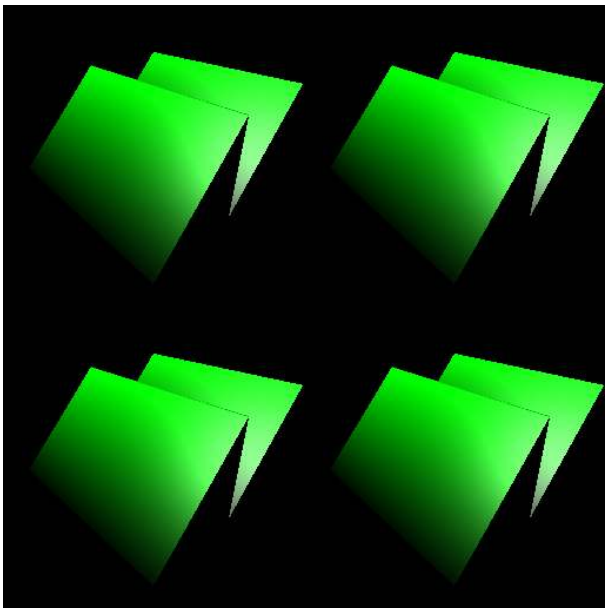

Fig. 6.10 - Basic scene set up for each viewport (after Step 3)

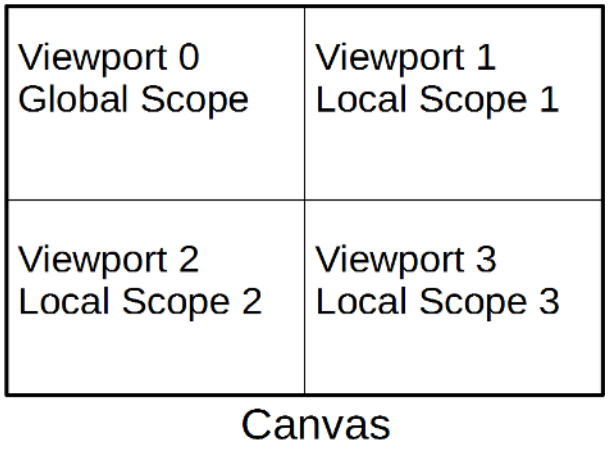

Fig. 6.11 - Viewport arrangement and scope binding

The localization of the scene elements are the important part of the multi-view rendering. (The details is in the function localize_scene_element() of the example code.) We use the DiCE scope mechanism to localize the scene element. This allows us to make a minor modification.

We localize the camera, compute techniques, and line set for the corresponding viewport. These scene elements are all non-distributed data. Please note, only the non-distributed data can be efficiently localized. The detail of distributed data or not is in the multi_view_volume example documentation.

Figure 6.12 (page 45) shows the multi-view rendering result of this example.

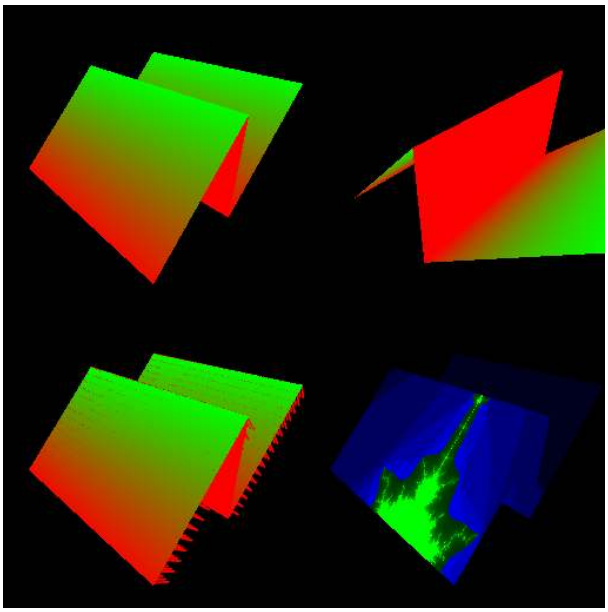

Fig. 6.12 - Multi-view rendering result of multi_view_heightfield example

You can see the color mapping is applied for the three viewports, and one heightfield has a compute texture (Mandelbrot). The lower left viewport additionally has a line set, which is visualizing the heightfield normal direction. Also, the upper right viewport has the different camera parameters. Please note, all the viewports share the scene and only one instance of heightfield in this example. This example demonstrate the compute technique and the line set as an annotation of the heightfield data.

6.3.8.2 Picking

This example code includes picking operation for multi-view. The function pick_test_call() uses NVIDIA IndeX scene query mechanism to pick the scene elements in the scene. You see the pick() operation gets an IViewport_list, and returns the IScene_pick_results_list. This IScene_pick_results_list contains a list of IScene_pick_results, which contains the picking information same as the single viewport case.

6.3.9 Multi-view: triangle mesh

Source code
`multi_view_trimesh.cpp` (page 391)

6.3.9.1 Example code: overview

This example code (multi_view_trimesh.cpp) shows how to use the NVIDIA IndeX multi-view capability. The basic structure of the example is similar with the multi_view_volume example. The multi_view_volume example documentation also describes multi-view rendering in general. (If you are not familiar with multi-view rendering concept,see Multi-view: volume (page 41)). In this example, we have a triangle mesh with a few labels in the scene. The basic flow of this example are following (this is the same as the multi_view_volume case):

1. Initialize and start the IndeX service

2. Create scene element and set up the scene description hierarchy.
3. Create viewports and its scope. Append viewports to the viewport list. Arrange the viewports on the canvas.
4. Localize the scene elements with the scope associated with the viewport.
5. Render the scene with the viewport list.

You can find the details of initializing and starting the IndeX service (Step 1) in other examples (such as "Rendering a scene" (page 32)). Step 2 creates the scene elements and Step 3 creates all the viewports. Figure 6.13 shows the result of Step 3. You can see the scene set up of all the viewport are the same. Figure 6.14 shows the viewport arrangements and its scope binding of this example.

Fig. 6.13 - Basic scene set up for each viewport (after Step 3)

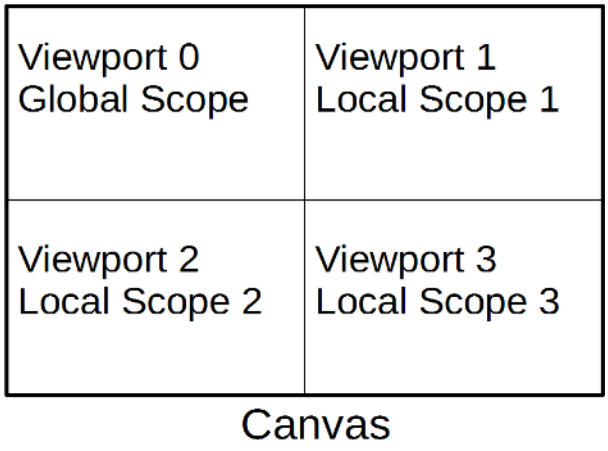

Fig. 6.14 - Viewport arrangement and scope binding

The localization of the scene elements are the important part of the multi-view rendering. (The details is in the function localize_scene_element() of the example code.) We use the DiCE scope mechanism to localize the scene element. This allows us to make a minor modification.

We localize the labels, the camera, and the materials for the corresponding viewport. These scene elements are all non-distributed data. Please note, the color of the triangle mesh alters, but this is a material of the triangle mesh. This is not the triangle mesh geometry. Since the geometry of the triangle mesh is a distributed data, NVIDIA IndeX can not have a minor modification of the distributed data. When a distributed data is modified, all the view see the modification. Or the application needs a copy of the distributed data. This means the modification of a distributed data is possible, but it is an expensive operation.

Figure 6.15 shows the multi-view rendering result of this example.

Fig. 6.15 - Multi-view rendering result of multi_view_shape example

You can see the camera change, the label text change, and the triangle mesh color change. The color of the triangle mesh is controlled by a material in the scene and a material is not a distributed data, so we can have a minor modification of a material. Please notice all the viewports show the exactly same geometry of the triangle mesh.

6.3.10 Multi-view: shape

Source code

`multi_view_shape.cpp` (page 356)

6.3.10.1 Example code: overview

This example code (multi_view_shape.cpp) shows how to use the NVIDIA IndeX multi-view capability. The basic structure of the example is similar with the multi_view_volume example. The multi_view_volume example documentation also describes multi-view rendering in general. (For more information about multi-view rendering, see Multi-view: volume (page 41).) In this example, we have many kind of shapes provided by NVIDIA IndeX. The basic flow of this example are following (this is the same as the multi_view_volume case):

1. Initialize and start the IndeX service.
2. Create scene element and set up the scene description hierarchy.

3. Create viewports and its scope. Append viewports to the viewport list. Arrange the viewports on the canvas.
4. Localize the scene elements with the scope associated with the viewport.
5. Render the scene with the viewport list.

You can find the details of initializing and starting the IndeX service (Step 1) in other examples (such as "Rendering a scene" (page 32)). Step 2 creates the scene elements and Step 3 creates all the viewports. Figure 6.16 shows the result of Step 3. You can see the scene set up of all the viewport are the same. Figure 6.17 shows the viewport arrangements and its scope binding of this example.

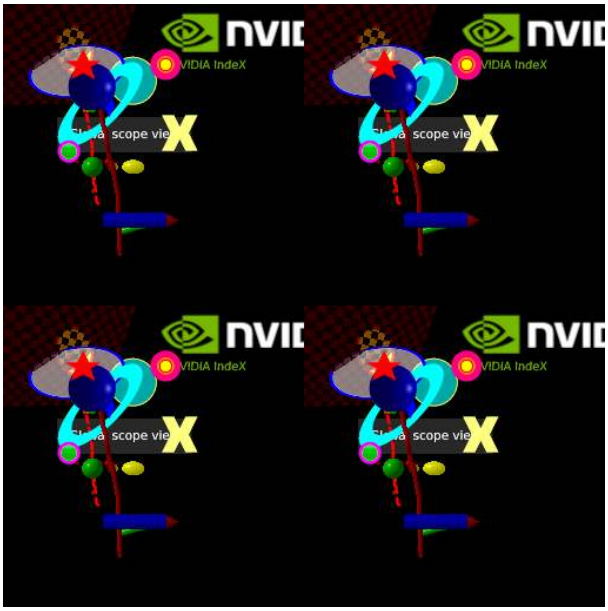

Fig. 6.16 - Basic scene set up for each viewport (after Step 3)

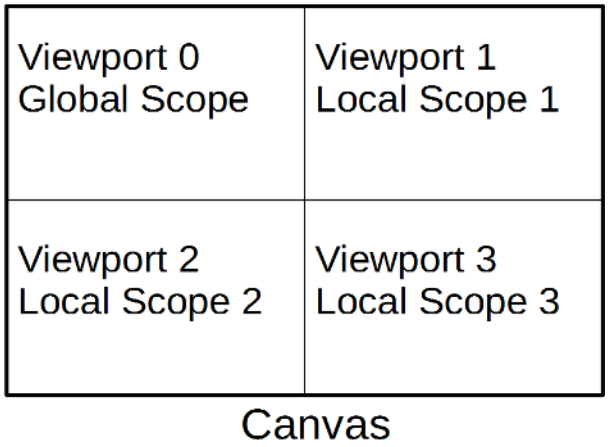

Fig. 6.17 - Viewport arrangement and scope binding

The localization of the scene elements are the important part of the multi-view rendering. (The details is in the function localize_scene_element() of the example code.) We use the DiCE scope mechanism to localize the scene element. This allows us to make a minor modification.

We localize shapes for the corresponding viewport. These scene elements are all non-distributed data.

Figure 6.18 shows the multi-view rendering result of this example.

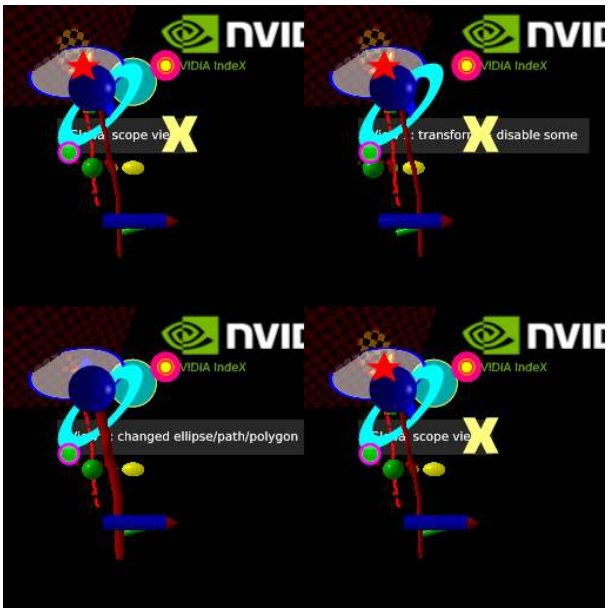

Fig. 6.18 - Multi-view rendering result of multi_view_shape example

Some of the scene elements are altered or disabled, depending on the viewport. For example:

- The label text displayed at the center of the scene is viewport dependent. This is an important use case for data annotation.
- Various polygon shapes are displayed in some viewports but not others.

7 Manipulating scene elements

7.1 Scene description attributes

Source code

`scene_description_attribute.cpp` (page 457)

7.1.1 Overview of scene description attribute

Attributes are part of the hierarchical scene description provided by IndeX. The attributes represent common means to define and parameterize the visual appearance of shapes and large-scale data sets, i.e., their rendering. Furthermore, attributes can be used to store meta information in the hierarchical scene description as well for later evaluation. That is, attributes can steer both the rendering as well as the processing and evaluation of data. Attributes are part of the hierarchical scene description.

In this example, we use an attribute to set an rendering attribute to shapes. A simple example is a depth offset. A depth offset parameter is attached to a rasterized shape and the IndeX rendering system aware these shapes have these attribute. For instance, when a 3D line and a 3D plane share the co-planer position, then the numerical problem causes inconsistent look result. However, if there is a depth offset has been set by the application as a rendering hint, the renderer renders these shapes according to the attached hint.

7.1.2 Problem when shapes have the same depth value

When shapes share the save depth value from the camera, any rendering system should determine which object is actually visible from the camera. In the real world, there is no such case that two physical tangible objects share the same position. However, in a rendering system, some shapes are not solid objects, i.e., a triangle, any rasterized shapes. This means the application can configure the scene that several shapes share the same depth value from the camera.

Figure 7.1 (page 51) shows the shapes share the same z-values. The points are rendered differently because of the visibility determination algorithm can not distinguish the depth difference. These are undefined behavior.

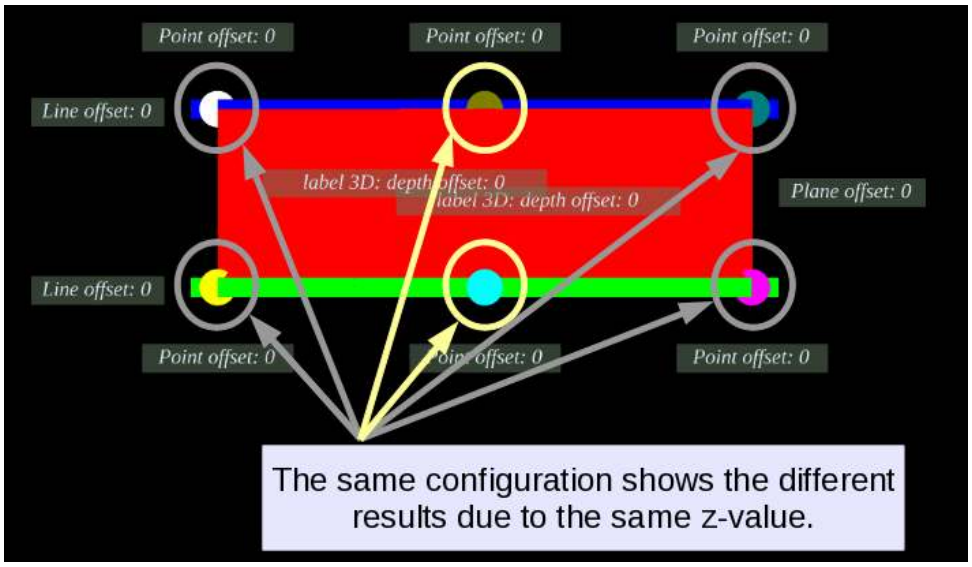

Fig. 7.1 - Shapes located at a co-planar position

One way to solve this is by a scene construction. Assume all the shapes are solid shape and then, the application can create the scene as physically correct way.

The second way is we explain here that given extra attributes by the application to the shape and the renderer recognize these attributes to determine the z-values to compute visibility. We have the following three visibility determination attributes.

- Depth offset attribute
- Painter's algorithm order attribute
- Z-test attribute

7.1.3 Depth offset solution

One solution is giving these shapes a depth offset attribute. When the depth from the camera is calculated, this offset attribute is considered which shape is visible. Figure 7.2 shows the same coordinates of shapes as in Figure 7.1, but given the offset attributes.

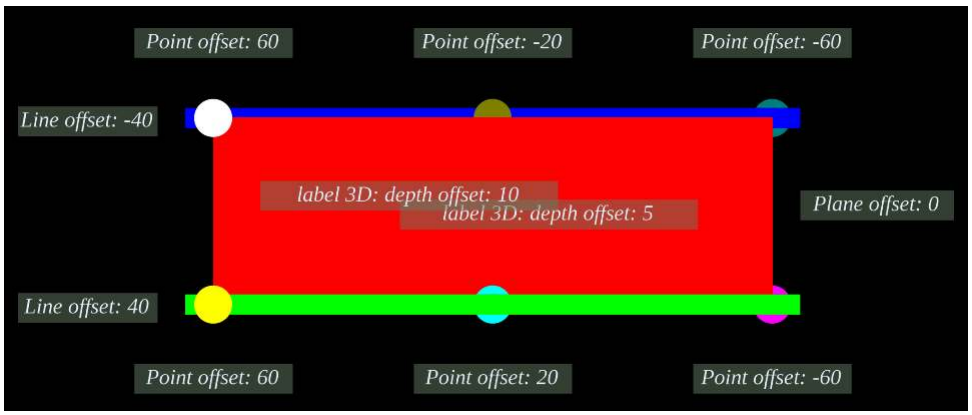

Fig. 7.2 - Shapes located at a co-planar position, but each has depth offset attribute

The application need to give the depth offset to IndeX as the `nv::index::IDepth_offset` attribute in the hierarchical scene description. The depth value depends on the shape size and the distance of the camera. For example as an extreme case, if the distance from a camera to a

shape is too large and the depth offset is too small, then the renderer can not determine which shape is visible.

Figure 7.3 shows the part of this depth offset example hierarchical scene description. The group node G1 contains two groups, G2 and G3. G2 is the group of plane shape. G2 contains a depth offset attribute A1 for the plane S1 and a plane S1 (S1 is the red plane shown in Figure 7.2 (page 51)) A1 contains depth offset value 0. G3 is the group of lines in this example. G3 has two lines and its attributes. A2 is a depth offset attribute for a Line S2 (S2 is the green line shown in Figure 7.2 (page 51)) and A3 is a depth offset attribute for a Line S3 (S2 is the blue line shown in Figure 7.2 (page 51)) A2 and A3 contains offset value 40 and -40, respectively.

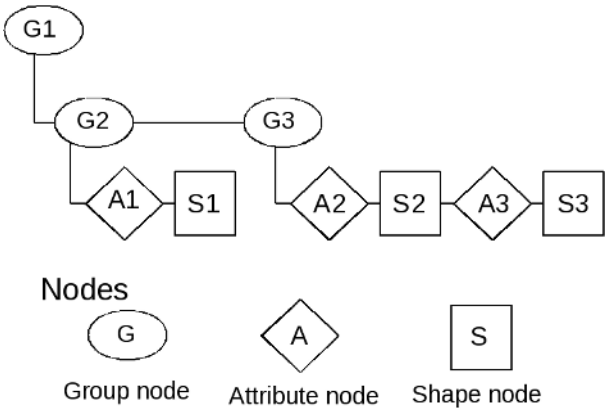

Fig. 7.3 - Example depth offset hierarchical scene description structure

7.1.4 Depth comparison operator solution

The depth offset needs one parameter value for an offset to solve numerical precision problem. This means application need to give an offset value depending on the scene scale. For example the object distance from the camera must be considered. Another solution is when the renderer detected the same depth value, we can give an operator to the renderer for comparing the depth value. Please notice this is a complementary solution with the depth offset attribute, there is a case it is useful and there is a case this is not effective. These are depends on the scene scale since this is for solving a numerical problem.

IndeX provide two depth comparison operators, Depth_test_mode::`TEST_LESS` and Depth_test_mode::`TEST_LESS_EQUAL`. By default, this operator is Depth_test_mode::`TEST_LESS_EQUAL`. This means a fragment produced by the renderer ends up in the depth buffer if and only if the depth value is smaller than or equal to (`<=`) the depth value already stored in the depth buffer at that 2D sample location. Depth_test_mode::`TEST_LESS` operator attribute computes the depth with smaller than (`<`) operation. An instance of the depth test attribute applies to all successive 3D primitives defined in the hierarchical scene description until overwritten by another successive attribute instance.

Figure 7.4 (page 53) 4 shows the rendering result with depth test attribute. In this scene, there are four instances of `nv::index::ISphere` and four instances of `nv::index::IPlane`. On the left side, the two upper spheres have a red and a green color, however they share the center and the radius. Therefore the rendering result shows only one sphere, at the upper case, the green one. The lower two has the same configuration, but only the red sphere is shown. On the right hand side, we see four planes, all the upper two planes and the lower two planes

have the same depth value, but the depth test attributes differ. In this way, an application can give a hint to the renderer that how the depth comparison will be done.

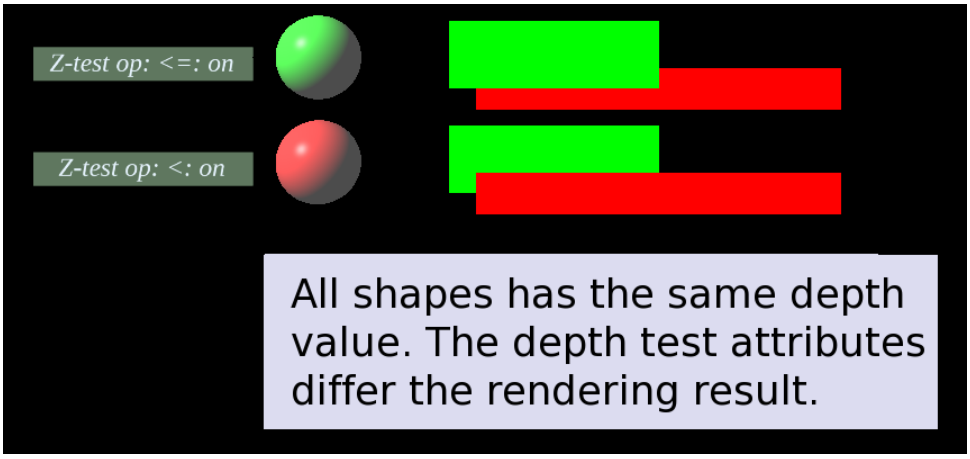

Fig. 7.4 - Example rendering result with depth test attribute node

Figure 7.5 shows the hierarchical scene description node structure of a part of Figure 7.4. This part shows the two spheres at the upper left side. The following are the details for each node.

`A1` – Depth test attribute with operator `<=`
`M1` – Material for color (red)
`S1` – A sphere
`A2` – Depth test attribute with operator `<=`
`M2` – Material for color (green)
`S2` – A sphere

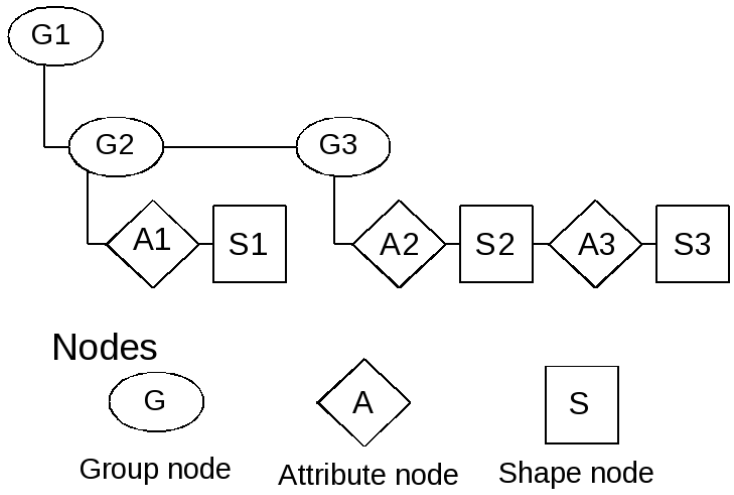

Fig. 7.5 - Example hierarchical scene description structure with depth test. This is the part of the four sphere's upper two spheres in Figure 7.4. There are four spheres in the scene, but each two shares position and radius, so only two are seen.

Note: The per object depth test control is applied to 3D objects only.

7.1.5 Painter's algorithm solution

If the 2D shape reference positions are the same in the 3D space, we can have a virtual plane defined by the reference position and the camera. In this specific situation, IndeX provides a classical pseudo depth visualization algorithm, so called painter algorithm. The application can provide the drawing priority via `nv::index::IRendering_order` attribute node. When the priority number is smaller, then the priority to be shown is higher. 0 is the highest priority. When the same priority shapes overlap, then which shape will be shown is undefined.

Figure 7.6 shows the example with polygons. The left side three red, green, blue color rectangles share the same reference position in the 3D space. But they are assigned different priority attribute. The right side three rectangles also share the same reference position. As you see in Figure 7.6, higher priority polygon was drawn to the front: the red rectangle in the left side, the blue rectangle in the right side.

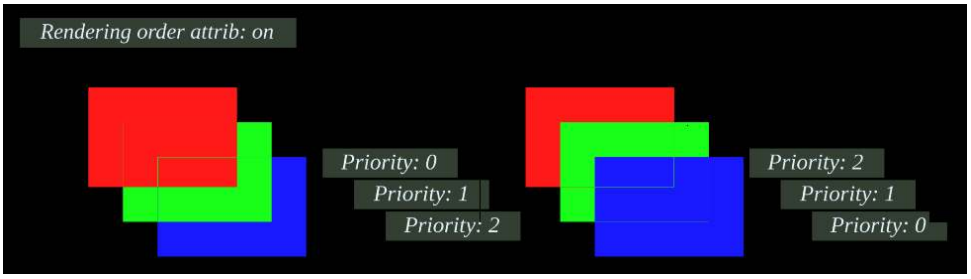

Fig. 7.6 - Example rendering result with painter's algorithm

7.2 Accessing distributed data

Source code

`distributed_sparse_volume_data.cpp` (page 259)

7.2.1 Overview

The previous examples showed how to create volume (page 10) and heightfield (page 12) data and how to add them to the scene. This example shows how to access the data after it is distributed to the cluster.

7.2.2 Distributed computing concepts

NVIDIA IndeX distributes data in a cluster-based host environment through the use of a custom distribution scheme. As a result, data cannot be accessed or edited locally on a single machine. NVIDIA IndeX provides general conceptual paradigms that allow the locality of the distributed data to be specified as well as accessed and edited.

The data locality, access and editing paradigms enable the implementation of arbitrary compute algorithms, such as auto-tracking techniques, that are primarily based on data hosted by the rendering cluster. In addition, user-defined query methods, such as snapping techniques or data export schemes, can be implemented efficiently. The general distributed computing paradigms typically rely on a 3D query region (implemented as a 3D bounding box), defined for use in applications by the application developer.

Data locality

The data locality paradigm provides an application with the identify of the host in the cluster on which the actual data of a distributed data set is stored. A single cluster host

stores zero or more subsets of the entire data. Data locality also specifies the corresponding 3D bounding box of each subset. A unique identifier provides access to each 3D bounding box and the subset it contains.

The invocation of parallel and distributed compute tasks is a common use case that requires data locality. Through data locality, the compute tasks can be directed to the hosts that actually store the distributed data, allowing local computation on data without transferring it across the network.

Distributed data access

In the distributed data access paradigm, all data in the query bounding box are transferred to the cluster machine that initiated the query. The queried data are then assembled to form a single subset of the data set, forming a regular 3D volume from their geometric union. This mechanism allows an application developer to inspect or export data on the calling machine. Note that 3D regular volume data sets can be very large — on the order of multiple terabytes — so an application developer should limit bounding box queries to the capability of available memory resources.

Distributed data editing

The distributed data editing paradigm enables distributed editing of the voxel values of a regular 3D volume data set. The application can implement an editing or computing task that can be applied to each voxel brick in the cluster environment. In combination with data locality (which allows directing the computation to the machines that store the brick data), the editing paradigm is a powerful tool that benefits from data distribution, enabling distributed and parallel data processing. NVIDIA IndeX interfaces not only support the processing of the data in main memory but also the processing of the data in GPU memory.

7.2.3 Scene setup

In NVIDIA IndeX, the visibility of volume data is affected by the camera. If the camera never sees the volume data, IndeX decides it is not necessary to allocate the resource for them. Therefore, we need a scene setup (such as IScene) and a camera setup.

The scene used in this example consists of a volume and a heightfield which are both created using the synthetic data generation methods introduced in the previous examples. The region of interest is chosen so that it includes the entire scene.

Since the purpose of this example is data access and not visualization, a dummy camera with default parameters is sufficient. However, to ensure that data is actually loaded, the data import needs to be triggered first. This is achieved using `nv::index::IConfig_settings::set_force_data_upload` followed by a render call. The rendering result is ignored, but with the configuration applied, it will load all data, even if it is not visible with the current camera.

7.2.4 Data access and editing

After all data is loaded, analyze_sparse_volume_data() accesses the sparse volume data and runs some simple data analysis on it. To determine which data to retrieve for the sparse volume, the function defines a bounding box. To print the cluster nodes where the data is located, the function uses retrieve_sparse_data_locality().

For the sparse volume scene element, `nv::index::IDistributed_data_access_factory` and `nv::index::IDistributed_data_access` instances are created. The sparse volume tag is given to the `nv::index::IDistributed_data_access_factory` instance. For this reason, the `nv::index::IDistributed_data_access` instance should be a `nv::index::ISparse_volume_subset` instance. The subset data descriptor of the sparse volume is contained in this `nv::index::ISparse_volume_subset` instance. The subset data descriptor provides the information required to access the data bricks of the sparse volume including the dimensions of bricks, attribute data of bricks, and brick data buffers. Because the sparse volume data is owned by the `nv::index::ISparse_volume_subset` instance, it will be freed automatically.

To enable write operations, edit_sparse_volume_data() gathers information that identifies the cluster nodes where the data is located. This information is passed to the `nv::index::IDistributed_data_job` instance, which modifies the data. In this example, the Sparse_volume_set_voxel_value instance performs this operation. It sets all voxel values in the given bounding box to a fixed value. The compute algorithm is a fragmented job, so it is started by calling an `mi::neuraylib::IDice_transaction::execute_fragmented` method.

To export the modified data, Distributed_sparse_volume_data::export_sparse_volume_data() is used. The approach is similar to that used with analyze_sparse_volume_data() — just a different bounding box. The data in this example is averaged, but it could also be written to a file.

The functions analyze_heightfield_data() and edit_heightfield_data() are very similar to their sparse volume counterparts. The major difference is the use of a two-dimensional bounding box for accessing heightfield data rather than the use of a three-dimensional bounding box for accessing sparse volume data.

7.2.5 Running the example

To run the example on a single machine, use the `run_example.sh` script.

To test the distributed nature of the data access, you must run the example in a cluster. This test works in combination with the technique described in "Cluster rendering" (page 33). Be sure to start this example on all selected cluster nodes except the last one. On the last cluster node, start `example_volume_data_access`; it will connect to all the other nodes, distribute the data to the machines, and then perform the data access operation.

7.3 Automatic normal recalculation of a heightfield

Source code

`normal_recalculation.cpp` (page 436)

7.3.1 Overview of automatic normal recalculation of a heightfield computation

The IndeX library can provide a default automatic normal recalculation of a heightfield computation. When an application launches a compute task, and edits the height values of a heightfield, the application can choose either the automatic normal recalculation or a user defined normal recalculation. This example shows how to use the automatic normal recalculation.

Figure 7.7 (page 57) and Figure 7.8 show the output of the example code.

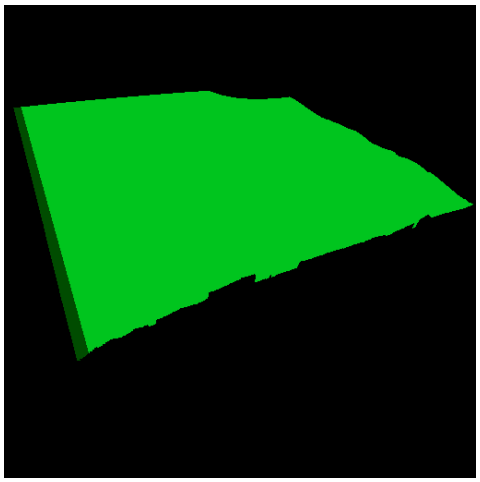

Fig. 7.7 - The user set the constant normal values to the heightfield

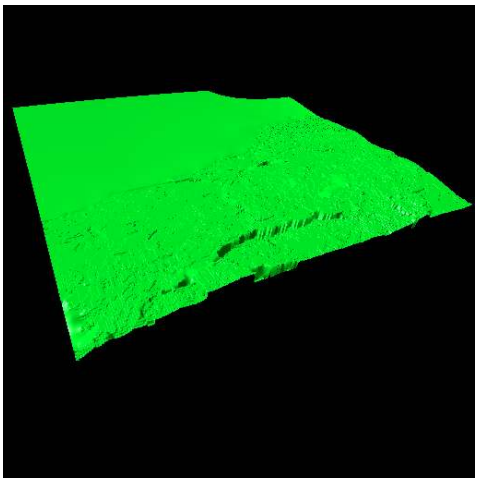

Fig. 7.8 - The automatic normal recalculation was performed

7.3.2 Editing heightfield by a compute task

The IndeX library manages massive data, such as volume and heightfield, in the distributed fashion. To edit these data, the application needs to launch a edit compute task. A edit compute task is a DiCE fragmented job, therefore, this task will be performed in parallel in a cluster. How to access to the data was shown in the volume (page 10) example. We use the same mechanism to access to the data via the `nv::index::IRegular_heightfield_compute_task` task.

In the example code, you can find two classes for computing the height values and normal values of a heightfield.

`Example_heightfield_operation`
: This is a DiCE fragmented job and also a `nv::index::IDistributed_data_job`. DiCE performs this fragmented job in parallel in the cluster using the DiCE fragmented job mechanism. Then this launches the heightfield compute tasks. That is the Example_height_fill_task.

`Example_height_fill_task`
: This is a compute task for the heightfield. IndeX library launches this task with the compute bounding box and the data. The most important method of this task is Example_height_fill_task::compute. You can find two overloaded `compute()` methods in the example. The difference of them will be explained in the next section.

7.3.3 Choosing the type of normal recalculation

To recalculate the normals for the heightfield, an application can use a method provided by IndeX. `nv::index::IRegular_heightfield_compute_task::compute` with three arguments performs the automatic normal recalculation after the height values are edited. The normal values adjacent to the elevation values will be averaged based on the slope of the surface in the neighborhood of an elevation value. Figure 7.8 shows the automatic normal recalculation result. One note is performance optimization. Since the normal recalculation is always performed on the entire active heightfield, when the application only changed a small

region of heightfield, the normal recalculation still performed on the entire active heightfield. Usually the application knows what was changed, the application could have a chance to optimized this computation.

A custom recalculation function can also be provided by the application. Using the second variant of the `nv::index::IRegular_heightfield_compute_task::compute` with four arguments, the application can pass the user defined normals to IndeX library. Figure 7.7 (page 57) shows when the flat normals are passed to the IndeX library.

7.3.4 Negative height value handling

The IndeX library can handle negative height field value. Holes in a height field are marked by `NaN`. This example negates the height values of the input heightfield, thus the negative height value were also rendered.

7.4 Manipulating point sets

Source code

`dynamic_point_set.cpp` (page 290)

7.4.1 Overview of dynamic manipulation of point set shape

The IndeX library can dynamically create, move, and delete scene elements. This example shows how to create them, edit or move them, and how to delete them at runtime.

Figure 7.9 (page 58) shows the example output using options `-resolution_x 512` and `-resolution_y 512`.

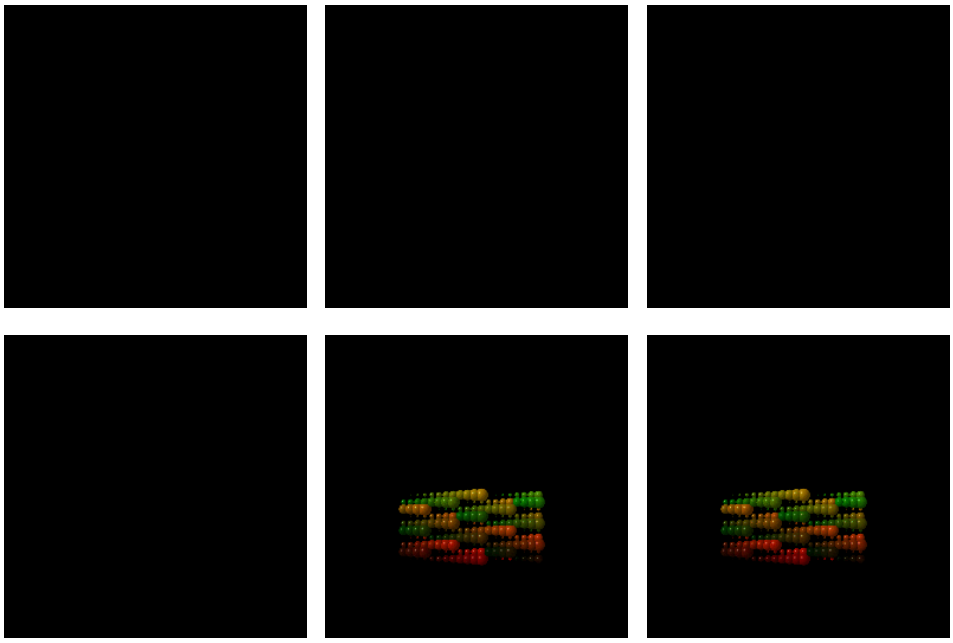

Fig. 7.9 - A dynamic manipulation of point set rendering result animation

7.4.2 Create point set shape

See the "Point sets" (page 19) example.

7.4.3 Edit point set shape

The application needs to edit the point set shape stored in the database. This is demonstrated in the example code by the `translate_point_set` function. The edit operation is an expensive operation so the application should only edit a database element if necessary.

7.4.4 Delete point set shape

A point set shape instance is a scene element that is managed in the scene through the session. The application therefore typically first accesses the session to provide for access to the scene. However, if a scene will be modified it cannot be accessed in read-only mode. This is shown in the example code by the `delete_scene_element` function. The scene element is managed through the scene, but the instance itself is managed through the DiCE database. The application therefore needs to call `delete_scene_element` on the scene and then call `mi::neuraylib::IDice_transaction::remove` on the dice transaction.

7.5 Manipulating planes

Source code

`dynamic_plane.cpp` (page 280)

7.5.1 Overview of dynamic manipulation of plane shape

The IndeX library can dynamically create, move, and delete scene elements. This example shows how to create them, edit or move them at runtime.

Figure 7.10 (page 59) shows the output of the example code.

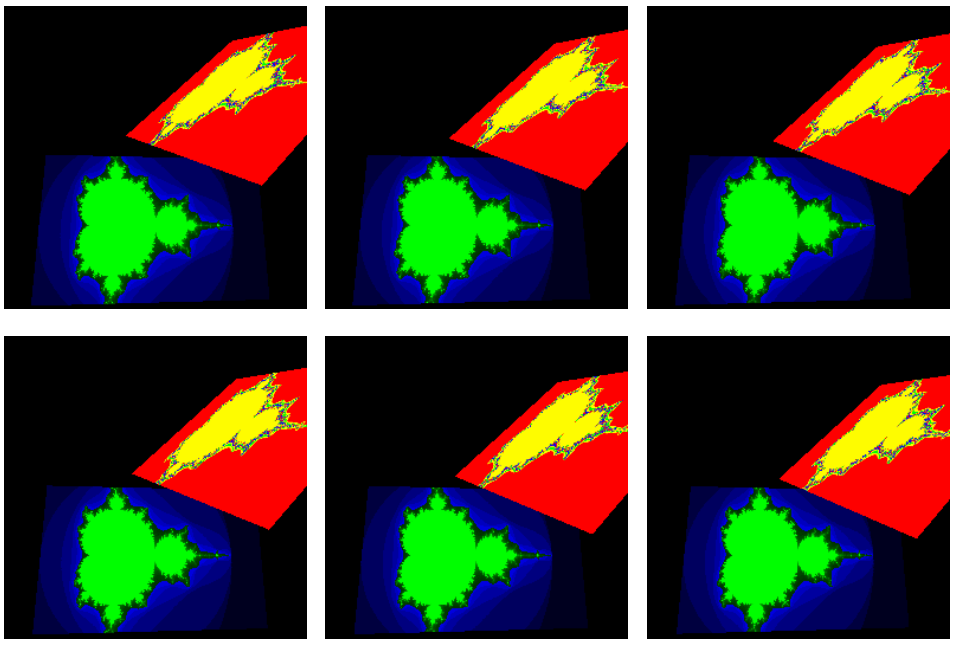

Fig. 7.10 - A dynamic manipulation of planes rendering result animation

7.5.2 Create planes

Please see the "Planes" (page 16) example. So that this example can move two planes, the `create_planes` function keeps these tags in the `App_data` instance.

7.5.3 Moving planes in the scene

The `move_planes` function moves the plane's position in the opposite direction to the plane's normal. The tags of the planes are maintained in the application data `App_data` instance. Through these tags, the example acquires the plane instances and edits their position in each frame.

7.6 Constructing hierarchical scenes

Source code

`build_scene_description.cpp` (page 68)

7.6.1 Introduction to the hierarchical scene description

IndeX provide a *hierarchical scene description* for shapes. A hierarchical scene description is a directed acyclic graph in which node is an `nv::index::IScene_group` or an `nv::index::IScene_element`. An `nv::index::IScene_element` can only be a leaf node. The graph structure is formed by appending `nv::index::IScene_group` or `nv::index::IScene_element` instances. As a scene description is created, the application must maintain the acyclic structure of the scene graph.

Figure 7.11 shows an example of a hierarchical scene description structure:

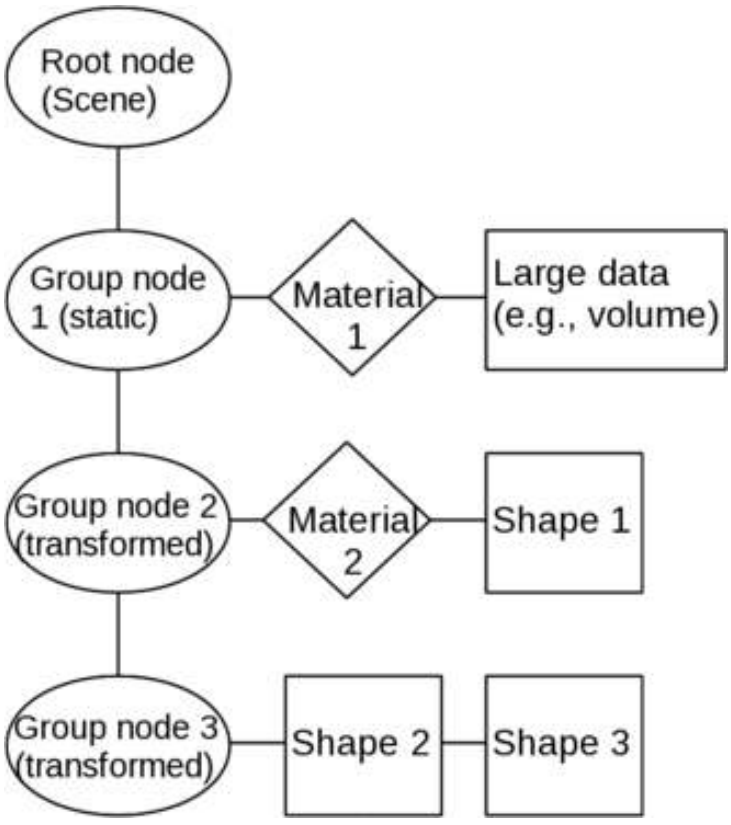

Fig. 7.11 - An example of a hierarchical scene description structure

The graphical shapes in Figure 7.11 (page 60) represent three types of data in a scene:

Ellipse
Group nodes of type `nv::index::IScene_group` (for example, `nv::index::ITransformed_scene_group`, `nv::index::IStatic_scene_group`, `nv::index::IShape_scene_group`, etc.)

Diamond
A material of type `nv::index::IMaterial` (for example, `nv::index::IPhong_gl`, etc.)

Rectangle
A scene element of type `nv::index::IScene_element` (for example, `nv::index::ILine_set`, `nv::index::IPoint_set`, `nv::index::IRegular_heightfield`, `nv::index::ISparse_volume_scene_element`, etc.)

The root node in any scene graph is always provided by the IndeX library. Three different scenarios are shown below the root node:

Group node 1: Large data scene element
A large data scene element must be inserted under an `nv::index::IStatic_scene_group` instance (Group node 1 in the figure) for special data handling. Typically, a material node (Material 1 in the figure) is followed by an object containing a large data set, for example, a volume or heightfield node.

Group node 2: Material-aware scene element
Shapes like the three-dimensional `nv::index::IPoint_set` are *material-aware* shapes. These shapes need to be structured under an `nv::index::ITransformed_scene_group` instance (Group node 2 in the figure) and an `nv::index::IMaterial` is needed (Material 2 in the figure). In the structure depicted by the diagram, Shape 1 is affected by the Material 2 attribute.

Group node 3: Material-independent scene element
Shapes like `nv::index::ILine_set` are *material-independent* shapes. In general, rasterizing shapes are independent of a material. Because these shapes don't need materials, they are attached directly under a group node. Shapes 2 and Shapes 3 in the figure are an example of this structure.

7.6.2 Structuring a scene

The API of the IndeX library exposes the interfaces class `nv::index::IScene_group` that provides the functionality of scene groups for structuring scenes. Each of the children is evaluated in linear order. The IndeX library distinguishes between two kinds of scene groups:

`nv::index::IScene_group`
An `nv::index::IScene_group` allows for an grouping of attributes, shapes, further scene groups.

`nv::index::ITransformed_scene_group`
An `nv::index::ITransformed_scene_group` represents a regular scene group but in addition defines an arbitrary transformation, which effects all its children. All transformations for composing a scene are defined through transformed scene groups. An `nv::index::ITransformed_scene_group` prevents adding an instance of

`nv::index::IScene_group` to avoid transformations applied to distributed large scale data.

7.6.3 Higher-level 3D shapes and attributes

Figure 7.12 shows an output image generated using the example code in `build_scene_description.cpp` (page 68)

.

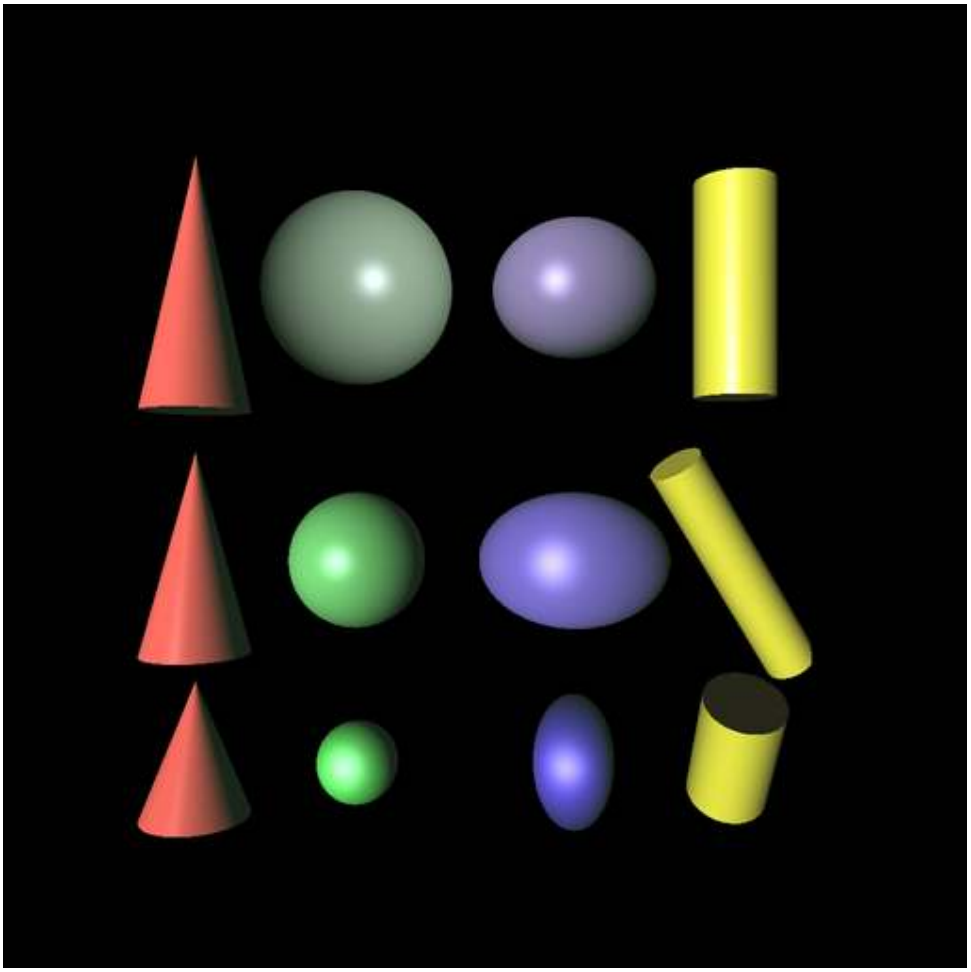

Fig. 7.12 - The higher-level 3D shapes. Their attributes and the light sources are composed within a hierarchical scene.

8 XAC — Accelerated Compute Interface

The “IndeX Accelerated Compute (XAC) Interface” section in the API documentation enables programmers to add real-time compiled sampling programs into the IndeX volume rendering pipeline. XAC programs are written in the CUDA programming language.

The data distribution, parallelization, and management is handled by IndeX while the XAC interface can directly modify the rendering output produced by Index.

8.1 Sample program overview

IndeX performs a front-to-back ray casting procedure for each rendered frame of a scene. A user-defined sampling program can be executed at each step of a ray generated by the ray caster.

There are two types of sampling programs:

Surface programs
Based on class `nv::index::ISurface_sample_program`

Volume programs
Based on class `nv::index::IVolume_sample_program`

Volume sampling programs are called for each step traversing through a volume during the ray casting procedure. In contrast, surface programs are only executed when a surface-based scene element is hit.

Listing 8.1 shows the basic structure of a volume sampling program.

Listing 8.1

```
using namespace nv::index::xac;

class Volume_sample_program
{
    NV_IDX_VOLUME_SAMPLE_PROGRAM

public:
    NV_IDX_DEVICE_INLINE_MEMBER
    void initialize()
    {
        ...  Initialize the program
    }

    NV_IDX_DEVICE_INLINE_MEMBER
    int execute(
```

```
        const Sample_info_self& sample_info,
              Sample_output&    sample_output)
    {
        ...  Perform computations
        return NV_IDX_PROG_OK;
    }
};
```

In Figure 8.1, sampling is performed at various positions along the course of a rays traversal from the camera into the scene:

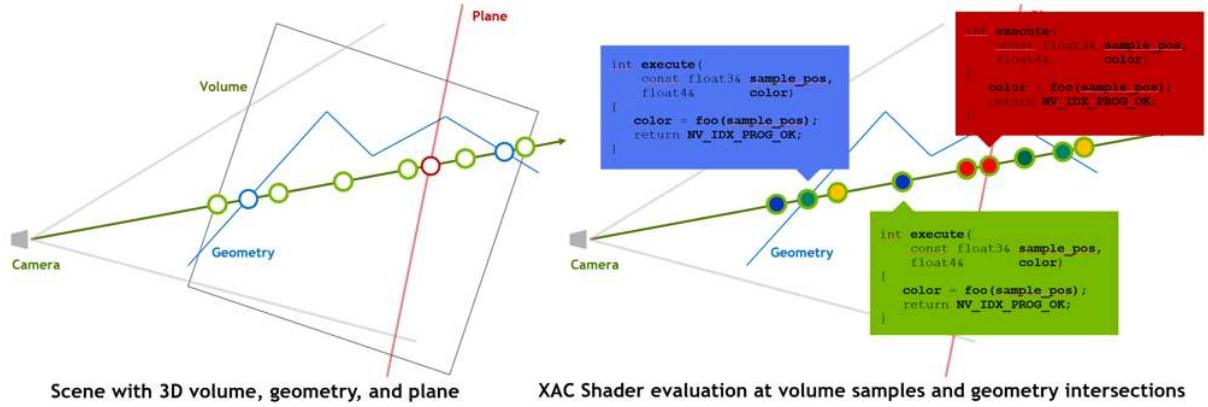

Fig. 8.1 - XAC code execution in volume and surface programs occurs when sampling a volume along a ray or when a ray intersects a surface.

Figure 8.2 shows a volume sample program in the code editor:

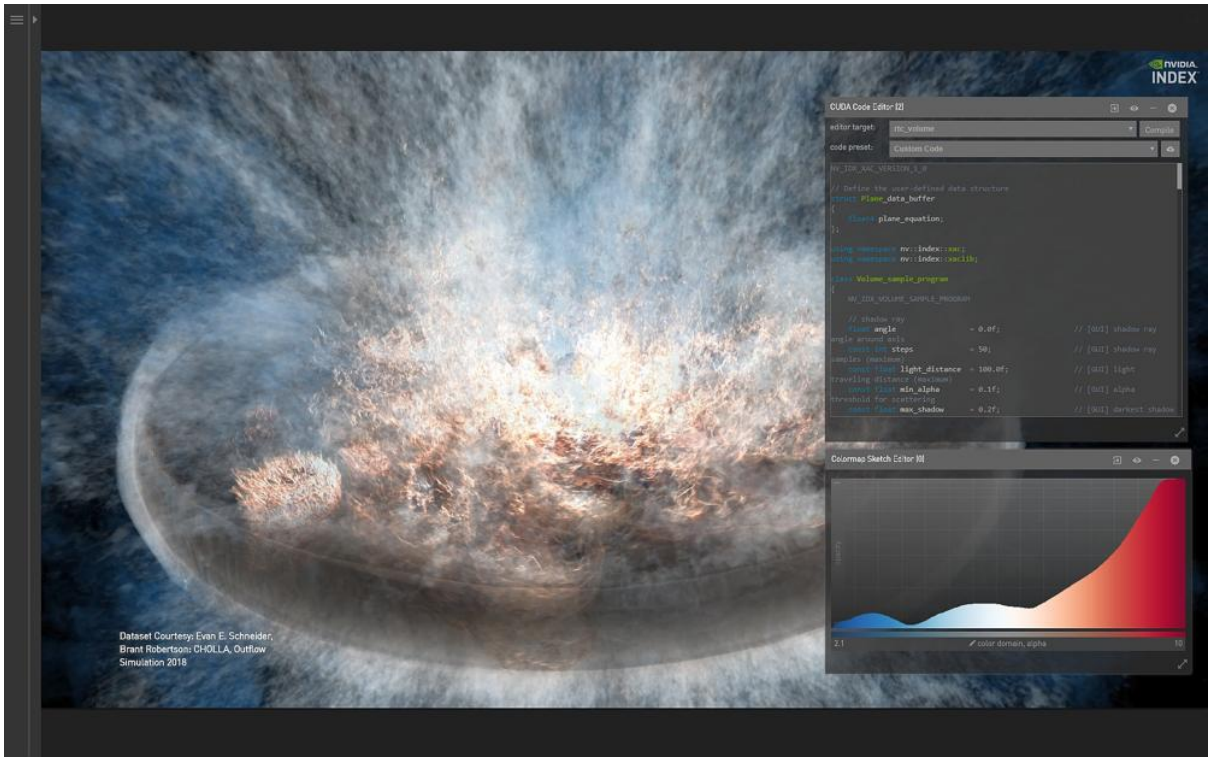

Fig. 8.2 - Code editor and image display

Listing 8.2 shows the basic structure of a surface sampling program.

Listing 8.2

```
using namespace nv::index::xac;

class Surface_sample_program
{
    NV_IDX_SURFACE_SAMPLE_PROGRAM

public:
    NV_IDX_DEVICE_INLINE_MEMBER
    void initialize()
    {
        ...  Initialize the program
    }

    NV_IDX_DEVICE_INLINE_MEMBER
    int execute(
        const Sample_info_self& sample_info,   Read-only

              Sample_output&    sample_output)  Write-only
    {
        ...  Perform computations
       return NV_IDX_PROG_OK;
    }
};
```

The available information that is passed as input in the `Sample_info_self` depends on the associated scene element for which the program is executed.

See the “API documentation for structs used in sampling” section in the API documentation.

8.2 Scene property access

Listing 8.3 demonstrates how the state object provides access to the associated scene element for which a sample program has been called.

Listing 8.3

```
class Volume_sample_program
{
    NV_IDX_VOLUME_SAMPLE_PROGRAM

    const Regular_volume volume = state.self;   Retrieve the associated scene element reference

    ...  Program using volume
}
```

Listing 8.4 (page 66) demonstrates how scene elements that have been specified in the IndeX scene can be accessed in sample programs.

Listing 8.4

```
const Regular_volume volume =
    state.scene.access<Regular_volume>(0);
```

Example object retrieval from the scene

Additionally, the scene provides access to a set of scene element handling and basic transformation functions.

See the "API documention for the XAC scene definition class" section in the API documentation.

8.3 Scene element overview

Predefinied elements that are part of the XAC interface can provide information about the current IndeX state, for example, rendering behavior and color mapping.

```
nv::index::xac::Ray
nv::index::xac::Camera
nv::index::xac::Colormap
nv::index::xac::Light
nv::index::xac::Material_phong
```

XAC also provides a set of geometric scene elements that are used for the final visualization output:

```
nv::index::xac::Regular_volume
nv::index::xac::Height_field
nv::index::xac::Triangle_mesh
nv::index::xac::Plane
nv::index::xac::Cone
nv::index::xac::Cylinder
nv::index::xac::Ellipsoid
```

Compute texture objects can store additional information for surface scene elements.

```
nv::index::xac::Compute_texture_tile
```

See the "API documention for XAC scene elements" section in the API documentation.

8.4 XAC library functionality

Standard CUDA math functionality is available within the sample programs. Additional headers can be provided by the scene and included within the sample program.

The XAC interface also provides a set of convenience macros and functions for printing debugging information, transformation handling, basic shading operations and generic gradient operators.

See also the “API documentation for XAC macros and functions” section in the API documentation.

9 Source code for examples

9.1 build_scene_description.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/iindex.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include <nv/index/icone.h>
#include <nv/index/icylinder.h>
#include <nv/index/iellipsoid.h>
#include <nv/index/ifont.h>
#include <nv/index/ilabel.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene_group.h>
#include <nv/index/isphere.h>

#include <nv/index/app/icanvas_factory.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/index_connect.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Arrow_manipulator :
 public mi::neuraylib::Element<0xea6aa6d4,0x01c0,0x4791,0xb1,0x34,0x30,0x26,0x99,0xfe,0x51,0xa3,
                 nv::index::IShape_scene_group_manipulator>
{
public:
  Arrow_manipulator()
   : m_color_1(1.f, 0.f, 0.f, 1.0f),
    m_color_2(0.f, 1.f, 0.f, 1.0f),
    m_phong_1_tag(mi::neuraylib::NULL_TAG),
    m_phong_2_tag(mi::neuraylib::NULL_TAG)
  {
  }

  // Creates the arrow geometry in the given shape group.
  //
```

```
  // This method should not be called directly, it will be executed by
  // IShape_scene_group::build().
  virtual void build(
    nv::index::IShape_scene_group*    shape_group,
    const nv::index::IScene*          scene,
    mi::neuraylib::IDice_transaction* dice_transaction)
  {
    const mi::Float32 arrow_length = 100.f; // Default length, can be changed later

    // Material for the cone
   mi::base::Handle<nv::index::IPhong_gl> phong_1(scene->create_attribute<nv::index::IPhong_gl>())
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(m_color_1 * 0.3f);
    phong_1->set_diffuse(m_color_1 * 0.4f);
    phong_1->set_specular(mi::math::Color(0.8f));
    phong_1->set_shininess(100.f);
    m_phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.get());
    check_success(m_phong_1_tag.is_valid());
    shape_group->append(m_phong_1_tag, dice_transaction);

    // Cone
   mi::base::Handle<nv::index::ICone> cone(scene->create_shape<nv::index::ICone>());
    check_success(cone.is_valid_interface());
   cone->set_center(mi::math::Vector<mi::Float32, 3>(arrow_length * 0.65f, 0.f, 0.f));
    cone->set_tip(mi::math::Vector<mi::Float32, 3>(arrow_length, 0.f, 0.f));
    cone->set_radius(30);
   mi::neuraylib::Tag cone_tag = dice_transaction->store_for_reference_counting(cone.get());
    check_success(cone_tag.is_valid());
    shape_group->append(cone_tag, dice_transaction);

    // Material for the cylinder
   mi::base::Handle<nv::index::IPhong_gl> phong_2(scene->create_attribute<nv::index::IPhong_gl>())
    check_success(phong_2.is_valid_interface());
    phong_2->set_ambient(m_color_2 * 0.3f);
    phong_2->set_diffuse(m_color_2 * 0.4f);
    phong_2->set_specular(mi::math::Color(0.8f));
    phong_2->set_shininess(100.f);
    m_phong_2_tag = dice_transaction->store_for_reference_counting(phong_2.get());
    check_success(m_phong_2_tag.is_valid());
    shape_group->append(m_phong_2_tag, dice_transaction);

    // Cylinder
   mi::base::Handle<nv::index::ICylinder> cylinder(scene->create_shape<nv::index::ICylinder>());
    check_success(cylinder.is_valid_interface());
    cylinder->set_bottom(mi::math::Vector<mi::Float32, 3>(0.f, 0.f, 0.f));
   cylinder->set_top(mi::math::Vector<mi::Float32, 3>(arrow_length * 0.65f, 0.f, 0.f));
    cylinder->set_radius(20);
    cylinder->set_capped(true);
   mi::neuraylib::Tag cylinder_tag = dice_transaction->store_for_reference_counting(cylinder.get())
    check_success(cylinder_tag.is_valid());
    shape_group->append(cylinder_tag, dice_transaction);
  }

  // Sets the colors for the materials used by the cylinder and the cone that form the arrow.
  //
  // This method can be called before and after the arrow shape group has been built.
  void set_colors(
```

```
    const mi::math::Color&            color_1,
    const mi::math::Color&            color_2,
    mi::neuraylib::IDice_transaction* dice_transaction)
  {
    // Store that color information in the manipulator.
    m_color_1 = color_1;
    m_color_2 = color_2;

    // Update the materials, if they are already there.
    mi::base::Handle<nv::index::IPhong_gl> phong_1(
      dice_transaction->edit<nv::index::IPhong_gl>(m_phong_1_tag));
    if (phong_1)
    {
      phong_1->set_ambient(m_color_1 * 0.3f);
      phong_1->set_diffuse(m_color_1 * 0.4f);
    }

    mi::base::Handle<nv::index::IPhong_gl> phong_2(
      dice_transaction->edit<nv::index::IPhong_gl>(m_phong_2_tag));
    if (phong_2)
    {
      phong_2->set_ambient(m_color_2 * 0.3f);
      phong_2->set_diffuse(m_color_2 * 0.4f);
    }
  }

  // Changes the length of an existing arrow.
  //
 // This method may only be called after the arrow shape group has been built, since it relies
  // on the cone and cylinder already stored in the shape group.
  void change_length(
    mi::Float32                       length,
    nv::index::IShape_scene_group*    shape_group,
    mi::neuraylib::IDice_transaction* dice_transaction)
  {
   // This is an example of how a manipulator can be used for modifying a shape group without
    // storing any state information itself.
    for (mi::Uint32 i = 0; i < shape_group->nb_elements(); ++i)
    {
      mi::neuraylib::Tag tag = shape_group->get_scene_element(i);

      mi::base::Handle<nv::index::ICone> cone(
        dice_transaction->edit<nv::index::ICone>(tag));
      if (cone)
      {
       cone->set_center(mi::math::Vector<mi::Float32, 3>(length * 0.65f, 0.f, 0.f));
        cone->set_tip(mi::math::Vector<mi::Float32, 3>(length, 0.f, 0.f));
      }

      mi::base::Handle<nv::index::ICylinder> cylinder(
        dice_transaction->edit<nv::index::ICylinder>(tag));
      if (cylinder)
      {
       cylinder->set_top(mi::math::Vector<mi::Float32, 3>(length * 0.65f, 0.f, 0.f));
      }
    }
  }
```

```
  //
  // Standard helper methods
  //

  virtual const char* get_class_name() const { return "Arrow_manipulator"; }

  virtual mi::neuraylib::IElement* copy() const
  {
    Arrow_manipulator* other = new Arrow_manipulator();

    other->m_color_1     = this->m_color_1;
    other->m_color_2     = this->m_color_2;
    other->m_phong_1_tag = this->m_phong_1_tag;
    other->m_phong_2_tag = this->m_phong_2_tag;

    return other;
  }

  virtual void serialize(mi::neuraylib::ISerializer *serializer) const
  {
    serializer->write(&m_color_1.r, m_color_1.SIZE);
    serializer->write(&m_color_2.r, m_color_2.SIZE);
    serializer->write(&m_phong_1_tag.id, 1);
    serializer->write(&m_phong_2_tag.id, 1);
  }

  virtual void deserialize(mi::neuraylib::IDeserializer* deserializer)
  {
    deserializer->read(&m_color_1.r, m_color_1.SIZE);
    deserializer->read(&m_color_2.r, m_color_2.SIZE);
    deserializer->read(&m_phong_1_tag.id, 1);
    deserializer->read(&m_phong_2_tag.id, 1);
  }

protected:
  mi::math::Color    m_color_1;
  mi::math::Color    m_color_2;

  mi::neuraylib::Tag m_phong_1_tag;
  mi::neuraylib::Tag m_phong_2_tag;
};

class Build_scene_description:
    public nv::index::app::Index_connect
{
public:
  Build_scene_description()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Build_scene_description() ctor";
  }

  virtual ~Build_scene_description()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
```

```
    // INFO_LOG << "DEBUG: ~Build_scene_description() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  // override
  virtual bool evaluate_options(nv::index::app::String_dict& options) CPP11_OVERRIDE;
  // override for unittest support
  bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    assert(network_configuration != 0);

    if ((nv::index::app::get_bool(options.get("unittest", "false"))) ||
      (options.get("dice::network::mode", "") == "OFF"))
    {
      // (unittest == true) || (dice::network::mode == OFF)
      INFO_LOG << "Network setup: disabled networking.";
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

  // override
  virtual bool register_serializable_classes(
    mi::neuraylib::IDice_configuration* configuration_interface,
    nv::index::app::String_dict&        options) CPP11_OVERRIDE
  {
    assert(configuration_interface != 0);
    bool is_registered = false;
   is_registered = get_index_interface()->register_serializable_class<Arrow_manipulator>();
    check_success(is_registered);
    INFO_LOG << "register Arrow_manipulator";

    return is_registered;
  }

private:
  // Create point set in the scene.
  //
  // \param[in] scene_edit       IScene for scene edit
  // \param[in] dice_transaction dice transaction
  // \return true when success
  bool create_scene(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  //-------------------------------------------------------------------------
  // Set up camera to see the entire scene
  //
  // \param[in] cam                      camera object
 // \param[in] orthographic_projection Orthographic (true) or perspective (false) projection
```

```
  void setup_camera(
    nv::index::ICamera* cam,
    bool                orthographic_projection) const;

  //-------------------------------------------------------------------------
  // setup a far away camera for a extreme transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  //
  // \param[in] dice_transaction dice transaction
  void setup_extreme_transformed_camera(
    nv::index::ICamera* cam) const;

  // setup a large scene transform for the extreme
  // transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  mi::math::Matrix<mi::Float32, 4, 4> get_extreme_transformed_matrix() const;

  //-------------------------------------------------------------------------
  // render a frame
  //
  // \param[in] arc              application rendering context
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(
    const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                  m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                    m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Application options
  std::string                                                            m_outfname;
  bool                                                                m_is_supersampling;
  bool                                                                 m_is_orthographic;
  bool                                                               m_is_large_translate;
  std::string                                                        m_verify_image_fname;
  std::string                                                                 m_export;
};

mi::Sint32 Build_scene_description::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    nv::index::IIndex* iindex = get_index_interface();
    // INFO_LOG << "iindex_ref: retain: "  << iindex->retain();
    // INFO_LOG << "iindex_ref: release: " << iindex->release();
    check_success(iindex != 0);
   nv::index::app::IApplication_layer* app_layer = get_application_layer_interface();
    check_success(app_layer != 0);
```

```
m_cluster_configuration = iindex->get_api_component<nv::index::ICluster_configuration>();
 check_success(m_cluster_configuration.is_valid_interface());

 // create image canvas in application_layer
 m_image_file_canvas = create_image_file_canvas(app_layer);
 check_success(m_image_file_canvas.is_valid_interface());

 // Verifying that local host has joined
 // This may fail when there is a license problem.
 check_success(is_local_host_joined(m_cluster_configuration.get()));

 // Setup the scene
 {
   mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
     m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
   check_success(dice_transaction.is_valid_interface());
   {
     // Setup session information
     m_session_tag = m_index_session->create_session(dice_transaction.get());
     check_success(m_session_tag.is_valid());
     mi::base::Handle<nv::index::ISession const> session(
       dice_transaction->access<nv::index::ISession const>(m_session_tag));
     check_success(session.is_valid_interface());

     mi::base::Handle<nv::index::IScene> scene_edit(
       dice_transaction->edit<nv::index::IScene>(
         session->get_scene()));
     check_success(scene_edit.is_valid_interface());

     // Enable supersampling
     if (m_is_supersampling)
     {
       INFO_LOG << "Enable supersampling.";

       mi::base::Handle<nv::index::IConfig_settings> config_settings(
     dice_transaction->edit<nv::index::IConfig_settings>(session->get_config()));
       check_success(config_settings.is_valid_interface());

       config_settings->set_rendering_samples(8);
     }

     //-------------------------------------------------------------------------
     // Scene setup: hierarchical scene description root node,
     // scene parameters, camera.
     //-------------------------------------------------------------------------
     // create a hierarchical scene description
     check_success(create_scene(scene_edit.get(), dice_transaction.get()));

     // Create and edit a camera. Data distribution is based on
     // the camera. (Because only visible massive data are
     // considered)
     mi::base::Handle<nv::index::ICamera> cam;
     if (m_is_orthographic)
     {
       mi::base::Handle<nv::index::ICamera> tmp_cam(
         scene_edit->create_camera<nv::index::IOrthographic_camera>());
```

```
      cam.swap(tmp_cam);
    }
    else
    {
      mi::base::Handle<nv::index::ICamera> tmp_cam(
        scene_edit->create_camera<nv::index::IPerspective_camera>());
      cam.swap(tmp_cam);
    }

    mi::math::Vector<mi::Uint32, 2> buffer_resolution(1024, 1024);
    if (m_is_large_translate)
    {
      INFO_LOG << "large translate test mode.";
      setup_extreme_transformed_camera(cam.get());
      buffer_resolution = mi::math::Vector<mi::Uint32, 2>(1602, 934);
    }
    else
    {
      setup_camera(cam.get(), m_is_orthographic);
    }

    const mi::neuraylib::Tag cam_tag = dice_transaction->store(cam.get());
    check_success(cam_tag.is_valid());

    // Setup the canvas resolution
    m_image_file_canvas->set_resolution(buffer_resolution);

    // Set up the scene
    const mi::math::Bbox<mi::Float32, 3> xyz_roi(
      0.0f,   0.0f,   0.0f,
      500.0f, 500.0f, 500.0f);

    // set the region of interest
    check_success(xyz_roi.is_volume());
    scene_edit->set_clipped_bounding_box(xyz_roi);

    // Set the scene global transformation matrix.
    // only change the coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
      0.0f,  0.0f,  0.0f,  1.0f
      );

    if (m_is_large_translate)
    {
      transform_mat = get_extreme_transformed_matrix();
    }

    scene_edit->set_transform_matrix(transform_mat);

    // Set the current camera to the scene.
    check_success(cam_tag.is_valid());
    scene_edit->set_camera(cam_tag);
  }
  // Finish the scene setup transaction
```

```
      dice_transaction->commit();
    }

    // Render the scene
    {
      // Render a frame and save the rendered image to a file.
      const mi::Sint32  frame_idx = 0;
      const std::string fname = get_output_file_name(m_outfname, frame_idx);
     mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));

    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      if (err_set->any_errors())
      {
        std::ostringstream os;
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

      ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        exit_code = 1;
      }

      // verify the generated frame
    if (!verify_canvas_result(app_layer, m_image_file_canvas.get(), m_verify_image_fname, get_optic
      {
        exit_code = 1;
      }

      // Session export
      if (!m_export.empty())
      {
        std::string export_str;
        mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
          m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
        check_success(dice_transaction.is_valid_interface());
        {
          export_session(m_session_tag, dice_transaction.get(), export_str);
        }
        dice_transaction->commit();

        INFO_LOG << "Writing export to file '" << m_export << "'";
        std::ofstream f(m_export.c_str());
        f << export_str;
        if (!f)
        {
          ERROR_LOG << "Failed to export to file: " << m_export;
        }
        f.close();
      }
    }
  }
```

```
  return exit_code;
}

bool Build_scene_description::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  if (nv::index::app::get_bool(sdict.get("unittest", "false")))
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
    sdict.insert("verify_image_fname_sequence", ""); // disable this in unit test
  }

  // Set own options
  m_outfname           = sdict.get("outfname", "");
  m_is_supersampling   = nv::index::app::get_bool(sdict.get("supersampling", "0"));
  m_is_orthographic    = nv::index::app::get_bool(sdict.get("orthographic",  "0"));
 m_is_large_translate = nv::index::app::get_bool(sdict.get("is_large_translate", "0"));
  m_verify_image_fname = sdict.get("verify_image_fname", "");
  m_export             = sdict.get("export", "");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout(std::string("outfname = [") + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = " + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-supersampling bool]\n"
      << "            when true then supersampling is enabled. "
      << "(default: " << sdict.get("supersampling") << ")\n"

      << "        [-orthographic bool]\n"
    << "         when true use orthographic projection, else perspective projection. "
      << "(default: " << sdict.get("orthographic") << ")\n"

      << "        [-is_large_translate bool]\n"
      << "            on/off large translation mode. "
      << "(default: " << sdict.get("is_large_translate") << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << sdict.get("outfname") << "])\n"
```

```
      << "        [-verify_image_fname [image_fname]]\n"
      << "           when image_fname exist, verify the rendering image. (default: ["
      << sdict.get("verify_image_fname") << "])\n"

      << "        [-export string]\n"
      << "            output file for session export, use 'stdout' to print\n"
      << "            to standard output.\n"
      << "            (default: [" << sdict.get("outfname") << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << sdict.get("unittest") << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

bool Build_scene_description::create_scene(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  // Add a light and a material
  {
    // Set the default light source
    mi::base::Handle<nv::index::IDirectional_light> light_global(
      scene_edit->create_attribute<nv::index::IDirectional_light>());
    check_success(light_global.is_valid_interface());
    light_global->set_direction(mi::math::Vector<mi::Float32, 3>(0.5f, 0.f, 1.f));
   const mi::neuraylib::Tag light_global_tag = dice_transaction->store_for_reference_counting(light
    check_success(light_global_tag.is_valid());
    scene_edit->append(light_global_tag, dice_transaction);

    // Set the default material
   mi::base::Handle<nv::index::IPhong_gl> phong_global(scene_edit->create_attribute<nv::index::IPh
    check_success(phong_global.is_valid_interface());
    phong_global->set_ambient(mi::math::Color(0.1f, 0.15f, 0.1f, 1.0f));
    phong_global->set_diffuse(mi::math::Color(0.95f, 0.3f, 0.3f, 1.0f));
    phong_global->set_specular(mi::math::Color(0.4f, 0.4f, 0.5f, 1.0f));
    phong_global->set_shininess(100);
   const mi::neuraylib::Tag phong_global_tag = dice_transaction->store_for_reference_counting(phong
    check_success(phong_global_tag.is_valid());
    scene_edit->append(phong_global_tag, dice_transaction);
  }

  // Create and store a static scene group
  mi::base::Handle<nv::index::IStatic_scene_group> static_scene(
    scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
  check_success(static_scene.is_valid_interface());
 const mi::neuraylib::Tag static_scene_tag = dice_transaction->store_for_reference_counting(static_
  check_success(static_scene_tag.is_valid());

  {
```

```
 // Set up the transformation matrix, define a translation and create and store the transform group
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  transform_mat.translate(mi::math::Vector<mi::Float32, 3>(40.f, 0.f, 0.f));
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());
  group_node->set_transform(transform_mat);

  // Create some cones
 mi::base::Handle<nv::index::ICone> cone_1(scene_edit->create_shape<nv::index::ICone>());
  check_success(cone_1.is_valid_interface());
  cone_1->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 10.f, 100.f));
  cone_1->set_tip(mi::math::Vector<mi::Float32, 3>(0.f, 110.f, 100.f));
  cone_1->set_radius(40);
 const mi::neuraylib::Tag cone_1_tag = dice_transaction->store_for_reference_counting(cone_1.get(
  check_success(cone_1_tag.is_valid());
  group_node->append(cone_1_tag, dice_transaction);

 mi::base::Handle<nv::index::ICone> cone_2(scene_edit->create_shape<nv::index::ICone>());
  check_success(cone_2.is_valid_interface());
  cone_2->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 130.f, 100.f));
  cone_2->set_tip(mi::math::Vector<mi::Float32, 3>(0.f, 280.f, 100.f));
  cone_2->set_radius(40);
 const mi::neuraylib::Tag cone_2_tag = dice_transaction->store_for_reference_counting(cone_2.get(
  check_success(cone_2_tag.is_valid());
  group_node->append(cone_2_tag, dice_transaction);

 mi::base::Handle<nv::index::ICone> cone_3(scene_edit->create_shape<nv::index::ICone>());
  check_success(cone_2_tag.is_valid());
  cone_3->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 310.f, 100.f));
  cone_3->set_tip(mi::math::Vector<mi::Float32, 3>(0.f, 500.f, 100.f));
  cone_3->set_radius(40);
 const mi::neuraylib::Tag cone_3_tag = dice_transaction->store_for_reference_counting(cone_3.get(
  check_success(cone_3_tag.is_valid());
  group_node->append(cone_3_tag, dice_transaction);

 const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_n
  check_success(group_node_tag.is_valid());
  INFO_LOG << "Appending transformed scene group node to the root node.";
  scene_edit->append(group_node_tag, dice_transaction);
}

{
 // Set up the transformation matrix, define a translation and create and store the transform group
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  transform_mat.translate(mi::math::Vector<mi::Float32, 3>(160.f, 0.f, 0.f));
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());
  group_node->set_transform(transform_mat);

  // Create some spheres with materials
 mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
  check_success(phong_1.is_valid_interface());
  phong_1->set_ambient(mi::math::Color(0.1f, 0.15f, 0.1f, 1.0f));
  phong_1->set_diffuse(mi::math::Color(0.3f, 0.85f, 0.3f, 1.0f));
  phong_1->set_specular(mi::math::Color(0.4f, 0.4f, 0.5f, 1.0f));
```

```
  phong_1->set_shininess(10);
 const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
  check_success(phong_1_tag.is_valid());
  group_node->append(phong_1_tag, dice_transaction);

 mi::base::Handle<nv::index::ISphere> sphere_1(scene_edit->create_shape<nv::index::ISphere>());
  check_success(sphere_1.is_valid_interface());
  sphere_1->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 50.f, 100.f));
  sphere_1->set_radius(30);
 const mi::neuraylib::Tag sphere_1_tag = dice_transaction->store_for_reference_counting(sphere_1.
  check_success(sphere_1_tag.is_valid());
  group_node->append(sphere_1_tag, dice_transaction);

 mi::base::Handle<nv::index::IPhong_gl> phong_2(scene_edit->create_attribute<nv::index::IPhong_g
  check_success(phong_2.is_valid_interface());
  phong_2->set_ambient(mi::math::Color(0.1f, 0.15f, 0.1f, 1.0f));
  phong_2->set_diffuse(mi::math::Color(0.4f, 0.75f, 0.4f, 1.0f));
  phong_2->set_specular(mi::math::Color(0.4f, 0.4f, 0.5f, 1.0f));
  phong_2->set_shininess(50);
 const mi::neuraylib::Tag phong_2_tag = dice_transaction->store_for_reference_counting(phong_2.ge
  check_success(phong_2_tag.is_valid());
  group_node->append(phong_2_tag, dice_transaction);

 mi::base::Handle<nv::index::ISphere> sphere_2(scene_edit->create_shape<nv::index::ISphere>());
  check_success(sphere_2.is_valid_interface());
  sphere_2->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 200.f, 100.f));
  sphere_2->set_radius(50);
 const mi::neuraylib::Tag sphere_2_tag = dice_transaction->store_for_reference_counting(sphere_2.
  check_success(sphere_2_tag.is_valid());
  group_node->append(sphere_2_tag, dice_transaction);

 mi::base::Handle<nv::index::IDirectional_light> light(scene_edit->create_attribute<nv::index::I
  light->set_direction(mi::math::Vector<mi::Float32, 3>(-0.5f, 0.f, 1.f));
 const mi::neuraylib::Tag light_tag = dice_transaction->store_for_reference_counting(light.get())
  check_success(light_tag.is_valid());
  group_node->append(light_tag, dice_transaction);

 mi::base::Handle<nv::index::IPhong_gl> phong_3(scene_edit->create_attribute<nv::index::IPhong_g
  check_success(phong_3.is_valid_interface());
  phong_3->set_ambient(mi::math::Color(0.1f, 0.15f, 0.1f, 1.0f));
  phong_3->set_diffuse(mi::math::Color(0.5f, 0.55f, 0.5f, 1.0f));
  phong_3->set_specular(mi::math::Color(0.4f, 0.4f, 0.5f, 1.0f));
  phong_3->set_shininess(80);
 const mi::neuraylib::Tag phong_3_tag = dice_transaction->store_for_reference_counting(phong_3.ge
  check_success(phong_3_tag.is_valid());
  group_node->append(phong_3_tag, dice_transaction);

 mi::base::Handle<nv::index::ISphere> sphere_3(scene_edit->create_shape<nv::index::ISphere>());
  check_success(sphere_3.is_valid_interface());
  sphere_3->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 400.f, 100.f));
  sphere_3->set_radius(70);
 const mi::neuraylib::Tag sphere_3_tag = dice_transaction->store_for_reference_counting(sphere_3.
  check_success(sphere_3_tag.is_valid());
  group_node->append(sphere_3_tag, dice_transaction);

 const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_n
  check_success(group_node_tag.is_valid());
```

```
   INFO_LOG << "Appending transformed scene group node to the static scene group...";
    scene_edit->append(group_node_tag, dice_transaction);
  }

  {
   // Set up the transformation matrix, define a translation and create and store the transform group
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
    transform_mat.translate(mi::math::Vector<mi::Float32, 3>(320.f, 0.f, 0.f));
    mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
      scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
    check_success(group_node.is_valid_interface());
    group_node->set_transform(transform_mat);

    // Create some spheres mit materials
  mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(mi::math::Color(0.1f, 0.15f, 0.1f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.3f, 0.2f, 0.85f, 1.0f));
    phong_1->set_specular(mi::math::Color(0.4f, 0.4f, 0.5f, 1.0f));
    phong_1->set_shininess(10);
  const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
    check_success(phong_1_tag.is_valid());
    group_node->append(phong_1_tag, dice_transaction);

  mi::base::Handle<nv::index::IEllipsoid> ellipsoid_1(scene_edit->create_shape<nv::index::IEllips
    check_success(ellipsoid_1.is_valid_interface());
    ellipsoid_1->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 50.f, 100.f));
    ellipsoid_1->set_semi_axes(
      mi::math::Vector<mi::Float32, 3>(30.f, 0.f, 0.f),
      mi::math::Vector<mi::Float32, 3>(0.f, 50.f, 0.f),
      30.f);
  const mi::neuraylib::Tag ellipsoid_1_tag = dice_transaction->store_for_reference_counting(ellips
    check_success(ellipsoid_1_tag.is_valid());
    group_node->append(ellipsoid_1_tag, dice_transaction);

  mi::base::Handle<nv::index::IPhong_gl> phong_2(scene_edit->create_attribute<nv::index::IPhong_g
    check_success(phong_2.is_valid_interface());
    phong_2->set_ambient(mi::math::Color(0.1f, 0.15f, 0.1f, 1.0f));
    phong_2->set_diffuse(mi::math::Color(0.4f, 0.3f, 0.75f, 1.0f));
    phong_2->set_specular(mi::math::Color(0.4f, 0.4f, 0.5f, 1.0f));
    phong_2->set_shininess(50);
  const mi::neuraylib::Tag phong_2_tag = dice_transaction->store_for_reference_counting(phong_2.ge
    check_success(phong_2_tag.is_valid());
    group_node->append(phong_2_tag, dice_transaction);

  mi::base::Handle<nv::index::IEllipsoid> ellipsoid_2(scene_edit->create_shape<nv::index::IEllips
    check_success(ellipsoid_2.is_valid_interface());
    ellipsoid_2->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 200.f, 100.f));
    ellipsoid_2->set_semi_axes(
      mi::math::Vector<mi::Float32, 3>(70.f, 0.f, 0.f),
      mi::math::Vector<mi::Float32, 3>( 0.f, 50.f, 0.f),
      80.f);
  const mi::neuraylib::Tag ellipsoid_2_tag = dice_transaction->store_for_reference_counting(ellips
    check_success(ellipsoid_2_tag.is_valid());
    group_node->append(ellipsoid_2_tag, dice_transaction);

  mi::base::Handle<nv::index::IPhong_gl> phong_3(scene_edit->create_attribute<nv::index::IPhong_g
```

```
  check_success(phong_3.is_valid_interface());
  phong_3->set_ambient(mi::math::Color(0.1f, 0.15f, 0.1f, 1.0f));
  phong_3->set_diffuse(mi::math::Color(0.5f, 0.4f, 0.65f, 1.0f));
  phong_3->set_specular(mi::math::Color(0.4f, 0.4f, 0.5f, 1.0f));
  phong_3->set_shininess(80);
 const mi::neuraylib::Tag phong_3_tag = dice_transaction->store_for_reference_counting(phong_3.ge
  check_success(phong_3_tag.is_valid());
  group_node->append(phong_3_tag, dice_transaction);

 mi::base::Handle<nv::index::IEllipsoid> ellipsoid_3(scene_edit->create_shape<nv::index::IEllips
  check_success(ellipsoid_3.is_valid_interface());
  ellipsoid_3->set_center(mi::math::Vector<mi::Float32, 3>(0.f, 400.f, 100.f));
  ellipsoid_3->set_semi_axes(
    mi::math::Vector<mi::Float32, 3>(60.f, 0.f, 0.f),
    mi::math::Vector<mi::Float32, 3>(0.f, 50.f, 0.f),
    70.f);
 const mi::neuraylib::Tag ellipsoid_3_tag = dice_transaction->store_for_reference_counting(ellips
  check_success(ellipsoid_3_tag.is_valid());
  group_node->append(ellipsoid_3_tag, dice_transaction);

 const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_n
  check_success(group_node_tag.is_valid());
 INFO_LOG << "Appending transformed scene group node to the static scene group...";
  scene_edit->append(group_node_tag, dice_transaction);
}

{
 // Set up the transformation matrix, define a translation and create and store the transform group
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  transform_mat.translate(mi::math::Vector<mi::Float32, 3>(440.f, 0.f, 0.f));
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());
  group_node->set_transform(transform_mat);

 mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
  check_success(phong_1.is_valid_interface());
  phong_1->set_ambient(mi::math::Color(0.15f, 0.15f, 0.1f, 1.0f));
  phong_1->set_diffuse(mi::math::Color(0.85f, 0.85f, 0.2f, 1.0f));
  phong_1->set_specular(mi::math::Color(0.2f, 0.2f, 0.7f, 1.0f));
  phong_1->set_shininess(80);
 const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
  check_success(phong_1_tag.is_valid());
  group_node->append(phong_1_tag, dice_transaction);

  // Create some cylinders
 mi::base::Handle<nv::index::ICylinder> cylinder_1(scene_edit->create_shape<nv::index::ICylinder
  check_success(cylinder_1.is_valid_interface());
  cylinder_1->set_bottom(mi::math::Vector<mi::Float32, 3>(0.f, 20.f, 125.f));
  cylinder_1->set_top(mi::math::Vector<mi::Float32, 3>(0.f, 100.f, 75.f));
  cylinder_1->set_radius(30);
  cylinder_1->set_capped(false);
 const mi::neuraylib::Tag cylinder_1_tag = dice_transaction->store_for_reference_counting(cylinde
  check_success(cylinder_1_tag.is_valid());
  group_node->append(cylinder_1_tag, dice_transaction);

 mi::base::Handle<nv::index::ICylinder> cylinder_2(scene_edit->create_shape<nv::index::ICylinder
```

```
    check_success(cylinder_2.is_valid_interface());
    cylinder_2->set_bottom(mi::math::Vector<mi::Float32, 3>(50.f, 120.f, 125.f));
    cylinder_2->set_top(mi::math::Vector<mi::Float32, 3>(-50.f, 270.f, 75.f));
    cylinder_2->set_radius(20);
   const mi::neuraylib::Tag cylinder_2_tag = dice_transaction->store_for_reference_counting(cylinde
    check_success(cylinder_2_tag.is_valid());
    group_node->append(cylinder_2_tag, dice_transaction);

   mi::base::Handle<nv::index::ICylinder> cylinder_3(scene_edit->create_shape<nv::index::ICylinder
    check_success(cylinder_3.is_valid_interface());
    cylinder_3->set_bottom(mi::math::Vector<mi::Float32, 3>(0.f, 320.f, 100.f));
    cylinder_3->set_top(mi::math::Vector<mi::Float32, 3>(0.f, 480.f, 100.f));
    cylinder_3->set_radius(30);
   const mi::neuraylib::Tag cylinder_3_tag = dice_transaction->store_for_reference_counting(cylinde
    check_success(cylinder_3_tag.is_valid());
    group_node->append(cylinder_3_tag, dice_transaction);

   const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_n
    check_success(group_node_tag.is_valid());
   INFO_LOG << "Appending transformed scene group node to the static scene group...";
    scene_edit->append(group_node_tag, dice_transaction);
  }

  //
  // Add an arrow shape, which is implemented using a shape group
  //
  mi::neuraylib::Tag arrow_tag;
  {
    // Create the arrow manipulator
    mi::base::Handle<Arrow_manipulator> arrow_manipulator(new Arrow_manipulator());

    // Set default colors (will be overwritten later), just to show it is possible
    arrow_manipulator->set_colors(
      mi::math::Color(0.f, 0.f, 0.f),
      mi::math::Color(1.f, 1.f, 1.f),
      dice_transaction);

    // Store the manipulator
   mi::neuraylib::Tag arrow_manipulator_tag = dice_transaction->store_for_reference_counting(arrow_
    check_success(arrow_manipulator_tag.is_valid());

    // Create a shape group for the arrow
    mi::base::Handle<nv::index::IShape_scene_group> arrow(
      scene_edit->create_scene_group<nv::index::IShape_scene_group>());
    check_success(arrow.is_valid_interface());

    // Add a transformation
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
    transform_mat.rotate(0.0f, static_cast<mi::Float32>(MI_PI / 7.0), 0.0f);
    transform_mat.translate(mi::math::Vector<mi::Float32, 3>(120.f, 290.f, 150.f));
    arrow->set_transform(transform_mat);

   // Build the arrow geometry, associating the manipulator with the arrow shape group
    arrow->build(arrow_manipulator_tag, scene_edit, dice_transaction);

    // Store the arrow shape group and append it to the scene
    arrow_tag = dice_transaction->store_for_reference_counting(arrow.get());
```

```
    check_success(arrow_tag.is_valid());
    scene_edit->append(arrow_tag, dice_transaction);
  }

  // Now apply some modifications to the newly created arrow shape group
  {
    mi::base::Handle<nv::index::IShape_scene_group> arrow(
      dice_transaction->edit<nv::index::IShape_scene_group>(arrow_tag));

    // We also need the manipulator
    mi::base::Handle<Arrow_manipulator> arrow_manipulator(
      dice_transaction->edit<Arrow_manipulator>(arrow->get_manipulator()));

    // Call the manipulator to change the contents of the arrow shape group
    arrow_manipulator->set_colors(
      mi::math::Color(1.f, 0.f, 0.f),
      mi::math::Color(0.f, 1.f, 0.f),
      dice_transaction);

    arrow_manipulator->change_length(
      250.f,
      arrow.get(),
      dice_transaction);
  }

  // A static scene group can only be attached to the scene root (scene)
  // or another static scene group.
  INFO_LOG << "Appending scene group to the scene root...";
  scene_edit->append(static_scene_tag, dice_transaction);

  INFO_LOG << "Hierarchical scene description complete.";
  return true;
}

void Build_scene_description::setup_camera(
  nv::index::ICamera* cam,
  bool                orthographic_projection) const
{
  check_success(cam != 0);

  if (orthographic_projection)
  {
    mi::base::Handle<nv::index::IOrthographic_camera> ocam(
      cam->get_interface<nv::index::IOrthographic_camera>());
    check_success(ocam.is_valid_interface());

    // Set the camera parameters to see the whole scene
    const mi::math::Vector<mi::Float32, 3> from(254.f, 254.f,  550.f);
    const mi::math::Vector<mi::Float32, 3> to  (255.f, 255.f, -255.f);
    const mi::math::Vector<mi::Float32, 3> up  (0.f,     1.f,    0.f);
    mi::math::Vector<mi::Float32, 3> viewdir = to - from;
    viewdir.normalize();

    ocam->set(from, viewdir, up);
    ocam->set_aperture(0.033f);
    ocam->set_aspect(1.f);
    ocam->set_clip_min(10.f);
```

```
    ocam->set_clip_max(5000.f);

    // Set the aperature to get a similar view as with perspective projection.
    // The focal length has no effect on the orthographic projection.
    ocam->set_aperture(700.f);
  }
  else
  {
    mi::base::Handle<nv::index::IPerspective_camera> pcam(
      cam->get_interface<nv::index::IPerspective_camera>());
    check_success(pcam.is_valid_interface());

    // Set the camera parameters to see the whole scene
    const mi::math::Vector<mi::Float32, 3> from(254.f, 254.f,  550.f);
    const mi::math::Vector<mi::Float32, 3> to  (255.f, 255.f, -255.f);
    const mi::math::Vector<mi::Float32, 3> up  (0.f,     1.f,    0.f);
    mi::math::Vector<mi::Float32, 3> viewdir = to - from;
    viewdir.normalize();

    pcam->set(from, viewdir, up);
    pcam->set_aperture(0.033f);
    pcam->set_aspect(1.f);
    pcam->set_focal(0.03f);
    pcam->set_clip_min(10.f);
    pcam->set_clip_max(5000.f);
  }
}

void Build_scene_description::setup_extreme_transformed_camera(
  nv::index::ICamera* cam) const
{
  check_success(cam != 0);

  mi::base::Handle<nv::index::IPerspective_camera> pcam(
    cam->get_interface<nv::index::IPerspective_camera>());
  check_success(pcam.is_valid_interface());

  // *** Camera:
  // index::camera::eye_point = 1808409.125 9981818 10948.5126953125
  // index::camera::view_direction = (1808409.125 9981818 -1500) - eye_point
  // index::camera::up_direction = 0 1 0
  // index::camera::aspect = 1.7152034261242
  // index::camera::aperture = 0.033
  // index::camera::focal = 0.03
  // index::camera::clip_min = 124.485130310059
  // index::camera::clip_max = 17585.455078125
  // index::canvas_resolution = 1602 934

  // Adjusted the camera position for p = 23+1.
 mi::math::Vector< mi::Float32, 3 > const from( 1805060.125f + 256.0f,  997852.0f + 256.0f, 1024.0f);
 mi::math::Vector< mi::Float32, 3 > const to  ( 1805060.125f + 256.0f,  997852.0f + 256.0f, -256.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  pcam->set(from, viewdir, up);
  pcam->set_aperture(0.033f);
```

```
  pcam->set_aspect(1.71520f); // not 1.7152034261242f, see IEEE754
  pcam->set_focal(0.03f);
  pcam->set_clip_min(124.485f); // not 124.485130310059f, see IEEE754
  pcam->set_clip_max(17585.6f); // not 17585.55078125f,   see IEEE754

  INFO_LOG << "Set camera extreme mode";
}

mi::math::Matrix<mi::Float32, 4, 4> Build_scene_description::get_extreme_transformed_matrix() const
{
  // transform =
  //  [ 20.6100006103516   0                 0       1805060
  //    0                 20.6100006103516  0       9978520
  //    0                  0               -4       0
  //    0                  0                0       1       ]

  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f,        0.0f,        0.0f,  0.0f,
    0.0f,        1.f,         0.0f,  0.0f,
    0.0f,        0.0f,       -1.0f,  0.0f,
    1805060.0f,  997852.0f,   0.0f,  1.0f
    );
  return transform_mat;
}

nv::index::IFrame_results* Build_scene_description::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
```

```
  // Build_scene_description settings
  sdict.insert("dice::verbose", "3");                                  // log level
 sdict.insert("outfname",    "frame_build_scene_description"); // output file base name
 sdict.insert("export",     "");                       // output file for session export
  sdict.insert("verify_image_fname", "");                              // for unit test
  sdict.insert("unittest",      "0");                                  // default mode
 sdict.insert("supersampling", "0");                          // disable supersampling
 sdict.insert("orthographic",  "0");                        // orthographic projection (default: perspe
 sdict.insert("is_large_translate", "0");                    // large translation mode (default 0)
 sdict.insert("is_dump_comparison_image_when_failed", "1");     // default: dump images when failed
 sdict.insert("is_call_from_test", "0");                     // default: not call from make check.

  // DiCE settings
  sdict.insert("dice::network::mode", "OFF");

  // IndeX settings
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer settings
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // get the command line (argc, argv) to override in sdict.
  Build_scene_description build_scene_description;
  build_scene_description.initialize(argc, argv, sdict);
  check_success(build_scene_description.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = build_scene_description.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.2 cluster_performance_mainhost.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#define USE_NVINDEX_ACCESS
#include "utility/example_shared.h"
#include "utility/example_performance_logger.h"

#include <mi/dice.h>

#include <nv/index/iindex.h>
#include <nv/index/isession.h>
#include <nv/index/iscene.h>
#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>

#include <iostream>
#include <sstream>

#include "utility/app_rendering_context.h"

#include <nv/index/app/forwarding_logger.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>

class App_data
{
public:
  // default constructor
  App_data()
    :
    m_colormap_tag(mi::neuraylib::NULL_TAG),
    m_camera_tag(mi::neuraylib::NULL_TAG),
    m_performance_logger()
  {
    // empty
  }

  // destructor
  ~App_data()
  {
    // empty
  }

  // peek performance logger
  //
  // Note: this method peeking the inside of this instance. use
  // with care.
  Example_performance_logger * peek_performance_logger()
  {
    return &m_performance_logger;
  }
```

```
public:
  // colormap tag
  mi::neuraylib::Tag m_colormap_tag;
  // camera tag
  mi::neuraylib::Tag m_camera_tag;
  // example performance logger
  Example_performance_logger m_performance_logger;

private:
  // copy constructor. prohibit until proved useful.
  App_data(App_data const &);
  // operator=. prohibit until proved useful.
  App_data const & operator=(App_data const &);
};

static void config_network_parameters(Nvindex_access & nvindex_accessor,
                   std::map< std::string, std::string > & opt_map)
{
  mi::base::Handle<mi::neuraylib::IDebug_configuration> debug_configuration(
 nvindex_accessor.get_interface()->get_api_component<mi::neuraylib::IDebug_configuration>());
  check_success(debug_configuration.is_valid_interface());

  if(opt_map["dice::network::additional_unicast_sockets"] != "0")
  {
    std::stringstream sstr;
    sstr << "dice::network::additional_unicast_sockets="
      << opt_map["dice::network::additional_unicast_sockets"];
    const std::string additional_unicast_sockets = sstr.str();
    debug_configuration->set_option(additional_unicast_sockets.c_str());

   INFO_LOG << "Additional unicast sockets: " << opt_map["dice::network::additional_unicast_sockets"
  }

  if(opt_map["send_elements_only_to_owners"] == "1")
  {
  debug_configuration->set_option("send_elements_only_to_owners=1");
    INFO_LOG << "Send elements only to owners: enabled";
  }

  if(!opt_map["disk_cache_path"].empty())
  {
    std::stringstream sstr;
    sstr << "disk_cache_path=\"" << opt_map["disk_cache_path"] << '"';
  debug_configuration->set_option(sstr.str().c_str());
    INFO_LOG << "Disk cache path: " << opt_map["disk_cache_path"];
  }
}

static void start_nvindex_accessor(Nvindex_access & nvindex_accessor,
                  std::map< std::string, std::string > & opt_map)
{
 info_cout(std::string("NVIDIA IndeX version: ") + nvindex_accessor.get_interface()->get_version(),

  initialize_log_module(nvindex_accessor, opt_map);

  // Configure networking
  mi::base::Handle<mi::neuraylib::INetwork_configuration> inetwork_configuration(
```

```
   nvindex_accessor.get_interface()->get_api_component<mi::neuraylib::INetwork_configuration>());
  check_success(inetwork_configuration.is_valid_interface());

  check_success(opt_map.find("unittest") != opt_map.end());
  bool const is_unittest = nv::index::app::get_bool(opt_map["unittest"]);
  if(is_unittest){
   info_cout("NETWORK: disabled networking mode due to the unit test mode.", opt_map);
   inetwork_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
  }
  else
  {
    info_cout("NETWORK: Enabling UDP networking mode.", opt_map);
   inetwork_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_UDP);

    check_success(opt_map.find("dice::network::multicast_address") != opt_map.end());
    check_success(!(opt_map["dice::network::multicast_address"].empty()));
    const std::string multicast_addr(opt_map["dice::network::multicast_address"]);
   info_cout(std::string("NETWORK: UDP network is configured with multicast address [") +
         multicast_addr + std::string("]."), opt_map);
    inetwork_configuration->set_multicast_address(multicast_addr.c_str());

    info_cout("NETWORK: no cluster interface address is set. "
      "If you have multiple network card on a machine, you need to set this.", opt_map);
    // const std::string cluster_if_adder("172.16.0.0/22:10001");
   // check_success(inetwork_configuration->set_cluster_interface(cluster_if_adder.c_str() == 0);
  }

  // general configuration
  mi::base::Handle<mi::neuraylib::IGeneral_configuration> igeneral_configuration(
   nvindex_accessor.get_interface()->get_api_component<mi::neuraylib::IGeneral_configuration>());
  check_success(igeneral_configuration.is_valid_interface());
 igeneral_configuration->set_host_property("sub_cluster_id", "0"); // set default cluster id 0

  // Register serializable classes.
  {
    // No serializable class registration
    // bool is_registered = false;
  }

  // Define service mode to rendering and compositing (main host)
  mi::base::Handle<nv::index::ICluster_configuration> rendering_properties_query(
    nvindex_accessor.get_interface()->
    get_api_component<nv::index::ICluster_configuration>());
  rendering_properties_query->set_service_mode("rendering_and_compositing");
  info_cout("NETWORK: set rendering_and_compositing.", opt_map);

  config_network_parameters(nvindex_accessor, opt_map);

  mi::base::Handle<mi::neuraylib::IGeneral_configuration> general_configuration(
   nvindex_accessor.get_interface()->get_api_component<mi::neuraylib::IGeneral_configuration>());
  check_success(general_configuration.is_valid_interface());
  general_configuration->set_admin_http_address("0.0.0.0:12345");

  // Start IndeX service
  nvindex_accessor.start_service();
}
```

```
static void get_top_view_param(mi::math::Bbox< mi::Float32, 3 > const & vbbox,
                const mi::Float32 aspect,
                const mi::Float32 fovy_2,
                mi::math::Vector< mi::Float32, 3 > & from,
                mi::math::Vector< mi::Float32, 3 > & to,
                mi::math::Vector< mi::Float32, 3 > & up,
                mi::Float32                        & clip_min,
                mi::Float32                        & clip_max)
{
  check_success((0.0 < fovy_2) && (fovy_2 < M_PI_2));
 const mi::Float32 dist = static_cast<mi::Float32>((0.25 * mi::math::euclidean_distance(vbbox.max, v
                          (sqrt(2.0) * tan(fovy_2)));
 from = -(dist * mi::math::Vector< mi::Float32, 3 >(0.0f, 0.0f, -1.0f)) + vbbox.center();
  to   = vbbox.center();
  up   = mi::math::Vector< mi::Float32, 3 >(0.0f, 1.0f, 0.0f);
  clip_min = 0.1f  * dist;
  clip_max = 10.0f * dist;
}


static mi::neuraylib::Tag create_synthetic_volume(
  const mi::neuraylib::Tag &               colormap_tag,
  nv::index::IScene*                       scene,
  const std::string &                      synthetic_volume_type_str,
  const std::string &                      synthetic_volume_parameter_str,
  const mi::math::Vector<mi::Uint32, 3> &  volume_size,
  const std::string&                       volume_name,
  const mi::math::Vector<mi::Float32, 3> & scale,
  const mi::Float32                        rotate_k,
  const mi::math::Vector<mi::Float32, 3> & translate,
  bool                                     is_rendering_enabled,
  mi::neuraylib::IDice_transaction*        dice_transaction)
{
  const std::string fn = "create_synthetic_volume: ";
  check_success(volume_size.x > 0);
  check_success(volume_size.y > 0);
  check_success(volume_size.z > 0);

  mi::math::Bbox_struct<mi::Uint32, 3> ijk_bbox;
  ijk_bbox.min.x = 0;
  ijk_bbox.min.y = 0;
  ijk_bbox.min.z = 0;
  ijk_bbox.max.x = volume_size.x;
  ijk_bbox.max.y = volume_size.y;
  ijk_bbox.max.z = volume_size.z;

  // When a volume is created, we need to inform the IndeX library
  // how to create the volume. This is a volume creation
  // configuration. This configuration is also a factory to generate
  // a volume generator job.
 nv::index_common::Synthetic_volume_generator* generator = new nv::index_common::Synthetic_volume_g
   ijk_bbox, nv::index_common::Synthetic_volume_generator::VOXEL_FORMAT_UINT_8, synthetic_volume_ty
  check_success(generator);

  check_success(dice_transaction != 0);
  check_success(colormap_tag.is_valid());
```

```
  // Create the volume scene element
  mi::base::Handle<nv::index::IRegular_volume> volume(
   scene->create_volume(scale, rotate_k, translate, volume_size, generator, dice_transaction));

  if(!volume.is_valid_interface())
  {
    ERROR_LOG << fn << "failed to create a volume.";
    return mi::neuraylib::NULL_TAG;
  }

  // Assign colormap
  volume->assign_colormap(colormap_tag);

  // Set the volume name
  volume->set_name(volume_name.c_str());

  // Enable rendering initially depending on global state
  volume->set_enabled(is_rendering_enabled);

  // Currently no clipping support for this example code.
  //   volume->set_IJK_region_of_interest(clip_bbox);

  DEBUG_LOG << "setup_imported_volume: name: "  << volume_name
     << ", size: "    << volume_size.x << " " << volume_size.y << " " << volume_size.z << " "
     << ", scale: "    << scale.x     << " " << scale.y     << " " << scale.z     << " "
       << ", rotate_k: "                    << rotate_k
     << ", translate: " << translate.x  << " "  << translate.y << " "  << translate.z  << " "
       << ", colormap: "                    << colormap_tag.id
       << ", is_rendering_enabled: "        << is_rendering_enabled
    ;

  mi::neuraylib::Tag volume_tag =
    dice_transaction->store_for_reference_counting(volume.get());

  if(!volume_tag.is_valid()){
    ERROR_LOG << fn << "fail to set up volume.";
    return mi::neuraylib::NULL_TAG;
  }

  return volume_tag;
}

static void setup_camera(
  nv::index::IPerspective_camera*          cam,
  std::map< std::string, std::string >&    opt_map,
  const mi::math::Vector< mi::Uint32, 3 >& volume_size)
{
  cam->set_aperture(0.033f);
  cam->set_focal(0.03f);

  const std::string resolution_x_str = opt_map["resolution_x"];
  const std::string resolution_y_str = opt_map["resolution_y"];
  mi::math::Vector< mi::Sint32, 2 > res(1024, 1024);
  res.x = nv::index::app::get_sint32(resolution_x_str);
  res.y = nv::index::app::get_sint32(resolution_y_str);

  check_success((res.x > 0) && (res.y > 0));
```

```
  const mi::Float32 pix_aspect_ratio_1x1 = 1.0f;
  // static_cast< mi::Float32 >(res.x) / static_cast< mi::Float32 >(res.y);

  cam->set_aspect(pix_aspect_ratio_1x1);

  // get top view for this scene
  mi::math::Bbox< mi::Float32, 3 > const
    vbbox(0.0f, 0.0f, 0.0f,
     static_cast< mi::Float32>(volume_size.x), static_cast< mi::Float32>(volume_size.y),
       static_cast< mi::Float32>(volume_size.z));
  mi::math::Vector< mi::Float32, 3 > from(0.0f, 0.0f, 0.0f);
  mi::math::Vector< mi::Float32, 3 > to  (0.0f, 0.0f, 0.0f);
  mi::math::Vector< mi::Float32, 3 > up  (0.0f, 0.0f, 0.0f);
  mi::Float32 clip_min =  0.1f;
  mi::Float32 clip_max = 10.0f;
  get_top_view_param(vbbox,
            static_cast<mi::Float32>(cam->get_aspect()),
            static_cast<mi::Float32>((cam->get_fov_y_rad() / 2.0)),
            from, to, up, clip_min, clip_max);

  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();
  cam->set(from, viewdir, up);
  cam->set_clip_min(clip_min);
  cam->set_clip_max(clip_max);
}


static bool setup_main_host(Nvindex_access & nvindex_accessor,
              App_rendering_context & arc,
              App_data & app_dat,
              std::map< std::string, std::string > & opt_map)
{
  // volume size
  std::map< std::string, mi::math::Vector< mi::Uint32, 3 > > volume_size_map;
  // small (/ (* 2039.0 2039.0 2039.0) 2048.0 2048.0 2048.0) = 8 GB
  // med   (/ (* 4039.0 4039.0 2039.0) 1024.0 1024.0 1024.0) = 32 GB
  // large (/ (* 6569.0 6569.0 2039.0) 1024.0 1024.0 1024.0) = 82 GB
  volume_size_map["unittest"] = mi::math::Vector< mi::Uint32, 3 >(  20,   20,   20);
  volume_size_map["small"]    = mi::math::Vector< mi::Uint32, 3 >(2039, 2039, 2039);
  volume_size_map["med"]      = mi::math::Vector< mi::Uint32, 3 >(4039, 4039, 2039);
  volume_size_map["large"]    = mi::math::Vector< mi::Uint32, 3 >(6569, 6569, 2039);

  arc.m_database =
    nvindex_accessor.get_interface()->get_api_component<mi::neuraylib::IDatabase>();
  check_success(arc.m_database.is_valid_interface());

  arc.m_global_scope = arc.m_database->get_global_scope();
  check_success(arc.m_global_scope.is_valid_interface());

 // ------------------------------------------------------------------------------------
  // Access the IndeX configurations to create a session
  arc.m_iindex_session =
   nvindex_accessor.get_interface()->get_api_component<nv::index::IIndex_session>();
  check_success(arc.m_iindex_session.is_valid_interface());
```

```
  // Access the IndeX rendering interface for rendering
  arc.m_iindex_rendering =
    nvindex_accessor.get_interface()->create_rendering_interface();
  check_success(arc.m_iindex_rendering.is_valid_interface());

 // Access the IndeX rendering query interface for querying performance values and pick results
  arc.m_icluster_configuration =
   nvindex_accessor.get_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(arc.m_icluster_configuration.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(arc.is_local_host_joined());

  // DiCE database access
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    arc.m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    // Setup information
    arc.m_session_tag =
      arc.m_iindex_session->create_session(dice_transaction.get());
    check_success(arc.m_session_tag.is_valid());
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<const nv::index::ISession>(arc.m_session_tag));
    check_success(session.is_valid_interface());
    mi::base::Handle<nv::index::IScene> scene(
      dice_transaction->edit<nv::index::IScene>(session->get_scene()));

    //------------------------------------------------------------------------
    // Scene setup
    //   Add a synthetic volume to the scene.
   mi::Sint32 const colormap_entry_id = 1; // same as demo application's colormap file 1
   app_dat.m_colormap_tag = create_colormap(colormap_entry_id, scene.get(), dice_transaction.get())
    check_success(app_dat.m_colormap_tag.is_valid());

    // volume size
    check_success(!(opt_map["unittest"].empty()));
    check_success(!(opt_map["dataset_size"].empty()));
    std::string dataset_size_id = opt_map["dataset_size"];
    if(opt_map["unittest"] == "1"){
      dataset_size_id = "unittest";
      INFO_LOG << "Activated unittest mode.";
    }
    if(volume_size_map.find(dataset_size_id) == volume_size_map.end()){
    ERROR_LOG << "no such dataset_size [" + dataset_size_id + "], use 2039x2039x2039.";
      dataset_size_id = "small";
    }
   const mi::math::Vector< mi::Uint32, 3 > volume_size = volume_size_map[dataset_size_id];

    const std::string synthetic_volume_type_str = "sphere_0";
    const std::string synthetic_volume_parameter_str = "";
    const std::string volume_name = "synthetic_sphere_0";
    const mi::math::Vector<mi::Float32, 3> scale(1.0f, 1.0f, 1.0f);
    const mi::Float32 rotate_k = 0.0f;
    const mi::math::Vector<mi::Float32, 3> translate(0.0f, 0.0f, 0.0f);
    const bool is_rendering_enabled = true;
```

```
const mi::neuraylib::Tag volume_tag =
  create_synthetic_volume(
    app_dat.m_colormap_tag,
    scene.get(),
    synthetic_volume_type_str,
    synthetic_volume_parameter_str,
    volume_size,
    volume_name,
    scale,
    rotate_k,
    translate,
    is_rendering_enabled,
    dice_transaction.get());

check_success(volume_tag.is_valid());

// Add the new volume to the hierarchical scene description
scene->append(volume_tag, dice_transaction.get());

INFO_LOG << "Creating a synthetic volume: size = [" << volume_size.x
   << " " << volume_size.y << " " << volume_size.z << "], tag = " << volume_tag.id;
INFO_LOG << "This will take a while...";

// Create and edit a camera. Data distribution is based on the
// camera. (Because only visible massive data are considered)
mi::base::Handle< nv::index::IPerspective_camera > cam(
  scene->create_camera<nv::index::IPerspective_camera>());
check_success(cam.is_valid_interface());
setup_camera(cam.get(), opt_map, volume_size);

app_dat.m_camera_tag = dice_transaction->store(cam.get());
check_success(app_dat.m_camera_tag.is_valid());

const std::string resolution_x_str = opt_map["resolution_x"];
const std::string resolution_y_str = opt_map["resolution_y"];
mi::math::Vector< mi::Uint32, 2 > res(1024, 1024);
res.x = nv::index::app::get_uint32(resolution_x_str);
res.y = nv::index::app::get_uint32(resolution_y_str);
arc.m_canvas.set_resolution(res);

// Set up the scene
mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
  { 0.0f, 0.0f, 0.0f, },
  { static_cast< mi::Float32 >(volume_size.x),
   static_cast< mi::Float32 >(volume_size.y),
   static_cast< mi::Float32 >(volume_size.z), },
};

// set the region of interest
const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
check_success(xyz_roi.is_volume());
scene->set_clipped_bounding_box(xyz_roi_st);

// Set the scene global transformation matrix.
// only change the coordinate system
mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
```

```
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
      0.0f,  0.0f,  0.0f,  1.0f
      );
    scene->set_transform_matrix(transform_mat);

    // Set the current camera to the scene.
    check_success(app_dat.m_camera_tag.is_valid());
    scene->set_camera(app_dat.m_camera_tag);
  }
  dice_transaction->commit();

  INFO_LOG << "Initialization complete.";

  return true;
}

static nv::index::IFrame_results* render_frame(
  Nvindex_access&                  nvindex_accessor,
  App_rendering_context&           arc,
  const std::string&               output_fname)
{
  check_success(arc.m_iindex_rendering.is_valid_interface());

  // set output filename, empty string is valid
  arc.m_canvas.set_save_filename(output_fname);

  check_success(arc.m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    arc.m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  arc.m_iindex_session->update(arc.m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    arc.m_iindex_rendering->render(
      arc.m_session_tag,
      &arc.m_canvas,
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

static bool update_scene_and_render_frame(
  Nvindex_access & nvindex_accessor,
  App_rendering_context & arc,
  App_data & app_dat,
  mi::Sint32 frame_idx,
  mi::Sint32 max_frame_count,
  std::map< std::string, std::string > & opt_map)
{
```

```
  bool success = true;

  // Change the scene element, camera, etc. here. This example
  // does not change any, but usually you can change camera
  // parameters, add/remove scene elements here.

  // Render a frame and save the rendered image to a file.
  // Only save the file at the end of the iteration.
  std::string fname = "";
  if (frame_idx == max_frame_count)
  {
    fname = get_output_file_name(opt_map["outfname"], frame_idx);
  }

  // Performance value. Rendering statistics.
  mi::base::Handle<nv::index::IFrame_results> frame_results(
    render_frame(nvindex_accessor, arc, fname));
 const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
  if (err_set->any_errors())
  {
    std::ostringstream os;
   const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    const mi::Uint32 nb_err = err_set->get_nb_errors();
    for (mi::Uint32 e = 0; e < nb_err; ++e)
    {
      if (e != 0) os << '\n';
      const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
      os << err->get_error_string();
    }

   ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
         << os.str();

    success = false;
  }
  else
  {
    if (frame_idx==0)
    {
      // Print the performance header right
     INFO_LOG << "perf: Note: Not all the first frame's performance values are correct "
        << "due to some performance values depend on the last frame value, or scene creation.";
      INFO_LOG << app_dat.peek_performance_logger()->get_performance_header();
    }

    const mi::base::Handle<nv::index::IPerformance_values> performance_values(
      frame_results->get_performance_values());
    INFO_LOG << app_dat.peek_performance_logger()->
      get_performance_string(performance_values.get());
  }

  return success;
}

static bool mainhost_rendering_loop(
  Nvindex_access & nvindex_accessor,
  App_rendering_context & arc,
```

```
  App_data & app_dat,
  std::map< std::string, std::string > & opt_map)
{
  check_success(!(opt_map["max_iter"].empty()));
  const std::string max_iter_str = opt_map["max_iter"];
  mi::Sint32 const max_iter = nv::index::app::get_sint32(max_iter_str);
  check_success(max_iter > 0);
  mi::Sint32 const max_frame_count = max_iter - 1;
  INFO_LOG << "set max iteration: " << max_iter_str;

  // rendering loop
  bool is_ok = true;
  for(mi::Sint32 i = 0; i < max_iter; ++i)
  {
    is_ok = update_scene_and_render_frame(nvindex_accessor,
                      arc, app_dat, i, max_frame_count, opt_map);
    if(!is_ok){
      break;
    }
  }

  INFO_LOG << "Finished main host rendering loop.";

  return is_ok;
}

static void config_automatic_span_control(App_rendering_context & arc,
                     std::map< std::string, std::string > & opt_map)
{
  check_success(!(opt_map["index::enable_automatic_span_control"].empty()));
 const std::string enable_auto_span_str = opt_map["index::enable_automatic_span_control"];
  bool const is_enable_auto_span = nv::index::app::get_bool(enable_auto_span_str);

  check_success(!(opt_map["index::max_span_per_machine"].empty()));
 const std::string max_span_per_machine_str = opt_map["index::max_span_per_machine"];
 mi::Sint32 const max_span_per_machine = nv::index::app::get_sint32(max_span_per_machine_str);
  check_success(max_span_per_machine > 0);

  INFO_LOG << "index::enable_automatic_span_control: " << enable_auto_span_str
      << ", index::max_span_per_machine: " << max_span_per_machine_str;

  check_success(arc.m_global_scope.is_valid_interface());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    arc.m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    check_success(arc.m_session_tag.is_valid());
    mi::base::Handle< nv::index::ISession const > session(
      dice_transaction->access< nv::index::ISession const >(
        arc.m_session_tag));
    check_success(session.is_valid_interface());

    mi::base::Handle<nv::index::IConfig_settings> edit_config_settings(
      dice_transaction->edit<nv::index::IConfig_settings>(
        session->get_config()));
    check_success(edit_config_settings.is_valid_interface());
```

```
    // automatic span control configuration
    edit_config_settings->set_max_spans_per_machine(max_span_per_machine);
    edit_config_settings->set_automatic_span_control(is_enable_auto_span);
  }
  dice_transaction->commit();
}

void config_performance_monitoring(App_rendering_context & arc,
                  std::map< std::string, std::string > & opt_map)
{
  check_success(arc.m_global_scope.is_valid_interface());
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    arc.m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    check_success(arc.m_session_tag.is_valid());
    mi::base::Handle< nv::index::ISession const > session(
      dice_transaction->access< nv::index::ISession const >(
        arc.m_session_tag));
    check_success(session.is_valid_interface());

    mi::base::Handle<nv::index::IConfig_settings> edit_config_settings(
      dice_transaction->edit<nv::index::IConfig_settings>(
        session->get_config()));
    check_success(edit_config_settings.is_valid_interface());

    // performance related configuration examples.
    edit_config_settings->set_step_size_min(10);
    edit_config_settings->set_step_size_max(10);

    if(opt_map["compositing_mode"] == "0")
    edit_config_settings->set_compositing_mode(nv::index::IConfig_settings::COMPOSITING_ALL);
    else if(opt_map["compositing_mode"] == "1")
    edit_config_settings->set_compositing_mode(nv::index::IConfig_settings::COMPOSITING_LOCAL_ONLY
    else if(opt_map["compositing_mode"] == "2")
    edit_config_settings->set_compositing_mode(nv::index::IConfig_settings::COMPOSITING_REMOTE_ONL

    //   edit_config_settings->set_parallel_rendering_and_compositing(is_parallel);
    //   edit_config_settings->set_size_of_rendering_results_in_queue(1);
    edit_config_settings->set_monitor_performance_values(true);

    // compression configuration
    if(opt_map["compression"] == "1")
    {
    nv::index::IConfig_settings::Data_transfer_config cfg = edit_config_settings->get_data_transfe
    cfg.span_compression_level = 1;  // compression level: 1(fastest and sufficient)-9(slowest), 0(no
    cfg.tile_compression_level = 1;  // compression level: 1(fastest and sufficient)-9(slowest), 0(no
    cfg.span_image_encoding = true;  // enable/disable delta encoding for span compression (transfers
    cfg.tile_image_encoding = true;  // enable/disable delta encoding for tile compression (transfers
    cfg.span_alpha_channel =  false ; // 'true' ensures alpha values to the viewer and 'false' suppres
      edit_config_settings->set_data_transfer_config(cfg);
    }
    else
    {
    nv::index::IConfig_settings::Data_transfer_config cfg = edit_config_settings->get_data_transfe
    cfg.span_compression_level = 0;  // compression level: 1(fastest and sufficient)-9(slowest), 0(no
```

```
    cfg.tile_compression_level = 0;  // compression level: 1(fastest and sufficient)-9(slowest), 0(no
    cfg.span_image_encoding = false; // enable/disable delta encoding for span compression (transfers
    cfg.tile_image_encoding = false; // enable/disable delta encoding for tile compression (transfers
    cfg.span_alpha_channel =  true ; // 'true' ensures alpha values to the viewer and 'false' suppresse
      edit_config_settings->set_data_transfer_config(cfg);
    }
  }
  dice_transaction->commit();
}

int main(int argc, char* argv[])
{
  const std::string com_name(argv[0]);

  std::map< std::string, std::string > opt_map;
  opt_map["dice::verbose"]  = "3";                    // log level
 opt_map["dice::network::multicast_address"] = "224.1.3.2"; // default multicast address
 opt_map["max_iter"] = "200";                 // default max rendering loop iterations
  opt_map["resolution"] = "";                         // obsolete option
  opt_map["resolution_x"] = "1024";                   // resolution x
  opt_map["resolution_y"] = "1024";                   // resolution y
  opt_map["dataset_size"] = "small";                  // max number of spans
 opt_map["index::enable_automatic_span_control"] = "1"; // automatic span rendering (0...disabled)
 opt_map["index::max_span_per_machine"] = "12";    // max number of spans, do not exceed 12
 opt_map["dice::network::additional_unicast_sockets"] = "0"; // default number of unicast sockets
  opt_map["send_elements_only_to_owners"] = "0";    // default multicast to everyone
  opt_map["disk_cache_path"] = "";                   // default multicast to everyone
  opt_map["outfname"] = "frame_cluster";              // output file base name
  opt_map["unittest"] = "0";                          // default mode
  opt_map["compression"] = "1";                     // image data compression enabled
 opt_map["compositing_mode"] = "0";            // compositing mode: compositing on all machines
 opt_map["is_dump_comparison_image_when_failed"] = "1";  // default: dump images when failed.
 opt_map["is_call_from_test"] = "0";              // default: not call from make check.

  process_command_line(argc, argv, opt_map);

  if (nv::index::app::get_bool(opt_map["unittest"]))
  {
    if (nv::index::app::get_bool(opt_map["is_call_from_test"]))
    {
      opt_map["is_dump_comparison_image_when_failed"] = "0";
    }
    opt_map["outfname"] = "";  // turn off file output in the unit test mode
    opt_map["max_iter"] = "8"; // set unittest mode iterations
    opt_map["dice::verbose"] = "2";
  }
  info_cout(std::string("running ") + com_name, opt_map);
  info_cout(std::string("outfname = [") + opt_map["outfname"] +
       "], max_iter = "            + opt_map["max_iter"] +
       ", dice::verbose = "        + opt_map["dice::verbose"], opt_map);

  if(!(opt_map["resolution"].empty())){
   std::cout << "error: found obsolete option [resolution]. replace this with -resolution_x,\n"
         << "error: and -resolution_y. see -h option." << std::endl;
    exit(1);
  }
```

```
  // print help and exit if -h
  if(opt_map.find("h") != opt_map.end()){
    std::cout
      << "Usage: " + com_name + " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info.). (default: " + opt_map["dice::verbose"]
      << ")\n"
      << "        [-dice::network::multicast_address address]\n"
      << "            set multicast address. (default: "
      << opt_map["dice::network::multicast_address"] + ")\n"
      << "        [-max_iter number_of_iteration]\n"
      << "            set rendering loop iterations. (default: "
      << opt_map["max_iter"] + " frames)\n"
      << "        [-resolution_x int] (default: "
      << opt_map["resolution_x"] + ")\n"
      << "        [-resolution_y int] (default: "
      << opt_map["resolution_y"] + ")\n"
      << "            resolution hint\n"
      << "            1024x1024                 (aka baseline)\n"
      << "            1280x720  HD (HD 720)\n"
      << "            1920x1080 HD (HD 1080[ip]) (aka FHD)\n"
      << "            2048x1536 Double HD, or 2160p\n"
      << "            2560x1440 WQHD            (aka WQHD)\n"
      << "            3840x2160 4k              (aka QFHD)\n"
      << "            4096x3072 4k, QHD\n"
      << "            please make sure cuda memory size to have enough room\n"
      << "            for the framebuffer. Otherwise you will see out_of_memory.\n"
      << "        [-index::enable_automatic_span_control [0|1]\n"
      << "            1...enable automatic span control, 0...disable (default: "
      << opt_map["index::enable_automatic_span_control"] + ")\n"
      << "        [-index::max_span_per_machine int]\n"
      << "            number of max spans per machine (default: "
      << opt_map["index::max_span_per_machine"] + ")\n"
      << "        [-dice::network::additional_unicast_sockets int]\n"
      << "            set number of unicast sockets for viewer node. (default: "
      << opt_map["dice::network::additional_unicast_sockets"] + ")\n"
      << "        [-send_elements_only_to_owners bool]\n"
      << "            do not multicast data if set (default: "
      << opt_map["send_elements_only_to_owners"] + ")\n"
      << "        [-disk_cache_path string]\n"
      << "            path to a cache to flush data to (default: ["
      << opt_map["disk_cache_path"] + "])\n"
      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" + opt_map["outfname"] + "])\n"
      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << opt_map["unittest"] + "])\n"
      << "        [-compression bool]\n"
      << "            when false, all compression will be deactivated.\n"
      << "            Otherwise all compression is activated.\n"
      << "           (default: " + opt_map["compression"] + ")\n"
      << "        [-compositing_mode unsigned int]\n"
      << "            0: compositing on all machines,\n"
```

```
    << "            1: compositing on local machines only,\n"
    << "            2: compositing on remote machines only,\n"
    << "           (default: " + opt_map["compositing_mode"] + ")"
    << std::endl;
  exit(1);
}

Nvindex_access nvindex_accessor;
check_success(nvindex_accessor.load_library() == true);

start_nvindex_accessor(nvindex_accessor, opt_map);
if (!nvindex_accessor.is_initialized()) {
  printf("error: Fatal error! Initialization of the IndeX library failed.\n");
  return 1;
}

// required on host side too
nv::index::app::util::time::sleep(0.1); // wait for 0.1 second

// setup performance logger.
//   printing out performance item's keys
char const * const p_key[] = {
  // "nb_subcubes", "nb_subcubes_rendered", "size_volume_data_rendered",
  // "size_horizon_data_rendered", "nb_horizon_triangles_rendered",
  // "time_rendering_horizon", "time_rendering_volume",
  // "time_rendering_volume_and_horizon", "time_intersection_lines_points",
  // "time_rendering_only",
  "time_gpu_upload", // "time_gpu_download",
  // "time_rendering",
  // "time_rendering_total_sum",
  // "size_volume_data_upload", "size_rendering_results_download",
  // "size_pinned_host_memory", "size_unpinned_host_memory"
  // "size_gpu_memory",
  // "nb_fragments",
  // "is_using_gpu",
  "size_span_transfer",
  "size_span_transfer_compressed",
  "size_intermediate_image_transfer",
  "size_intermediate_image_transfer_compressed",
  // "size_intermediate_image_composited",
  // "time_compositing_stage",
  // "time_image_compositing",
  // "time_image_transfer",
  // "nb_composited_leafs", "nb_considered_leafs_per_span",
  "time_total_rendering",
  "time_total_compositing",
  // "time_total_final_compositing",
  "time_complete_frame",
  // "time_frame_setup", "time_frame_finish", "time_avg_rendering",
  "frames_per_second",
  // "nb_active_hosts",
  // "nb_horizontal_spans", //"nb_rendering_results_in_queue",
  0,
};

App_data app_dat;           // This has a performance logger.
std::vector< std::string > performance_key_vec;
```

```
  for(mi::Sint32 i = 0; p_key[i] != 0; ++i){
    performance_key_vec.push_back(p_key[i]);
  }
 app_dat.peek_performance_logger()->append_performance_item_key_list(performance_key_vec);

  // setup main host
  App_rendering_context arc;
  check_success(setup_main_host(nvindex_accessor, arc, app_dat, opt_map));

  // set automatic span control configuration
  config_automatic_span_control(arc, opt_map);

  // enable performance monitoring
  config_performance_monitoring(arc, opt_map);

  // call main host rendering loop
  mainhost_rendering_loop(nvindex_accessor, arc, app_dat, opt_map);

  // cleanup
  arc.shutdown();

  nvindex_accessor.shutdown();

  return 0;
}
```

9.3 cluster_rendering_mainhost.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/isparse_volume_rendering_properties.h>

#include <iostream>
#include <sstream>

#include <nv/index/app/forwarding_logger.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/index_connect.h>

class Cluster_rendering_mainhost:
  public nv::index::app::Index_connect
{
public:
  Cluster_rendering_mainhost()
    :
    Index_connect(),
    m_is_unittest(false),
    m_outfname(),
    m_max_iter(0)
  {
    // INFO_LOG << "DEBUG: Cluster_rendering_mainhost() ctor";
  }

  virtual ~Cluster_rendering_mainhost()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Cluster_rendering_mainhost() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  // override
  virtual bool evaluate_options(nv::index::app::String_dict& options) CPP11_OVERRIDE;
  // override for unittest support
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
```

```
    check_success(network_configuration != 0);

    if (m_is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

  // override
  virtual bool initialize_debug_configuration(
    mi::neuraylib::IDebug_configuration* debug_configuration,
    nv::index::app::String_dict&         options) CPP11_OVERRIDE
  {
    check_success(debug_configuration != 0);
    if (options.is_defined("dice::idebug_config::retention"))
    {
      bool conv_stat = false;
      const std::string retension = options.get("dice::idebug_config::retention");
    const mi::Sint32  debug_retention = nv::index::app::get_sint32(retension, &conv_stat);
      if (!conv_stat)
      {
     ERROR_LOG << "initialize_index_debug_configuration: failed integer conversion of dice::idebug_c
             << debug_retention;
        return false;
      }

      if(debug_retention > 0)
      {
        std::stringstream sstr;
        sstr << "retention=" << debug_retention;
        debug_configuration->set_option(sstr.str().c_str());
        INFO_LOG << "Set IDebug_configuration: " << sstr.str();
      }
    }

    return true;
  }

  // override
  virtual bool initialize_general_configuration(
    mi::neuraylib::IGeneral_configuration* general_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(general_configuration != 0);
   general_configuration->set_host_property("sub_cluster_id", "0"); // set default cluster id 0
    return true;
  }

private:
  // setup camera to see this example scene
  // \param[in] cam camera object
  void setup_camera(nv::index::IPerspective_camera* cam) const;
```

```
  // Create a synthetic volume.
  //
  // \note The new volume is not added to the hierachical scene description.
  //
  // \param[in] scene            scene root object to update the scene
  // \param[in] volume_size      volume size to be created
  // \param[in] dice_transaction dice db transaction
  // \return tag of the newly created volume
  mi::neuraylib::Tag create_synthetic_sparse_volume(
    nv::index::IScene*                    scene,
    const mi::math::Vector<mi::Uint32, 3>& volume_size,
    mi::neuraylib::IDice_transaction*     dice_transaction) const;

  // Add a synthetic height_field to the scene.
  //
  // \param[in] scene_edit        the scene to create the height field
  // \param[in] height_field_size height_field size
  // \param[in] dice_transaction  dice db transaction
  // \return true when succeeded
  mi::neuraylib::Tag create_synthetic_height_field(
    nv::index::IScene*                    scene_edit,
    const mi::math::Vector<mi::Uint32, 2>& height_field_size,
    mi::neuraylib::IDice_transaction*     dice_transaction) const;

  // set up as the main host
  //
  // \return true when success
  bool setup_main_host();

  // Render one frame
  //
  // \param[in] frame_idx       current frame index
  // \param[in] max_frame_count max frame number
  // \return true when success
  bool render_one_frame(
    mi::Sint32 frame_idx,
    mi::Sint32 max_frame_count) const;

  // main host rendering loop
  // \return true when success
  bool mainhost_rendering_loop() const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // camera
  mi::neuraylib::Tag                                                     m_camera_tag;
  // Application options
  bool                                                                  m_is_unittest;
  std::string                                                              m_outfname;
  mi::Sint32                                                               m_max_iter;
};

mi::Sint32 Cluster_rendering_mainhost::launch()
```

```
{
  setup_main_host();
  mainhost_rendering_loop();

  return 0;
}

bool Cluster_rendering_mainhost::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
    sdict.insert("verify_image_fname_sequence", ""); // disable this in unit test
  }

  m_outfname = sdict.get("outfname", "");
  m_max_iter = nv::index::app::get_sint32(sdict.get("max_iter", "0"));

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["       + m_outfname +
       "], max_iter = "     + nv::index::app::to_string(m_max_iter) +
       ", dice::verbose = " + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.get("h", "0") == "1")
  {
    std::cout
      << "info: Usage: " <<  com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            print out this message\n"

      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose") + ")\n"

      << "        [-dice::network::multicast_address address]\n"
    << "        set multicast address. (default: " << sdict.get("dice::network::multicast_address")

      << "        [-max_iter number_of_iteration]\n"
      << "            set rendering loop iterations. \n"
      << "            (default: " << sdict.get("max_iter") << " frames)\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When "", no output.\n"
      << "            The frame number and extension(.ppm) will be added.\n"
      << "            (default: " << sdict.get("outfname") << ")\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume).\n"
      << "            (default: " << sdict.get("unittest") << ")\n"
```

```
      << "        [-dice::idebug_config::retention INT]\n"
     << "          when set, set this idebug configuration option. not set when 0.\n"
     << "          (default: " << sdict.get("dice::idebug_config::retention") << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

void Cluster_rendering_mainhost::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  const mi::math::Vector<mi::Float32, 3> from(700, -1200, -300);
  const mi::math::Vector<mi::Float32, 3> to  (542.0f,  380.6f, -586.3f);
  const mi::math::Vector<mi::Float32, 3> up  (  0.0f,   0.0f,     1.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(100.0f);
  cam->set_clip_max(20000.0f);
}

mi::neuraylib::Tag Cluster_rendering_mainhost::create_synthetic_sparse_volume(
  nv::index::IScene*                    scene,
  const mi::math::Vector<mi::Uint32, 3>& volume_size,
  mi::neuraylib::IDice_transaction*     dice_transaction) const
{
  mi::math::Bbox<mi::Float32, 3> ijk_bbox;
  ijk_bbox.min.x = 0.0f;
  ijk_bbox.min.y = 0.0f;
  ijk_bbox.min.z = 0.0f;
  ijk_bbox.max.x = static_cast<mi::Float32>(volume_size.x);
  ijk_bbox.max.y = static_cast<mi::Float32>(volume_size.y);
  ijk_bbox.max.z = static_cast<mi::Float32>(volume_size.z);

  // sparse volume creation parameter
  nv::index::app::String_dict sparse_volume_opt;
  sparse_volume_opt.insert("args::type",     "sparse_volume");
  sparse_volume_opt.insert("args::importer", "synthetic");
  std::stringstream sstr;
  sstr << "0 0 0 " << volume_size.x << " " << volume_size.y << " " << volume_size.z;
  sparse_volume_opt.insert("args::bbox",           sstr.str());
  sparse_volume_opt.insert("args::voxel_format",   "uint8");
  sparse_volume_opt.insert("args::synthetic_type", "sphere_0");

  nv::index::IDistributed_data_import_callback* importer_callback =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::base_importer.Sparse_volume_generator_synthetic",
```

```
      sparse_volume_opt);

  // Create the sparse volume scene element
  mi::math::Matrix<mi::Float32, 4, 4>    transform_mat(1.0f); // Identity matrix
  const mi::math::Vector<mi::Float32, 3> scale(1.0f, 1.0f, 1.0f);
  const mi::Float32                      rotate_k = 0.0f;
  const mi::math::Vector<mi::Float32, 3> translate(0.0f, 0.0f, 0.0f);
  const bool                             is_rendering_enabled = true;

  mi::base::Handle<nv::index::ISparse_volume_scene_element> sparse_volume(
   scene->create_sparse_volume(ijk_bbox, transform_mat, importer_callback, dice_transaction));
  check_success(sparse_volume.is_valid_interface());
  sparse_volume->set_enabled(is_rendering_enabled);

  DEBUG_LOG << "setup_imported_volume: "
     << "size: "      << volume_size.x << " " << volume_size.y << " " << volume_size.z << " "
     << ", scale: "    << scale.x     << " " << scale.y     << " " << scale.z     << " "
       << ", rotate_k: "                    << rotate_k
     << ", translate: " << translate.x  << " "  << translate.y << " "  << translate.z  << " "
       << ", is_rendering_enabled: "       << is_rendering_enabled
     ;

  const mi::neuraylib::Tag sparse_volume_tag =
    dice_transaction->store_for_reference_counting(sparse_volume.get());
  check_success(sparse_volume_tag.is_valid());

  INFO_LOG << "Creating a synthetic volume: size = [" << volume_size.x
    << " " << volume_size.y << " " << volume_size.z << "], tag = " << sparse_volume_tag.id;
  INFO_LOG << "This will take a while...";

  return sparse_volume_tag;
}

mi::neuraylib::Tag Cluster_rendering_mainhost::create_synthetic_height_field(
  nv::index::IScene*                     scene_edit,
  const mi::math::Vector<mi::Uint32, 2>& height_field_size,
  mi::neuraylib::IDice_transaction*      dice_transaction) const
{
  const mi::math::Vector<mi::Float32, 2> height_field_value_range(0.0f, 1000.0f);
  const mi::math::Vector<mi::Float32, 3> height_field_scale(1.0f, 1.0f, 1.0f);
  const mi::Float32                      height_field_rotate_k           = 0.0f;
  const mi::math::Vector<mi::Float32, 3> height_field_translate(0.0f, 0.0f, 400.0f);
  check_success (dice_transaction != 0);

  // height_field creation option
  nv::index::app::String_dict hight_field_opt;
  hight_field_opt.insert("args::type",     "height_field");
  hight_field_opt.insert("args::importer", "synthetic");
  {
    std::stringstream sstr;
    sstr << height_field_size.x << " " << height_field_size.y;
    hight_field_opt.insert("args::size", sstr.str());
  }
  {
    std::stringstream sstr;
    sstr << height_field_value_range.x << " " << height_field_value_range.y;
    hight_field_opt.insert("args::range", sstr.str());
```

```
  }
  hight_field_opt.insert("args::synthetic_type", "i");

  nv::index::IDistributed_data_import_callback* importer_callback =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::base_importer.Height_field_generator_synthetic",
      hight_field_opt);
  check_success(importer_callback);

 mi::base::Handle<nv::index::IHeight_field_scene_element> height_field_scene_element(
    scene_edit->create_height_field(
      height_field_scale, height_field_rotate_k, height_field_translate,
      height_field_size,
      height_field_value_range,
      importer_callback,
      dice_transaction));
  check_success(height_field_scene_element.is_valid_interface());

  DEBUG_LOG << "create_synthetic_height_field: "
       << "size: "                        << height_field_size
       << ", scale: "                     << height_field_scale
       << ", rotate_k: "                  << height_field_rotate_k
       << ", translate: "                 << height_field_translate
       << ", (value) range: "             << height_field_value_range
    ;

  const mi::neuraylib::Tag height_field_tag =
   dice_transaction->store_for_reference_counting(height_field_scene_element.get());
  check_success(height_field_tag.is_valid());

  return height_field_tag;
}

bool Cluster_rendering_mainhost::setup_main_host()
{
 // Access the IndeX rendering query interface for querying performance values and pick results
  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  // DiCE database access
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    // Setup session information
    m_session_tag =
      m_index_session->create_session(dice_transaction.get());
```

```
check_success(m_session_tag.is_valid());
mi::base::Handle<const nv::index::ISession> session(
  dice_transaction->access<nv::index::ISession>(
    m_session_tag));
check_success(session.is_valid_interface());

//------------------------------------------------------------------------
// Scene setup
mi::base::Handle<nv::index::IScene> scene_edit(
  dice_transaction->edit<nv::index::IScene>(session->get_scene()));

// Create static group node for large data
mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
  scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
check_success(static_group_node.is_valid_interface());

// Added a light and a material to the static group node
{
  // Add a light to the scene
  mi::base::Handle<nv::index::IDirectional_headlight> headlight(
    scene_edit->create_attribute<nv::index::IDirectional_headlight>());
  check_success(headlight.is_valid_interface());
  const mi::math::Color color_intensity(1.f, 1.f, 1.f, 1.f);
  headlight->set_intensity(color_intensity);
 headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));

const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headlig
  check_success(headlight_tag.is_valid());
  static_group_node->append(headlight_tag, dice_transaction.get());
  INFO_LOG << "Created a headlight: tag = " << headlight_tag.id;

  // Add a material for the height_field to the scene
mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_
  check_success(phong_1.is_valid_interface());

  phong_1->set_ambient(mi::math::Color(0.f, 0.3f, 0.0f, 0.3f));
  phong_1->set_diffuse(mi::math::Color(0.f, 0.8f, 0.2f, 0.3f));
  phong_1->set_specular(mi::math::Color(0.6f));
  phong_1->set_shininess(100.f);
  const mi::neuraylib::Tag phong_1_tag
    = dice_transaction->store_for_reference_counting(phong_1.get());
  check_success(phong_1_tag.is_valid());
  static_group_node->append(phong_1_tag, dice_transaction.get());
  INFO_LOG << "Created a phong material: tag = " << phong_1_tag.id;

  // Add a sparse_volume_render_properties to the scene.
mi::base::Handle<nv::index::ISparse_volume_rendering_properties> sparse_render_prop(
 scene_edit->create_attribute<nv::index::ISparse_volume_rendering_properties>());
  sparse_render_prop->set_filter_mode(nv::index::SPARSE_VOLUME_FILTER_NEAREST);
  sparse_render_prop->set_sampling_distance(            1.0);
  sparse_render_prop->set_reference_sampling_distance(  1.0);
sparse_render_prop->set_voxel_offsets(          mi::math::Vector<mi::Float32, 3>(0.0f, 0.0f,
  sparse_render_prop->set_preintegrated_volume_rendering(false);
  sparse_render_prop->set_lod_rendering_enabled(        false);
  sparse_render_prop->set_lod_pixel_threshold(          2.0);
  sparse_render_prop->set_debug_visualization_option(   0);
  const mi::neuraylib::Tag sparse_render_prop_tag
```

```
      = dice_transaction->store_for_reference_counting(sparse_render_prop.get());
    check_success(sparse_render_prop_tag.is_valid());
    static_group_node->append(sparse_render_prop_tag, dice_transaction.get());
   INFO_LOG << "Created a sparse_render_prop_tag: tag = " << sparse_render_prop_tag;

    // Add a colormap to the scene.
  const mi::Sint32 colormap_entry_id = 1; // same as demo application's colormap file 1
    const mi::neuraylib::Tag colormap_tag =
     create_colormap(colormap_entry_id, scene_edit.get(), dice_transaction.get());
    check_success(colormap_tag.is_valid());
    static_group_node->append(colormap_tag, dice_transaction.get());
  }

  // mi::math::Vector<mi::Uint32, 3> volume_size(1020, 1020, 1020);
  mi::math::Vector<mi::Uint32, 3> volume_size(600, 64, 64);
  if (m_is_unittest)
  {
    volume_size = mi::math::Vector<mi::Uint32, 3>(64, 64, 64);
  }

  const mi::neuraylib::Tag volume_tag =
    create_synthetic_sparse_volume(
      scene_edit.get(),
      volume_size,
      dice_transaction.get());
  check_success(volume_tag.is_valid());

  // Add the new volume to the hierachical scene description
  static_group_node->append(volume_tag, dice_transaction.get());

  // Add a synthetic heightfield to the scene.
 const mi::math::Vector<mi::Uint32, 2> height_field_size(volume_size.x, volume_size.y);
  const mi::neuraylib::Tag height_field_tag =
    create_synthetic_height_field(
      scene_edit.get(),
      height_field_size,
      dice_transaction.get());
  check_success(height_field_tag.is_valid());
  static_group_node->append(height_field_tag, dice_transaction.get());
  INFO_LOG << "Created a synthetic height_field: size = ["
      << height_field_size.x << " " << height_field_size.y
      << "], tag = " << height_field_tag.id;

  // All scene elements are set up, store to the DB
  const mi::neuraylib::Tag static_group_node_tag =
    dice_transaction->store_for_reference_counting(static_group_node.get());
  check_success(static_group_node_tag.is_valid());
  INFO_LOG << "Created a static group node: tag = " << static_group_node_tag.id;
  scene_edit->append(static_group_node_tag, dice_transaction.get());

  // Create and edit a camera.
  mi::base::Handle< nv::index::IPerspective_camera > cam(
    scene_edit->create_camera<nv::index::IPerspective_camera>());
  check_success(cam.is_valid_interface());
  setup_camera(cam.get());

  m_camera_tag = dice_transaction->store(cam.get());
```

```
    const mi::math::Vector<mi::Uint32, 2> canvas_resolution(512, 512);
    m_image_file_canvas->set_resolution(canvas_resolution);

    // Set up the scene
    const mi::math::Bbox<mi::Float32, 3> xyz_roi_st(
      0.0f, 0.0f, 0.0f,
      static_cast<mi::Float32>(volume_size.x),
      static_cast<mi::Float32>(volume_size.y),
      static_cast<mi::Float32>(volume_size.z));

    // set the region of interest
    const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
    check_success(xyz_roi.is_volume());
    scene_edit->set_clipped_bounding_box(xyz_roi_st);

    // Set the scene global transformation matrix.
    // only change the coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
      0.0f,  0.0f,  0.0f,  1.0f
      );
    scene_edit->set_transform_matrix(transform_mat);

    // Set the current camera to the scene.
    check_success(m_camera_tag.is_valid());
    scene_edit->set_camera(m_camera_tag);
  }
  dice_transaction->commit();

  INFO_LOG << "Initialization complete.";

  return true;
}

bool Cluster_rendering_mainhost::render_one_frame(
  mi::Sint32 frame_idx,
  mi::Sint32 max_frame_count) const
{
  bool success = true;

  // Change the scene element, camera, etc. here. This example
  // does not change any, but usually you can change camera
  // parameters, add/remove scene elements here.

  // Render a frame and save the rendered image to a file.
  // Only save the file at the end of the iteration.
  std::string fname = "";
  if (frame_idx == max_frame_count)
  {
    fname = get_output_file_name(m_outfname, frame_idx);
  }

  // set up canvas. output_fname.empty() is valid (no output file)
  m_image_file_canvas->set_rgba_file_name(fname.c_str());
```

```
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  // Check the rendering result.
 const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
  if (err_set->any_errors())
  {
    std::ostringstream os;
   const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    const mi::Uint32 nb_err = err_set->get_nb_errors();
    for (mi::Uint32 e = 0; e < nb_err; ++e)
    {
      if (e != 0) os << '\n';
      const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
      os << err->get_error_string();
    }

   ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
         << os.str();
    success = false;
  }

  return success;
}

bool Cluster_rendering_mainhost::mainhost_rendering_loop() const
{
  check_success(m_max_iter > 0);
  const mi::Sint32 max_frame_count = m_max_iter - 1;

  // rendering loop
  bool is_ok = true;
  for(mi::Sint32 i = 0; i < m_max_iter; ++i)
  {
    is_ok = render_one_frame(i, max_frame_count);
    if (!is_ok)
    {
      break;
    }
    INFO_LOG << "frame: " << i;
  }

  INFO_LOG << "Finished main host rendering loop.";
```

```
  return is_ok;
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;

  // Application (Cluster_rendering_mainhost) settings
  sdict.insert("dice::verbose", "3");                              // log level
  sdict.insert("dice::network::mode", "UDP");                      // network mode
 sdict.insert("dice::network::multicast_address", "224.1.3.2"); // multicast address
 sdict.insert("max_iter", "10");                       // max rendering loop iterations
 sdict.insert("outfname", "frame_cluster_rendering_mainhost");  // output file base name
  sdict.insert("unittest", "0");                                   // default mode
  sdict.insert("dice::idebug_config::retention", "0");             // retention
 sdict.insert("is_dump_comparison_image_when_failed", "1");    // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                 // default: not call from make check.

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "yes");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "no");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io");
  // plugin: base_importer
  sdict.insert("index::app::plugins::base_importer::enabled", "true");

  Cluster_rendering_mainhost cluster_rendering_mainhost;
  cluster_rendering_mainhost.initialize(argc, argv, sdict);
  check_success(cluster_rendering_mainhost.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = cluster_rendering_mainhost.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.4 cluster_rendering_remotehost.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/iindex.h>

#include <iostream>
#include <sstream>
#include <map>
#include <stdlib.h>

#include "utility/app_rendering_context.h"

#include <nv/index/app/forwarding_logger.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>
#include <nv/index/app/index_connect_.h> // FIXME should be updated to index_connect.h

namespace {                     // anonymous namespace for the local functions

class Cluster_rendering_remotehost_index_connect:
    public nv::index::app::Index_connect
{
public:
  Cluster_rendering_remotehost_index_connect()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Cluster_rendering_remotehost_index_connect() ctor";
  }

  virtual ~Cluster_rendering_remotehost_index_connect()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Cluster_rendering_remotehost_index_connect() dtor";
  }

protected:
  // override for unittest support
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
```

```
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

  // override
  virtual bool register_serializable_classes(
    mi::neuraylib::IDice_configuration* configuration_interface,
    nv::index::app::String_dict&        options) CPP11_OVERRIDE
  {
    // bool is_registered = false;
   // is_registered = get_index_interface()->register_serializable_class<Arrow_manipulator>();
    // check_success(is_registered);
    // return is_registered;
    return true;
  }

  // override
  virtual bool initialize_debug_configuration(
    mi::neuraylib::IDebug_configuration* debug_configuration,
    nv::index::app::String_dict&         options) CPP11_OVERRIDE
  {
    check_success(debug_configuration != 0);
    if (options.is_defined("dice::idebug_config::retention"))
    {
      bool conv_stat = false;
      const std::string retension = options.get("dice::idebug_config::retention");
    const mi::Sint32  debug_retention = nv::index::app::get_sint32(retension, &conv_stat);
      if (!conv_stat)
      {
     ERROR_LOG << "initialize_index_debug_configuration: failed integer conversion of dice::idebug_c
             << debug_retention;
        return false;
      }

      if(debug_retention > 0)
      {
        std::stringstream sstr;
        sstr << "retention=" << debug_retention;
        debug_configuration->set_option(sstr.str().c_str());
        INFO_LOG << "Set IDebug_configuration: " << sstr.str();
      }
    }

    return true;
  }

  // override
  virtual bool initialize_general_configuration(
    mi::neuraylib::IGeneral_configuration* general_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(general_configuration != 0);
   general_configuration->set_host_property("sub_cluster_id", "0"); // set default cluster id 0
    return true;
  }
};
```

```
bool remote_rendering_loop(
  Cluster_rendering_remotehost_index_connect& index_connect,
  App_rendering_context&                      arc,
  nv::index::app::String_dict&                sdict)
{
  mi::base::Handle<nv::index::ICluster_configuration> rendering_properties_query(
   index_connect.get_index_interface()->get_api_component<nv::index::ICluster_configuration>());
  check_success(rendering_properties_query.is_valid_interface());

  mi::Uint32 number_of_hosts = rendering_properties_query->get_number_of_hosts();

  std::stringstream sstr;
  sstr << "*****************************************************************\n"
    << "info: NVIDIA IndeX remote service running and waiting for connections,\n"
   << "info: Cluster host id = " << rendering_properties_query->get_local_host_id() << "\n"
    << "info: ***********************************************************";
  INFO_LOG << sstr.str();

  check_success(sdict.is_defined("unittest"));

  const bool is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));
  const mi::Sint32 remote_max_iter_when_unittest = 8;

  for(mi::Sint32 i = 0; true; ++i)
  {
   const mi::Uint32 new_number_of_hosts = rendering_properties_query->get_number_of_hosts();

    // Finish on lost host
    if (new_number_of_hosts < number_of_hosts)
    {
      std::stringstream sstr;
    sstr << "Leaving the NVIDIA IndeX remote service loop because at least one other cluster host has l
        << "Previously, the cluster was composed by "
        << number_of_hosts << " hosts. Now, only "
        << new_number_of_hosts << " cluster hosts are left.";
      INFO_LOG << sstr.str();

      break;
    }

    // exit if unittest mode with max iters
    if (is_unittest&& (i>= remote_max_iter_when_unittest))
    {
    INFO_LOG << "The unit test exits because the test has reached the maximum number of iterations.";

      break;
    }

    number_of_hosts = new_number_of_hosts;
    nv::index::app::util::time::sleep(0.1f); // wait for 0.1 second
  }

  INFO_LOG << "Finished the NVIDIA IndeX remote service loop.";

  return true;
}
```

```
bool setup_remote_host(
  Cluster_rendering_remotehost_index_connect& index_connect,
  App_rendering_context&                      arc)
{
 arc.m_database = index_connect.get_index_interface()->get_api_component<mi::neuraylib::IDatabase>
  check_success(arc.m_database.is_valid_interface());

  arc.m_global_scope = arc.m_database->get_global_scope();
  check_success(arc.m_global_scope.is_valid_interface());

  return true;
}

void config_network_parameters(
  Cluster_rendering_remotehost_index_connect& index_connect,
  nv::index::app::String_dict&                sdict)
{
  if (sdict.get("dice::network::additional_unicast_sockets", "0") != "0")
  {
    mi::base::Handle<mi::neuraylib::IDebug_configuration> debug_configuration(
    index_connect.get_index_interface()->get_api_component<mi::neuraylib::IDebug_configuration>()
    check_success(debug_configuration.is_valid_interface());

    std::stringstream sstr;
    sstr << "dice::network::additional_unicast_sockets="
      << sdict.get("dice::network::additional_unicast_sockets", "<undef>");
    const std::string additional_unicast_sockets = sstr.str();
    debug_configuration->set_option(additional_unicast_sockets.c_str());
  }
}

} // namespace anonymous

int main(int argc, char* argv[])
{
  const std::string com_name(argv[0]);
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                              // log level
  sdict.insert("dice::network::mode", "UDP");                      // network mode
 sdict.insert("dice::network::multicast_address", "239.43.1.1"); // multicast address
 sdict.insert("dice::network::additional_unicast_sockets", "0"); // default number of unicast socke
  sdict.insert("unittest", "0");                                   // default mode
  sdict.insert("dice::idebug_config::retention", "0");             // retention
 sdict.insert("is_dump_comparison_image_when_failed", "1");    // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                 // default: not call from make check.

  Cluster_rendering_remotehost_index_connect index_connect;
  index_connect.process_command_line_arguments(argc, argv, sdict);

  if (nv::index::app::get_bool(sdict.get("unittest", "false")))
  {
   // try to randomize the multicast address for the test purpose. This does not guarantee, but practic
   mi::Uint32 cur_time = static_cast<mi::Uint32>(nv::index::app::util::time::get_time() * 10000);
    srand(cur_time);
    mi::Sint64 r1 = (rand() % 254) + 1;
    mi::Sint64 r2 = (rand() % 254) + 1;
```

```
    std::stringstream sstr;
    sstr << "239.43." << r1 << "." << r2;
    std::string mc_addr = sstr.str();

    std::stringstream sstr2;
    sstr2 << "Cur_time: " << cur_time << ", Using multicast address: " << mc_addr;
    info_cout(sstr2.str(), sdict);

    sdict.insert("dice::network::multicast_address", mc_addr);

    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.get("dice::verbose", "2");
  }
  info_cout("running " + com_name, sdict);
  info_cout("dice::verbose = " + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h/-help
  if (sdict.is_defined("h") || sdict.is_defined("help"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"

      << "        [-dice::verbose severity_level]\n"
      << "            verbose severity level (3 is info.).\n"
      << "            (default: " + sdict.get("dice::verbose", "<undef>") + ")\n"

      << "        [-dice::network::multicast_address address]\n"
      << "            set multicast address.\n"
    << "        (default: " << sdict.get("dice::network::multicast_address", "<undef>") << ")\n"

      << "        [-dice::network::additional_unicast_sockets int]\n"
      << "            set number of unicast sockets for remote host.\n"
    << "        (default: " << sdict.get("dice::network::additional_unicast_sockets", "<undef>") <<

      << "        [-unittest bool]\n"
      << "            when test true, unit test mode.\n"
      << "            (default: " << sdict.get("unittest", "<undef>") << ")"

      << "        [-dice::idebug_config::retention int]\n"
     << "           when set, set this idebug configuration option. not set when 0.\n"
    << "        (default: " << sdict.get("dice::idebug_config::retention", "<undef>") << ")"
      << std::endl;
    exit(1);
  }

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
```

```
        "canvas_infrastructure data_analysis_and_processing image io");
  // plugin: base_importer
  sdict.insert("index::app::plugins::base_importer::enabled", "true");

  // set network parameters for DiCE
  {
    config_network_parameters(index_connect, sdict);
  }

  info_cout(std::string("start IndeX via index_connect"), sdict);
  index_connect.initialize(sdict);
  check_success(index_connect.is_initialized());

  // setup
  App_rendering_context arc;
  check_success(setup_remote_host(index_connect, arc));

  // call remote rendering loop
  remote_rendering_loop(index_connect, arc, sdict);

  printf("Shutting down the NVIDIA IndeX library.\n");
  arc.shutdown();
  // index_connect will shutdown when out of scope

  printf("Terminating the NVIDIA IndeX remote service.\n");
  return 0;
}
```

9.5 configuration.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <sstream>
#include <string>
#include <stdio.h>
#include <iostream>

#include <mi/dice.h>
#include <mi/base/handle.h>
#include <mi/base/ilogger.h>

#include <nv/index/iindex.h>
#include <nv/index/iindex_debug_configuration.h>

#include <nv/index/app/string_dict.h>
#include <nv/index/app/index_connect.h>

class My_receiving_logger : public mi::base::Interface_implement<mi::base::ILogger>
{
public:
  // constructor
  My_receiving_logger()
    :
    m_is_show_destruct_message(true)
  {
    // empty
  }

  // constructor with is_show_destruct_message argument.
  // \param[in] is_show_destruct_message when true show a message when destruct
  My_receiving_logger(bool is_show_destruct_message) :
    m_is_show_destruct_message(is_show_destruct_message)
  {
    // empty
  }

  // destructor
  virtual ~My_receiving_logger()
  {
    if(m_is_show_destruct_message){
      fprintf(stdout, "info: My_receiving_logger is destructed.\n");
    }
  }

  // message out method
  virtual 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(stdout, "info: Log level = '%s', module:category = '%s', message = '%s'\n",
        log_level, module_category, message);
  }

public:
  // message switch when destructed.
  bool m_is_show_destruct_message;

private:
  // get log level char*.
  const char* get_log_level(mi::base::Message_severity level)
  {
    switch (level)
    {
    case mi::base::MESSAGE_SEVERITY_FATAL:
      return "My_receiving_logger: FATAL";
    case mi::base::MESSAGE_SEVERITY_ERROR:
      return "My_receiving_logger: ERROR";
    case mi::base::MESSAGE_SEVERITY_WARNING:
      return "My_receiving_logger: WARNING";
    case mi::base::MESSAGE_SEVERITY_INFO:
      return "My_receiving_logger: INFO";
    case mi::base::MESSAGE_SEVERITY_VERBOSE:
      return "My_receiving_logger: VERBOSE";
    case mi::base::MESSAGE_SEVERITY_DEBUG:
      return "My_receiving_logger: DEBUG";
    default:
      return "";
    }
  }
};

class Configuration:
  public nv::index::app::Index_connect
{
public:
  Configuration()
    :
    Index_connect(),
    m_is_unittest_mode(false)
  {
    // INFO_LOG << "DEBUG: Configuration() ctor";
  }

  virtual ~Configuration()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Configuration() dtor";
  }

  // launch application
  mi::Sint32 launch();

private:
```

```
  bool m_is_unittest_mode;

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& options) CPP11_OVERRIDE
  {
    const std::string com_name = options.get("command:", "<unknown_command>");
    m_is_unittest_mode = nv::index::app::get_bool(options.get("unittest", "false"));

    // print help and exit if -h
    if(options.is_defined("h"))
    {
      std::cout << "info: Usage: " << com_name << "  [option]\n"
           << "Option: [-h]\n"
           << "            printout this message\n"
           << "        [-unittest {0|1}\n"
           << "            when true, unit test mode.\n"
           << "        [-dice::verbose severity_level]\n"
           << "            verbose severity level (3 is info.). (default: "
           << options.get("dice::verbose")
           << ")"
           << std::endl;
      exit(1);
    }
    return true;
  }

  // Logger configuration
  bool initialize_logging(
    mi::neuraylib::ILogging_configuration* logging_configuration,
    nv::index::app::String_dict&           option) CPP11_OVERRIDE
  {
    assert(logging_configuration != 0);
   // This is for devsl=1 build to make severity level works for components and plugins.
    // Otherwise you will see 'PLUGIN init' messages.
    nv::index::app::Forwarding_logger::set_index_instance(get_index_interface());
    set_forwarding_logger_installed(true);

    mi::base::Handle<mi::base::ILogger> nvindex_forwarding_logger(
      get_index_interface()->get_forwarding_logger());

    // Log level configuration
    // Note: experimental logger is the test for dice logger
    // filter functionality. Currently this is not fully supported by IndeX.
   info_cout("current log level = " + nv::index::app::to_string(logging_configuration->get_log_level
    logging_configuration->set_log_level(mi::base::MESSAGE_SEVERITY_ERROR);
   info_cout("set log level to " +   nv::index::app::to_string(logging_configuration->get_log_level(
    nvindex_forwarding_logger->message(mi::base::MESSAGE_SEVERITY_INFO,
                      "APP:MAIN", "This info is filered out.\n");
    logging_configuration->set_log_level(mi::base::MESSAGE_SEVERITY_INFO);

    if(m_is_unittest_mode)
    {
      // unit test mode. Set log level warn
      logging_configuration->set_log_level(mi::base::MESSAGE_SEVERITY_WARNING);
    }
    else
    {
```

```
    info_cout("set log level to " +   nv::index::app::to_string(logging_configuration->get_log_level
      nvindex_forwarding_logger->message(mi::base::MESSAGE_SEVERITY_INFO,
                        "APP:MAIN", "This should be seen.\n");
    }

    // Create and set my logger
   mi::base::Handle<mi::base::ILogger> logger(new My_receiving_logger(!m_is_unittest_mode));
    logging_configuration->set_receiving_logger(logger.get());

    return true;
  }

  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

  virtual bool initialize_index_debug_configuration(
    nv::index::IIndex_debug_configuration* index_debug_configuration,
    nv::index::app::String_dict&           options)
  {
    check_success(index_debug_configuration != 0);
    index_debug_configuration->set_option("disable_sse=1");
    info_cout("IIndex_debug_configuration: disabled sse.", options);

    return true;
  }
};

mi::Sint32 Configuration::launch()
{
  nv::index::app::String_dict option = get_options();
 info_cout(std::string("NVIDIA IndeX version: ") + get_index_interface()->get_version(), option);

  // Remove the user defined receiving logger at runtime
  mi::base::Handle<mi::neuraylib::ILogging_configuration> logging_configuration(
   get_index_interface()->get_api_component<mi::neuraylib::ILogging_configuration>());
  check_success(logging_configuration.is_valid_interface());
  logging_configuration->set_receiving_logger(0);
  info_cout("The receiving logger has been reset.", option);

  return 0;
}
```

```
int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("unittest",      "0");      // default mode
  sdict.insert("dice::verbose", "3");

  // Initialize configure app
  Configuration configuration;
  configuration.initialize(argc, argv, sdict);
  check_success(configuration.is_initialized());

  // Launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = configuration.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.6 create_annotations.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/ifont.h>
#include <nv/index/iindex.h>
#include <nv/index/ilabel.h>
#include <nv/index/ilight.h>
#include <nv/index/iline_set.h>
#include <nv/index/imaterial.h>
#include <nv/index/ipoint_set.h>
#include <nv/index/iscene.h>

#include <mi/math.h>

#include <nv/index/app/string_dict.h>
#include <nv/index/app/index_connect.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Create_annotations:
  public nv::index::app::Index_connect
{
public:
  Create_annotations()
    :
    Index_connect(),
    m_is_unittest(false)
  {
    // INFO_LOG << "DEBUG: create_annotations_index_connect() ctor";
  }

  virtual ~Create_annotations()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~create_annotations_index_connect() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override for unittest support
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
```

```
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Create point set in the scene.
  //
  // \param[in] scene_edit      IScene interface
  // \param[in] dice_transaction dice transaction
  // \return true when success
  mi::neuraylib::Tag create_scene(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction);

  // set a group node's transformation matrix
  //
  // \param[in] group_node_tag group node tag to be set
  // \param[in] frame_id   a frame id. The translation is depends on this id
  // \param[in] dice_transaction dice transaction
  void set_transformation(
    const mi::neuraylib::Tag&         group_node_tag,
    mi::Uint32                        frame_id,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  // setup camera to see this example scene
  //
  // \param[in] cam  a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // render a frame
  //
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(
    const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                 m_verify_image_fname_sequence;
```

```
  std::string                                                    m_font_fpath;
};

mi::Sint32 Create_annotations::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    m_cluster_configuration =
      get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    // This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));

    mi::neuraylib::Tag scene_group_tag = mi::neuraylib::NULL_TAG;
    mi::neuraylib::Tag camera_tag      = mi::neuraylib::NULL_TAG;

    {
      // DiCE database access
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        // Setup session information
        m_session_tag = m_index_session->create_session(dice_transaction.get());
        check_success(m_session_tag.is_valid());
        mi::base::Handle<const nv::index::ISession> session(
          dice_transaction->access<const nv::index::ISession>(m_session_tag));
        check_success(session.is_valid_interface());

        mi::base::Handle< nv::index::IScene > scene_edit(
          dice_transaction->edit<nv::index::IScene>(session->get_scene()));
        check_success(scene_edit.is_valid_interface());

        //------------------------------------------------------------------------
        // Scene setup: add annotation shape, scene parameters, camera.
        //------------------------------------------------------------------------
        // Add an hierarchical scene description node to the scene
        scene_group_tag = create_scene(scene_edit.get(), dice_transaction.get());
        check_success(scene_group_tag.is_valid());

        // Create and edit a camera. Data distribution is
        // based on the camera. (Because only visible massive
        // data are considered)
        mi::base::Handle< nv::index::IPerspective_camera > cam(
          scene_edit->create_camera<nv::index::IPerspective_camera>());
        check_success(cam.is_valid_interface());
        setup_camera(cam.get());
        camera_tag = dice_transaction->store(cam.get());
        check_success(camera_tag.is_valid());
```

```
    const mi::math::Vector<mi::Uint32,2> buffer_resolution(512, 512);
    m_image_file_canvas->set_resolution(buffer_resolution);

    // Set up the scene
    const mi::math::Bbox_struct< mi::Float32, 3 > xyz_roi_st = {
      {   0.0f,   0.0f,   0.0f, },
      { 500.0f, 500.0f, 500.0f, },
    };

    // set the region of interest
    const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
    check_success(xyz_roi.is_volume());
    scene_edit->set_clipped_bounding_box(xyz_roi_st);

    // Set the scene global transformation matrix.
    // only change the coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
      0.0f,  0.0f,  0.0f,  1.0f
      );
    scene_edit->set_transform_matrix(transform_mat);

    // Set the current camera to the scene.
    check_success(camera_tag.is_valid());
    scene_edit->set_camera(camera_tag);
  }
  dice_transaction->commit();
}

// Rendering
{
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    const std::string fname = get_output_file_name(m_outfname, 0);
    std::stringstream verify_image_fname;
    if(!m_verify_image_fname_sequence.empty())
    {
      verify_image_fname << m_verify_image_fname_sequence << "000.ppm";
    }
 mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));

 const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;

      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
```

```
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
             << os.str();
      }

      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
             m_image_file_canvas.get(), verify_image_fname.str(), get_options())))
      {
        exit_code = 1;
      }
    }
    dice_transaction->commit();
  }

  for(mi::Uint32 i = 1; i < 20; ++i)
  {
    {
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        set_transformation(scene_group_tag, i, dice_transaction.get());
      }
      dice_transaction->commit();
    }

    const std::string fname = get_output_file_name(m_outfname, i);

    std::stringstream verify_image_fname;
    if(!m_verify_image_fname_sequence.empty())
    {
      const size_t BUFSZ = 64;
      char buf[BUFSZ];
      snprintf(buf, BUFSZ - 1, "%03d.ppm", i);
      verify_image_fname << m_verify_image_fname_sequence << buf;
    }

   mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
  const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;
      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }

    ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
            << os.str();
      exit_code = 1;
    }
```

```
      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
               m_image_file_canvas.get(), verify_image_fname.str(), get_options())))
      {
        exit_code = 1;
      }
    }

  }

  return exit_code;
}


bool Create_annotations::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname                    = sdict.get("outfname", "");
  m_verify_image_fname_sequence = sdict.get("verify_image_fname_sequence", "");
  m_font_fpath                  = sdict.get("font_fpath");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["                        + m_outfname +
       "], verify_image_fname_sequence = [" + m_verify_image_fname_sequence +
       "], dice::verbose = "                + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"

      << "        [-dice::verbose severity_level]\n"
      << "            verbose severity level (3 is info.). (default: "
      <<            sdict.get("dice::verbose") << ")\n"

      << "        [-font_fpath FONT_FILE_PATH]\n"
      << "            font file path. (default: " << m_font_fpath << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
```

```
      << "        [-verify_image_fname_sequence [image_fname]]\n"
      << "           when image_fname exist, verify the rendering image. (default: ["
      << sdict.get("verify_image_fname_sequence") << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << sdict.get("unittest") << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

mi::neuraylib::Tag Create_annotations::create_scene(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Set up the transformation matrix
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  // Add a translation
  transform_mat.translate(mi::math::Vector<mi::Float32, 3>(5.0f, 0.0f, 0.0f));
  group_node->set_transform(transform_mat);

  // Add a light and a material
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
    headlight->set_intensity(color_intensity);
    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());
    group_node->append(headlight_tag, dice_transaction);

    // add material for 3D points shape (the material is only effective for 3D shape)
   mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(mi::math::Color(0.3f, 0.3f, 0.3f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    phong_1->set_specular(mi::math::Color(0.4f));
    phong_1->set_shininess(100.f);
   const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
    check_success(phong_1_tag.is_valid());
    group_node->append(phong_1_tag, dice_transaction);
  }
```

```
 // Add font to the scene description
mi::base::Handle<nv::index::IFont> font(scene_edit->create_attribute<nv::index::IFont>());
 check_success(font.is_valid_interface());

 if(font->set_file_name(m_font_fpath.c_str()))
 {
   INFO_LOG << "set the font path [" << m_font_fpath << "]";
 }
 else
 {
   ERROR_LOG << "Can not find the font path [" << m_font_fpath << "], "
        << "the rendering result may not correct.";
 }
 font->set_font_resolution(64.0f);
const mi::neuraylib::Tag font_tag = dice_transaction->store_for_reference_counting(font.get());
 check_success(font_tag.is_valid());
 group_node->append(font_tag, dice_transaction);
 // Add a label layout to the scene description
mi::base::Handle<nv::index::ILabel_layout> label_layout(scene_edit->create_attribute<nv::index::I
 check_success(label_layout.is_valid_interface());
 const mi::Float32 padding = 15.0f;
 label_layout->set_padding(padding);
mi::math::Color_struct foreground; foreground.r = 0.9f; foreground.g = 0.8f; foreground.b = 0.8f; fo
mi::math::Color_struct background; background.r = 0.4f; background.g = 0.5f; background.b = 0.4f; ba
 label_layout->set_color(foreground, background);
const mi::neuraylib::Tag label_layout_tag = dice_transaction->store_for_reference_counting(label_l
 check_success(label_layout_tag.is_valid());
 group_node->append(label_layout_tag, dice_transaction);

 // Some constants used to setup the scene (lines, points, labels) later...
 const mi::Float32 x_min = 0.f;
 const mi::Float32 x_max = 400.f;
 const mi::Float32 y_min = 0.f;
 const mi::Float32 y_max = 400.f;

 {
   // Point coordinates and its attributes. According to the
   // attributes, color and radius are defined.
   std::vector< mi::math::Vector_struct< mi::Float32, 3> > point_pos_vec;
   std::vector< mi::math::Color_struct>                    per_point_color;
   std::vector< mi::Float32 >                              per_point_radius;

   std::vector< mi::math::Vector_struct< mi::Float32, 3> > segment_vertices;
   std::vector< mi::math::Color_struct>                    color_per_segment;
   std::vector< mi::Float32 >                              width_per_segment;
  mi::math::Vector_struct< mi::Float32, 3> v0; v0.x = x_min; v0.y = y_min; v0.z = 30.f;
  mi::math::Vector_struct< mi::Float32, 3> v1; v1.x = x_max; v1.y = y_min; v1.z = 30.f;
  mi::math::Vector_struct< mi::Float32, 3> v2; v2.x = x_max; v2.y = y_max; v2.z = 30.f;
  mi::math::Vector_struct< mi::Float32, 3> v3; v3.x = x_min; v3.y = y_max; v3.z = 30.f;

   segment_vertices.push_back(v0); segment_vertices.push_back(v1);
   segment_vertices.push_back(v1); segment_vertices.push_back(v2);
   segment_vertices.push_back(v2); segment_vertices.push_back(v3);
   segment_vertices.push_back(v3); segment_vertices.push_back(v0);
   point_pos_vec.push_back(v0);
   point_pos_vec.push_back(v1);
   point_pos_vec.push_back(v2);
```

```
  point_pos_vec.push_back(v3);

  mi::math::Color_struct color;
  color.r = 0.8f; color.g = 0.8f; color.b = 0.8f; color.a = 1.f;
  color_per_segment.push_back(color);
  color_per_segment.push_back(color);
  color_per_segment.push_back(color);
  color_per_segment.push_back(color);
  per_point_color.push_back(color); per_point_color.push_back(color);
  per_point_color.push_back(color); per_point_color.push_back(color);

  width_per_segment.push_back(2.f);
  width_per_segment.push_back(2.f);
  width_per_segment.push_back(2.f);
  width_per_segment.push_back(2.f);
  per_point_radius.push_back(3.f); per_point_radius.push_back(3.f);
  per_point_radius.push_back(3.f); per_point_radius.push_back(3.f);

  // Create line set using the scene's factory
mi::base::Handle<nv::index::ILine_set> line_set(scene_edit->create_shape<nv::index::ILine_set>(
  check_success(line_set.is_valid_interface());
 // ... set the line style such a dashed or dotted or solid linestyle (see ILine_set)
  line_set->set_line_style(nv::index::ILine_set::LINE_STYLE_DOTTED);
 // ... set the line type. Currently only line segments are supported (see ILine_set)
  line_set->set_line_type(nv::index::ILine_set::LINE_TYPE_SEGMENTS);
  // ... and the lines with colors and widths all in line segment order.
  line_set->set_lines(&segment_vertices[0], segment_vertices.size());
  line_set->set_colors(&color_per_segment[0], color_per_segment.size());
  line_set->set_widths(&width_per_segment[0], width_per_segment.size());

  // Add the points to the database
 const mi::neuraylib::Tag line_set_tag = dice_transaction->store_for_reference_counting(line_set.
  // if you are not registered Line_set, the next check fails.
  check_success(line_set_tag.is_valid());
  group_node->append(line_set_tag, dice_transaction);

  // Create point set using the scene's factory.
mi::base::Handle<nv::index::IPoint_set> point_set(scene_edit->create_shape<nv::index::IPoint_se
  check_success(point_set.is_valid_interface());
  // ... set the point styles (see IPoint_set)
  // ... and the vertices with colors and radii.
  point_set->set_vertices(&point_pos_vec[0], point_pos_vec.size());
  point_set->set_colors(&per_point_color[0], per_point_color.size());
  point_set->set_radii(&per_point_radius[0], per_point_radius.size());

  // Storing the point set in the database.
 const mi::neuraylib::Tag point_set_scene_element_tag = dice_transaction->store_for_reference_cou
  check_success(point_set_scene_element_tag.is_valid());
  group_node->append(point_set_scene_element_tag, dice_transaction);
}

{
  mi::math::Color_struct color;
  color.r = 0.8f; color.g = 0.8f; color.b = 0.9f; color.a = 1.f;
  const mi::Float32 x_spacing = 25.f;
  const mi::Float32 y_height = 10.f;
  std::vector< mi::math::Vector_struct< mi::Float32, 3> > segment_vertices;
```

```
  std::vector< mi::math::Color_struct>                    color_per_segment;
  std::vector< mi::Float32 >                              width_per_segment;
mi::math::Vector_struct< mi::Float32, 3> v0; v0.x = x_min; v0.y = y_min; v0.z = 30.f;
mi::math::Vector_struct< mi::Float32, 3> v1; v1.x = x_max; v1.y = y_min; v1.z = 30.f;

  for(mi::Float32 i=x_min; i<=x_max; i+=x_spacing)
  {
    v0.x = i; v0.y = y_max; v0.z = 30.f;
    v1.x = i; v1.y = y_max+y_height; v1.z = 30.f;
    if(mi::math::fmod(i,100.f) == 0) v1.y+=y_height;

    segment_vertices.push_back(v0); segment_vertices.push_back(v1);
    color_per_segment.push_back(color);
    width_per_segment.push_back(2.f);

    if(mi::math::fmod(i,100.f) == 0.f)
    {
   mi::base::Handle<nv::index::ILabel_2D> label(scene_edit->create_shape<nv::index::ILabel_2D>(
      check_success(label.is_valid_interface());
      std::stringstream sstr;
      sstr << i;
      label->set_text(sstr.str().c_str());
      const mi::Float32 height = 40.f;
      const mi::Float32 width = 40.f;
   mi::math::Vector_struct<mi::Float32, 3>  position; position.x = v1.x-(width/2);  position.y = v
     mi::math::Vector_struct<mi::Float32, 2>  right; right.x = 1.f;  right.y = 0.f;
      mi::math::Vector_struct<mi::Float32, 2>  up; up.x = 0.f;  up.y = 1.f;
      // To test 'compute_label_width' create a label with width = -1 (auto width)
   // and then call label->compute_label_width() with the current font and label layout
   // the function will return the calculated label frame width for perfect fitting of the text
      // taking into account the frame-height, font, text and padding.
      label->set_geometry(position, right, up, height, width);//-1);
   // INFO_LOG << "Label :" << sstr.str() << ", width = " << label->compute_label_width(font.get(),
   const mi::neuraylib::Tag label_tag = dice_transaction->store_for_reference_counting(label.get
      check_success(label_tag.is_valid());
      group_node->append(label_tag, dice_transaction);
    }
  }

  // Create line set using the scene's factory
mi::base::Handle<nv::index::ILine_set> line_set(scene_edit->create_shape<nv::index::ILine_set>(
  check_success(line_set.is_valid_interface());
 // ... set the line style such a dashed or dotted or solid linestyle (see ILine_set)
  line_set->set_line_style(nv::index::ILine_set::LINE_STYLE_SOLID);
 // ... set the line type. Currently only line segments are supported (see ILine_set)
  line_set->set_line_type(nv::index::ILine_set::LINE_TYPE_SEGMENTS);
  // ... and the lines with colors and widths all in line segment order.
  line_set->set_lines(&segment_vertices[0], segment_vertices.size());
  line_set->set_colors(&color_per_segment[0], color_per_segment.size());
  line_set->set_widths(&width_per_segment[0], width_per_segment.size());

 const mi::neuraylib::Tag line_set_tag = dice_transaction->store_for_reference_counting(line_set.
  // if you are not registered Line_set, the next check fails.
  check_success(line_set_tag.is_valid());
  group_node->append(line_set_tag, dice_transaction);
}
```

```
  {
   mi::base::Handle<nv::index::ILabel_layout> label_layout(scene_edit->create_attribute<nv::index:
    check_success(label_layout.is_valid_interface());
    const mi::Float32 padding = 10.f;
    label_layout->set_padding(padding);
   mi::math::Color_struct foreground; foreground.r = 0.9f; foreground.g = 0.95f; foreground.b = 0.99f
   mi::math::Color_struct background; background.r = 0.4f; background.g = 0.5f; background.b = 0.4f; b
    label_layout->set_color(foreground, background);
   const mi::neuraylib::Tag label_layout_tag = dice_transaction->store_for_reference_counting(label
    check_success(label_layout_tag.is_valid());
    group_node->append(label_layout_tag, dice_transaction);

   mi::base::Handle<nv::index::ILabel_3D> label(scene_edit->create_shape<nv::index::ILabel_3D>());
    check_success(label.is_valid_interface());
    label->set_text("slice...");
   mi::math::Vector_struct<mi::Float32, 3> position; position.x = x_max; position.y = y_max; positi
   mi::math::Vector_struct<mi::Float32, 3> right; right.x = 0.f; right.y = -1.f; right.z = 0.f;
    mi::math::Vector_struct<mi::Float32, 3> up; up.x = 1.f; up.y = 0.f; up.z = 0.f;
    const mi::Float32 height = 40.f;
    const mi::Float32 width = 120.f;
    label->set_geometry(position, right, up, height, width);
   const mi::neuraylib::Tag label_tag = dice_transaction->store_for_reference_counting(label.get())
    check_success(label_tag.is_valid());
    group_node->append(label_tag, dice_transaction);
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());

  // Define the root and attributes that are active globally
 mi::base::Handle<nv::index::IDirectional_light> light_global(scene_edit->create_attribute<nv::ind
  check_success(light_global.is_valid_interface());
 const mi::neuraylib::Tag light_global_tag = dice_transaction->store_for_reference_counting(light_g
  check_success(light_global_tag.is_valid());
  scene_edit->append(light_global_tag, dice_transaction);

  scene_edit->append(group_node_tag, dice_transaction);

  INFO_LOG << "Hierarchical scene description creation complete.";
  return group_node_tag;
}

void Create_annotations::set_transformation(
  const mi::neuraylib::Tag&         group_node_tag,
  mi::Uint32                        frame_id,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  mi::base::Handle< nv::index::ITransformed_scene_group > group_node(
    dice_transaction->edit<nv::index::ITransformed_scene_group>(group_node_tag));
  check_success(group_node.is_valid_interface());

  // Set up the transformation matrix
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  // Add a translation
  mi::Float32 z = static_cast<mi::Float32>(frame_id) * 10.0f;
  transform_mat.translate(mi::math::Vector<mi::Float32, 3>(5.0f, 0.0f, z));
  group_node->set_transform(transform_mat);
```

```
}

void Create_annotations::setup_camera(
  nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from( 100.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_annotations::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("dice::network::mode", "OFF");                  // network mode
  sdict.insert("font_fpath", "/usr/share/fonts/dejavu/DejaVuSans.ttf"); // font path
```

```
  sdict.insert("outfname", "frame_create_annotations");      // output file base name
  sdict.insert("verify_image_fname_sequence", "");            // for unit test
  sdict.insert("unittest", "0");                              // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "yes");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "no");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io");

  // Initialize application
  Create_annotations create_annotations;
  create_annotations.initialize(argc, argv, sdict);
  check_success(create_annotations.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_annotations.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.7 create_attribute_line_set.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iindex.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include "attribute_line_set.h"

#include <nv/index/app/string_dict.h>
#include <nv/index/app/forwarding_logger.h>
#include <nv/index/app/index_connect.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Create_attribute_line_set:
    public nv::index::app::Index_connect
{
public:
  Create_attribute_line_set()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_attribute_line_set() ctor";
  }

  virtual ~Create_attribute_line_set()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_attribute_line_set() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
```

```
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

  // override
  virtual bool register_serializable_classes(
    mi::neuraylib::IDice_configuration* configuration_interface,
    nv::index::app::String_dict&        options) CPP11_OVERRIDE
  {
    bool is_registered = false;
   is_registered = get_index_interface()->register_serializable_class<Attribute_line_set>();
    check_success(is_registered);
    return is_registered;
  }

private:
  // create line set in the scene.
  //
  // \param[in] scene_edit       IScene for editing.
  // \param[in] dice_transaction dice transaction
  // \return true when success
  bool create_attribute_line_set(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction);

  // setup camera to see this example scene
  //
  // \param[in] cam              a camera to be set
  void setup_camera(
    nv::index::IPerspective_camera* cam) const;

  // render a frame
  //
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(
    const std::string& output_fname) const;


  // This session tag
  mi::neuraylib::Tag                                                  m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                    m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                            m_outfname;
  bool                                                                m_is_unittest;
  std::string                                                      m_verify_image_fname;

};
```

```
mi::Sint32 Create_attribute_line_set::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    m_cluster_configuration =
      get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    // This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));

    {
      // DiCE database access
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        // Setup session information
        m_session_tag =
          m_index_session->create_session(dice_transaction.get());
        check_success(m_session_tag.is_valid());
        mi::base::Handle< nv::index::ISession const > session(
          dice_transaction->access< nv::index::ISession const >(
            m_session_tag));
        check_success(session.is_valid_interface());

        mi::base::Handle< nv::index::IScene > scene_edit(
          dice_transaction->edit< nv::index::IScene >(session->get_scene()));
        check_success(scene_edit.is_valid_interface());

        //------------------------------------------------------------------------
        // Scene setup: add line set shape, scene parameters, camera.
        //------------------------------------------------------------------------
        // Add line set shape to the scene
     check_success(create_attribute_line_set(scene_edit.get(), dice_transaction.get()));

        // Create and edit a camera. Data distribution is based on
        // the camera. (Because only visible massive data are
        // considered)
        mi::base::Handle< nv::index::IPerspective_camera > cam(
          scene_edit->create_camera<nv::index::IPerspective_camera>());
        check_success(cam.is_valid_interface());
        setup_camera(cam.get());

        const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
        check_success(camera_tag.is_valid());

        const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
        m_image_file_canvas->set_resolution(buffer_resolution);
```

```
    // Set up the scene
    mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
      {   0.0f,   0.0f,   0.0f, },
      { 500.0f, 500.0f, 500.0f, },
    };

    // set the region of interest
    const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
    check_success(xyz_roi.is_volume());
    scene_edit->set_clipped_bounding_box(xyz_roi_st);

    // Set the scene global transformation matrix.
    // only change the coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
      0.0f,  0.0f,  0.0f,  1.0f
      );
    scene_edit->set_transform_matrix(transform_mat);

    // Set the current camera to the scene.
    check_success(camera_tag.is_valid());
    scene_edit->set_camera(camera_tag);
  }
  // Finish the setup transaction
  dice_transaction->commit();
}

// Rendering
{
  mi::Sint32  const frame_idx = 0;
  std::string const fname = get_output_file_name(m_outfname, frame_idx);
 mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));

const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
  if (err_set->any_errors())
  {
    std::ostringstream os;
    const mi::Uint32 nb_err = err_set->get_nb_errors();
    for (mi::Uint32 e = 0; e < nb_err; ++e)
    {
      if (e != 0) os << '\n';
      const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
      os << err->get_error_string();
    }

 ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
        << os.str();
    exit_code = 1;
  }

  // verify the generated frame
  if (!(verify_canvas_result(get_application_layer_interface(),
              m_image_file_canvas.get(), m_verify_image_fname, get_options())))
  {
    exit_code = 1;
```

```
      }
    }
  }
  // index_connect shutdown

  return exit_code;
}

bool Create_attribute_line_set::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname  = sdict.get("outfname");
  m_verify_image_fname = sdict.get("verify_image_fname");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"
      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"
      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")"
      << std::endl;
    exit(1);
  }
  return true;
}
```

```
bool Create_attribute_line_set::create_attribute_line_set(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  // hierarchical scene description node for the point set
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Line coordinates and its attributes. According to the
  // attributes, color and width are defined.
  // Here, we create two line sets.
  std::vector< mi::math::Vector_struct< mi::Float32, 3> > line_seg_vec[2];
  std::vector< mi::Float32 >                              attribute_vec[2];

  // Square shaped positions
  const mi::Sint32  column_count       = 20;
  const mi::Sint32  line_segment_count = 201;
  const mi::Sint32  line_set_count     = 2;
  const mi::Float32 x_org              =  0.0;
  const mi::Float32 y_org              =  0.0;
  const mi::Float32 x_mag              = 25.0;
  const mi::Float32 y_mag              = 25.0;
  const mi::Float32 z                  = 30.0;
  const mi::Float32 y_offset[2]        = { 0.0f, 280.0f };
  for(mi::Sint32 i = 0; i < line_segment_count; ++i)
  {
    mi::math::Vector_struct< mi::Float32, 3> pos[2];
    for(mi::Sint32 j = 0; j < line_set_count; ++j)
    {
      pos[j].x = x_org + static_cast< mi::Float32 >(i % column_count) * x_mag;
    pos[j].y = y_org + static_cast< mi::Float32 >(i / column_count) * y_mag + y_offset[j];
      pos[j].z = z;
      line_seg_vec[j]. push_back(pos[j]);
      attribute_vec[j].push_back(static_cast< mi::Float32 >(i));
    }
  }

  // Create two line sets
  for(mi::Sint32 j = 0; j < 2; ++j)
  {
    // Create attribute_line_set scene element and add it to the scene
    mi::base::Handle< Attribute_line_set > line_set(
      new Attribute_line_set(line_seg_vec[j], attribute_vec[j]));

    // Add the line segments to the database
    mi::neuraylib::Tag const line_set_scene_element_tag =
      dice_transaction->store_for_reference_counting(line_set.get());
    // if you are not registered Attribute_line_set, the next check fails.
    check_success(line_set_scene_element_tag.is_valid());
    // Add to the scene description
    group_node->append(line_set_scene_element_tag, dice_transaction);
   INFO_LOG << "Added line_set (size: " << line_set_count << ") to the scene (tag id: "
       << line_set_scene_element_tag.id << ").";
```

```
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  return true;
}

void Create_attribute_line_set::setup_camera(
  nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from( 254.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_attribute_line_set::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}
```

```
int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                       // log level
  sdict.insert("dice::network::mode", "OFF");               // network mode
 sdict.insert("outfname", "frame_create_attribute_line_set");     // output file base name
  sdict.insert("verify_image_fname", "");                   // for unit test
  sdict.insert("unittest", "0");                            // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                 // default: not call from make check.

  // Load IndeX library via Index_connect
  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // Initialize application
  Create_attribute_line_set create_attribute_line_set;
  create_attribute_line_set.initialize(argc, argv, sdict);
  check_success(create_attribute_line_set.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_attribute_line_set.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.8 create_attribute_point_set.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include "attribute_point_set.h"

#include <nv/index/app/forwarding_logger.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/index_connect.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Create_attribute_point_set:
  public nv::index::app::Index_connect
{
public:
  Create_attribute_point_set()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_attribute_point_set() ctor";
  }

  virtual ~Create_attribute_point_set()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_attribute_point_set() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);
```

```
    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

  // override
  virtual bool register_serializable_classes(
    mi::neuraylib::IDice_configuration* configuration_interface,
    nv::index::app::String_dict&        options) CPP11_OVERRIDE
  {
    bool is_registered = false;
   is_registered = get_index_interface()->register_serializable_class<Attribute_point_set>();
    check_success(is_registered);
    return is_registered;
  }

private:
  // create attribute point set in the scene.
  // \param[in] scene_edit       IScene for editing.
  // \param[in] dice_transaction dice transaction
  // \return true when success
  bool create_attribute_point_set(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  // setup camera to see this example scene
  // \param[in] cam              a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // render a frame
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;


  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                        m_verify_image_fname;
};

mi::Sint32 Create_attribute_point_set::launch()
{
  mi::Sint32 exit_code = 0;
```

```
// Get DiCE database components
{
 m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configura
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
 m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));
  {
    // DiCE database access
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());
    {
      // Setup session information
      m_session_tag = m_index_session->create_session(dice_transaction.get());
      check_success(m_session_tag.is_valid());
      mi::base::Handle< nv::index::ISession const > session(
        dice_transaction->access< nv::index::ISession const >(
          m_session_tag));
      check_success(session.is_valid_interface());

      mi::base::Handle< nv::index::IScene > scene_edit(
        dice_transaction->edit< nv::index::IScene >(session->get_scene()));
      check_success(scene_edit.is_valid_interface());

      //-------------------------------------------------------------------------
      // Scene setup: add point set shape, scene parameters, camera.
      //-------------------------------------------------------------------------
      // Add an point set shape to the scene
   check_success(create_attribute_point_set(scene_edit.get(), dice_transaction.get()));

      // Create and edit a camera. Data distribution is based on
      // the camera. (Because only visible massive data are
      // considered)
      mi::base::Handle< nv::index::IPerspective_camera > cam(
        scene_edit->create_camera<nv::index::IPerspective_camera>());
      check_success(cam.is_valid_interface());
      setup_camera(cam.get());
      const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
      check_success(camera_tag.is_valid());

      const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
      m_image_file_canvas->set_resolution(buffer_resolution);

      // Set up the scene
      mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
        {   0.0f,   0.0f,   0.0f, },
        { 500.0f, 500.0f, 500.0f, },
      };

      // set the region of interest
```

```
        const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
        check_success(xyz_roi.is_volume());
        scene_edit->set_clipped_bounding_box(xyz_roi_st);

        // Set the scene global transformation matrix.
        // only change the coordinate system
        mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
          1.0f,  0.0f,  0.0f,  0.0f,
          0.0f,  1.0f,  0.0f,  0.0f,
          0.0f,  0.0f, -1.0f,  0.0f,
          0.0f,  0.0f,  0.0f,  1.0f
          );
        scene_edit->set_transform_matrix(transform_mat);
        // Set the current camera to the scene.
        check_success(camera_tag.is_valid());
        scene_edit->set_camera(camera_tag);
      }
      // Finished scene setup
      dice_transaction->commit();
    }

    // Rendering
    {
      // Render a frame and save the rendered image to a file.
      const mi::Sint32  frame_idx = 0;
      const std::string fname = get_output_file_name(m_outfname, frame_idx);
     mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      if (err_set->any_errors())
      {
        std::ostringstream os;
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        exit_code = 1;
      }

      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), m_verify_image_fname, get_options())))
      {
        exit_code = 1;
      }
    }
  }

  return exit_code;
}

bool Create_attribute_point_set::evaluate_options(nv::index::app::String_dict& sdict)
```

```
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  // Set own options
  m_outfname           = sdict.get("outfname", "");
  m_verify_image_fname = sdict.get("verify_image_fname", "");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"
      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"
      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << sdict.get("unittest") << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

bool Create_attribute_point_set::create_attribute_point_set(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);
```

```
// hierarchical scene description node for the point set
mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
  scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
check_success(group_node.is_valid_interface());

// Add a light and a material
{
  // Add a light
  mi::base::Handle<nv::index::IDirectional_headlight> headlight(
    scene_edit->create_attribute<nv::index::IDirectional_headlight>());
  check_success(headlight.is_valid_interface());
  const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
  headlight->set_intensity(color_intensity);
  headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
 const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
  check_success(headlight_tag.is_valid());
  group_node->append(headlight_tag, dice_transaction);

  // add material for 3D points shape (the material is only effective for 3D shape)
 mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
  check_success(phong_1.is_valid_interface());
  phong_1->set_ambient(mi::math::Color(0.3f, 0.3f, 0.3f, 1.0f));
  phong_1->set_diffuse(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
  phong_1->set_specular(mi::math::Color(0.4f));
  phong_1->set_shininess(100.f);
 const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
  check_success(phong_1_tag.is_valid());
  group_node->append(phong_1_tag, dice_transaction);
}

// Point coordinates and its attributes. According to the
// attributes, color and radius are defined.
// Here, we create two point sets.
std::vector< mi::math::Vector_struct< mi::Float32, 3> > point_pos_vec[2];
std::vector< mi::Float32 >                              attribute_vec[2];

// square shaped positions
const mi::Sint32  column_count= 20;
const mi::Sint32  point_count = 200;
const mi::Float32 x_mag = 25.0;
const mi::Float32 y_mag = 25.0;
const mi::Float32 z     = 30.0;
const mi::Float32 y_offset[2] = { 0.0f, 280.0f };
for(mi::Sint32 i = 0; i < point_count; ++i)
{
  mi::math::Vector_struct< mi::Float32, 3> pos[2];
  for(mi::Sint32 j = 0; j < 2; ++j)
  {
    pos[j].x = static_cast< mi::Float32 >(i % column_count) * x_mag;
    pos[j].y = static_cast< mi::Float32 >(i / column_count) * y_mag + y_offset[j];
    pos[j].z = z;
    point_pos_vec[j].push_back(pos[j]);
    attribute_vec[j].push_back(static_cast< mi::Float32 >(i));
  }
}

// Create two point sets
```

```
  for(mi::Sint32 j = 0; j < 2; ++j)
  {

    // Create attribute_point_set scene element and add it to the scene
    nv::index::IPoint_set::Point_style style =
    (j==0) ? nv::index::IPoint_set::FLAT_CIRCLE : nv::index::IPoint_set::SHADED_CIRCLE;
    mi::base::Handle< Attribute_point_set > point_set(
      new Attribute_point_set(point_pos_vec[j], attribute_vec[j], style));
    // Add the points to the database
   const mi::neuraylib::Tag point_set_scene_element_tag = dice_transaction->store_for_reference_cou
    // if you are not registered Attribute_point_set, the next check fails.
    check_success(point_set_scene_element_tag.is_valid());
    // Add to the scene description
    group_node->append(point_set_scene_element_tag, dice_transaction);
    std::stringstream sstr;
    sstr << "Added point_set (size: " << point_count << ") to the scene (tag id: "
      << point_set_scene_element_tag.id << ").";
    INFO_LOG << sstr.str();
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  return true;
}

void Create_attribute_point_set::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from( 254.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_attribute_point_set::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
```

```
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("dice::network::mode", "OFF");                  // network mode
 sdict.insert("outfname", "frame_create_attribute_point_set"); // output file base name
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1");   // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");               // default: not call from make check.

  // Load IndeX library via Index_connect
  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // Initialize application
  Create_attribute_point_set create_attribute_point_set;
  create_attribute_point_set.initialize(argc, argv, sdict);
  check_success(create_attribute_point_set.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_attribute_point_set.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.9 create_circles_and_ellipses.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/iindex.h>
#include <nv/index/isession.h>
#include <nv/index/iscene.h>
#include <nv/index/icamera.h>
#include <nv/index/icircle.h>
#include <nv/index/iellipse.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Create_circles_and_ellipses:
  public nv::index::app::Index_connect
{
public:
  Create_circles_and_ellipses()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_circles_and_ellipses() ctor";
  }

  virtual ~Create_circles_and_ellipses()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_circles_and_ellipses() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
```

```
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Create circles and ellipses in the scene.
  //
  // \param[in] scene_edit       IScene for the scene edit.
  // \param[in] dice_transaction dice transaction
  // \return true when success
  bool create_circles_and_ellipses(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  // setup camera to see this example scene
  //
  // \param[in] camera a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // render a frame
  //
  // \param[in] index_connect access to the NVIDIA IndeX library
  // \param[in] arc              application rendering context
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                        m_verify_image_fname;
};

mi::Sint32 Create_circles_and_ellipses::launch()
{
  mi::Sint32 exit_code = 0;

  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
```

```
// This may fail when there is a license problem.
check_success(is_local_host_joined(m_cluster_configuration.get()));

{
  // DiCE database access
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    // Setup session information
    m_session_tag =
      m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle< nv::index::ISession const > session(
      dice_transaction->access< nv::index::ISession const >(m_session_tag));
    check_success(session.is_valid_interface());

    mi::base::Handle< nv::index::IScene > scene_edit(
      dice_transaction->edit<nv::index::IScene>(session->get_scene()));
    check_success(scene_edit.is_valid_interface());

    //-------------------------------------------------------------------------
   // Scene setup: create circles and ellipses in the scene, scene parameters, camera.
    //-------------------------------------------------------------------------
    // Add circles and ellipses shape to the scene
  check_success(create_circles_and_ellipses(scene_edit.get(), dice_transaction.get()));

    // Create and edit a camera. Data distribution is based on
    // the camera. (Because only visible massive data are
    // considered)
    mi::base::Handle< nv::index::IPerspective_camera > cam(
      scene_edit->create_camera<nv::index::IPerspective_camera>());
    check_success(cam.is_valid_interface());
    setup_camera(cam.get());
    const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
    check_success(camera_tag.is_valid());

    const mi::math::Vector<mi::Uint32, 2> buffer_resolution(1025, 1024);
    m_image_file_canvas->set_resolution(buffer_resolution);

    // Set up the scene
    mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
      {   0.0f,   0.0f,   0.0f, },
      { 500.0f, 500.0f, 500.0f, },
    };

    // set the region of interest
    const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
    check_success(xyz_roi.is_volume());
    scene_edit->set_clipped_bounding_box(xyz_roi_st);

    // Set the scene global transformation matrix.
    // only change the coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
```

```
        0.0f,  0.0f,  0.0f,  1.0f
        );
      scene_edit->set_transform_matrix(transform_mat);

      // Set the current camera to the scene.
      check_success(camera_tag.is_valid());
      scene_edit->set_camera(camera_tag);
    }
    dice_transaction->commit();
  }

  // Rendering
  {
    const mi::Sint32  frame_idx = 0;
    const std::string fname = get_output_file_name(m_outfname, frame_idx);
    mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
   const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;
      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }

    ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
          << os.str();
      exit_code = 1;
    }


    // verify the generated frame
    if (!(verify_canvas_result(get_application_layer_interface(),
                m_image_file_canvas.get(), m_verify_image_fname, get_options())))
    {
      exit_code = 1;
    }
  }

  return exit_code;
}

bool Create_circles_and_ellipses::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
```

```
    sdict.insert("dice::verbose", "2");
  }

  // Set own options
  m_outfname           = sdict.get("outfname", "");
  m_verify_image_fname = sdict.get("verify_image_fname", "");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"
      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")\n"
      << "        [-is_dump_comparison_image_when_failed bool]\n"
    << "        when true, dump the comparison images and generate the diff image for debug.\n"
      << "            You can also force to dump and set the filename.\n"
    << "        (default: " << sdict.get("is_dump_comparison_image_when_failed") << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

bool Create_circles_and_ellipses::create_circles_and_ellipses(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // square shaped positions
  const mi::Sint32  column_count = 15;
```

```
const mi::Sint32  point_count = 60;
const mi::Float32 x_mag = 35.f;
const mi::Float32 y_mag = 35.f;
const mi::Float32 z     = 30.f;
const mi::Float32 y_offset_points = 0.f;
// const mi::Float32 y_offset_lines  = 255.f;

// circle color, radius test.
for(mi::Sint32 i = 0; i < point_count; ++i)
{
 mi::base::Handle<nv::index::ICircle> circle(scene_edit->create_shape<nv::index::ICircle>());
  check_success(circle.is_valid_interface());

  mi::math::Vector_struct< mi::Float32, 3> pos;
  pos.x = static_cast< mi::Float32 >(i % column_count) * x_mag;
  pos.y = static_cast< mi::Float32 >(i / column_count) * y_mag + y_offset_points;
  pos.z = z;

 mi::Float32 radius = (static_cast<mi::Float32>(i%(column_count))/static_cast<mi::Float32>(colum

  circle->set_geometry(pos, radius);

  mi::math::Color_struct color;
  color.r = static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
  color.g = static_cast<mi::Float32>(0.3);
 color.b = 1.f - static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
  color.a = static_cast<mi::Float32>(1.f);
  circle->set_outline_style(color, 2.f);

  mi::math::Color_struct color_bkg = {
                      1.f - color.r,
                      1.f - color.g,
                      1.f - color.b,
                      1.f};
  circle->set_fill_style(color_bkg, nv::index::ICircle::FILL_SOLID);

 const mi::neuraylib::Tag circle_tag = dice_transaction->store_for_reference_counting(circle.get(
  check_success(circle_tag.is_valid());
  group_node->append(circle_tag, dice_transaction);
}

// circle, line_width and transparency test
for(mi::Sint32 i = 0; i < point_count; ++i)
{
 mi::base::Handle<nv::index::ICircle> circle(scene_edit->create_shape<nv::index::ICircle>());
  check_success(circle.is_valid_interface());

  mi::math::Vector_struct< mi::Float32, 3> pos;
  pos.x = static_cast< mi::Float32 >(i % column_count) * x_mag;
  pos.y = static_cast< mi::Float32 >(i / column_count) * y_mag + 140;
  pos.z = z;

  mi::Float32 radius = 20.f;
  circle->set_geometry(pos, radius);

  mi::math::Color_struct color = {1.f, 1.f, 1.f, 1.f};
```

```
 mi::Float32 line_width = ((static_cast<mi::Float32>(i)+3.5f) /static_cast<mi::Float32>(column_co
  circle->set_outline_style(color, line_width);

  mi::math::Color_struct color_bkg = {
                      0.5f,
                      0.5f,
                      1.0f,
                      1.0f - ((static_cast<mi::Float32>(i) + 0.5f)/
                          static_cast<mi::Float32>(point_count))};
  circle->set_fill_style(color_bkg, nv::index::ICircle::FILL_SOLID);

 const mi::neuraylib::Tag circle_tag = dice_transaction->store_for_reference_counting(circle.get(
  check_success(circle_tag.is_valid());
  group_node->append(circle_tag, dice_transaction);
}

// ellipse color and radius test.
for(mi::Sint32 i = 0; i < point_count; ++i)
{
 mi::base::Handle<nv::index::IEllipse> ellipse(scene_edit->create_shape<nv::index::IEllipse>());
  check_success(ellipse.is_valid_interface());

  mi::math::Vector_struct< mi::Float32, 3> pos;
  pos.x = static_cast< mi::Float32 >(i % column_count) * x_mag;
  pos.y = static_cast< mi::Float32 >(i / column_count) * y_mag + 280;
  pos.z = z;

 mi::Float32 radius_x = 8.0f  + 18.0f*((static_cast<mi::Float32>(i) + 0.5f)/static_cast<mi::Float32
 mi::Float32 radius_y = 26.0f - 18.0f*((static_cast<mi::Float32>(i) + 0.5f)/static_cast<mi::Float3

  ellipse->set_geometry(pos, radius_x, radius_y, 0.f);

  mi::math::Color_struct color;
  color.r = static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
  color.g = static_cast<mi::Float32>(0.3);
 color.b = 1.f - static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
  color.a = static_cast<mi::Float32>(1.f);
  ellipse->set_outline_style(color, 1.5f);

  mi::math::Color_struct color_bkg = {
                      1.f - color.r,
                      1.f - color.g,
                      1.f - color.b,
                      1.f};
  ellipse->set_fill_style(color_bkg, nv::index::IEllipse::FILL_SOLID);

 const mi::neuraylib::Tag ellipse_tag = dice_transaction->store_for_reference_counting(ellipse.ge
  check_success(ellipse_tag.is_valid());
  group_node->append(ellipse_tag, dice_transaction);
}

// ellipse roration and transparency test.
for(mi::Sint32 i = 0; i < point_count; ++i)
{
 mi::base::Handle<nv::index::IEllipse> ellipse(scene_edit->create_shape<nv::index::IEllipse>());
  check_success(ellipse.is_valid_interface());
```

```
    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = static_cast< mi::Float32 >(i % column_count) * x_mag;
    pos.y = static_cast< mi::Float32 >(i / column_count) * y_mag + 420;
    pos.z = z;

    mi::Float32 radius_x = 11.f;
    mi::Float32 radius_y = 26.f;
   mi::Float32 rotation = (static_cast<mi::Float32>(i)*3.14159f)/static_cast<mi::Float32>(point_cou

    ellipse->set_geometry(pos, radius_x, radius_y, rotation);

    mi::math::Color_struct color = {1.f, 1.f, 1.f, 1.f};
    ellipse->set_outline_style(color, 2.0f);

    mi::math::Color_struct color_bkg = {
                        0.5f,
                        0.5f,
                        1.0f,
                        1.0f - ((static_cast<mi::Float32>(i) + 0.5f)/
                            static_cast<mi::Float32>(point_count))};
    ellipse->set_fill_style(color_bkg, nv::index::IEllipse::FILL_SOLID);

   const mi::neuraylib::Tag ellipse_tag = dice_transaction->store_for_reference_counting(ellipse.ge
    check_success(ellipse_tag.is_valid());
    group_node->append(ellipse_tag, dice_transaction);
  }

  // background ellipse
  {
   mi::base::Handle<nv::index::IEllipse> ellipse(scene_edit->create_shape<nv::index::IEllipse>());
    check_success(ellipse.is_valid_interface());

    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = 250.f;
    pos.y = 290.f;
    pos.z = 5.f*z;

    mi::Float32 radius_x = 250.f;
    mi::Float32 radius_y = 500.f;
    mi::Float32 rotation = -3.14159f/4.f;

    ellipse->set_geometry(pos, radius_x, radius_y, rotation);

    mi::math::Color_struct color = {0.5f, 1.0f, 0.5f, 1.f};
    ellipse->set_outline_style(color, 5.0f);

    mi::math::Color_struct color_bkg = {1.0f, 0.5f, 0.5f, 1.f};
    ellipse->set_fill_style(color_bkg, nv::index::IEllipse::FILL_SOLID);

   const mi::neuraylib::Tag ellipse_tag = dice_transaction->store_for_reference_counting(ellipse.ge
    check_success(ellipse_tag.is_valid());
    group_node->append(ellipse_tag, dice_transaction);
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);
```

```
  return true;
}

void Create_circles_and_ellipses::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from( 254.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_circles_and_ellipses::render_frame(const std::string& output_fnar
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                              // log level
  sdict.insert("dice::network::mode", "OFF");                      // network mode
 sdict.insert("outfname", "frame_create_circles_and_ellipses"); // output file base name
```

```
  sdict.insert("verify_image_fname", "");                          // for unit test
 sdict.insert("is_dump_comparison_image_when_failed", "1");    // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                 // default: not call from make check.
  sdict.insert("unittest", "0");                                   // default mode

  // Load IndeX library via Index_connect
  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // Initialize application
  Create_circles_and_ellipses create_circles_and_ellipses;
  create_circles_and_ellipses.initialize(argc, argv, sdict);
  check_success(create_circles_and_ellipses.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_circles_and_ellipses.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.10 create_icons.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <iostream>
#include <sstream>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iindex.h>
#include <nv/index/iicon.h>
#include <nv/index/isession.h>
#include <nv/index/iscene.h>
#include <nv/index/iscene_group.h>
#include <nv/index/itexture.h>

#include <nv/index/app/iimage_importer.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>

#include "utility/canvas_utility.h"

class Create_icons:
  public nv::index::app::Index_connect
{
public:
  Create_icons()
    :
    Index_connect(),
    m_outfname()
  {
    // INFO_LOG << "DEBUG: Create_icons() ctor";
  }

  virtual ~Create_icons()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_icons() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& options) CPP11_OVERRIDE;
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
```

```
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // create the scene
  mi::neuraylib::Tag create_scene(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction);

  void move_scene(
    mi::neuraylib::Tag                group_node_tag,
    mi::Uint32                        frame_id,
    mi::neuraylib::IDice_transaction* dice_transaction);

  void setup_camera(nv::index::IPerspective_camera* cam);

  nv::index::IFrame_results* render_frame(
    const std::string& output_fname);

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                              m_iconfile;
  std::string                                                 m_verify_image_fname_sequence;
  std::string                                                    m_verify_image_fname;
};

mi::Sint32 Create_icons::launch()
{
  m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configurat
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  mi::neuraylib::Tag group_node = mi::neuraylib::NULL_TAG;
  mi::neuraylib::Tag camera_tag = mi::neuraylib::NULL_TAG;

  mi::Sint32 exit_code = 0;
```

```
{
  // Obtain a DiCE transaction
 mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
  check_success(dice_transaction.is_valid_interface());
  {
    // Setup session information
    m_session_tag = m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<const nv::index::ISession>(m_session_tag));
    check_success(session.is_valid_interface());
    mi::base::Handle< nv::index::IScene > scene_edit(
      dice_transaction->edit< nv::index::IScene >(session->get_scene()));
    check_success(scene_edit.is_valid_interface());

    // Scene setup
    group_node = create_scene(scene_edit.get(), dice_transaction.get());
    check_success(group_node.is_valid());

    // Create a camera
    mi::base::Handle< nv::index::IPerspective_camera > cam(
      scene_edit->create_camera<nv::index::IPerspective_camera>());
    check_success(cam.is_valid_interface());
    setup_camera(cam.get());

    camera_tag = dice_transaction->store(cam.get());
    check_success(camera_tag.is_valid());

    const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
    m_image_file_canvas->set_resolution(buffer_resolution);

    // Set up the region of interest
    const mi::math::Bbox_struct<mi::Float32, 3> xyz_roi_st = {
      {   0.0f,   0.0f,   0.0f, },
      { 500.0f, 500.0f, 500.0f, },
    };

    // set the region of interest
    const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
    check_success(xyz_roi.is_volume());
    scene_edit->set_clipped_bounding_box(xyz_roi_st);

    // Set the scene global transformation matrix.
    // only change the coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
      0.0f,  0.0f,  0.0f,  1.0f
      );
    scene_edit->set_transform_matrix(transform_mat);

    // Set the current camera to the scene.
    check_success(camera_tag.is_valid());
    scene_edit->set_camera(camera_tag);
  }
  dice_transaction->commit();
```

```
  }

  // Rendering
  {
    const std::string fname = get_output_file_name(m_outfname, 0);
    mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
   const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;
      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }

    ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
          << os.str();
    }

    // verify the generated frame
    if (!(verify_canvas_result(get_application_layer_interface(),
                m_image_file_canvas.get(), m_verify_image_fname, get_options())))
    {
      exit_code = 1;
    }
  }

  // Render a few more frames
  for (mi::Uint32 i = 1; i < 20; ++i)
  {
    // Move some scene elements
    {
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
      check_success(dice_transaction.is_valid_interface());
      {
        move_scene(group_node, i, dice_transaction.get());
      }
      dice_transaction->commit();
    }

    const std::string fname = get_output_file_name(m_outfname, i);
    mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
   const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;
      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }
```

```
    ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
           << os.str();
    }

    // verify the generated frame
   const std::string verify_image_fname = get_output_file_name(m_verify_image_fname_sequence, i);
    if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), verify_image_fname, get_options())))
    {
      exit_code = 1;
    }
  }

  return exit_code;
}

bool Create_icons::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
    sdict.insert("verify_image_fname_sequence", ""); // disable this in unit test
  }

  // Set own options
  m_outfname = sdict.get("outfname", "");
  m_iconfile = sdict.get("iconfile", "");
  m_verify_image_fname_sequence = sdict.get("verify_image_fname_sequence", "");
  if (m_is_unittest)
  {
    m_verify_image_fname = sdict.get("verify_image_fname", "");
  }
  else
  {
    if (!sdict.get("verify_image_fname_sequence", "").empty())
    {
      m_verify_image_fname = m_verify_image_fname_sequence + "000.ppm";
    }
  }

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["                      + m_outfname +
       "], iconfile = ["                    + m_iconfile +
       "], verify_image_fname_sequence = [" + m_verify_image_fname_sequence +
       "], verify_image_fname = ["          + m_verify_image_fname +
       "], dice::verbose = "                + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
```

```
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"
      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-iconfile ICON_FPATH]\n"
      << "            Specify an icon file. (default: ["
      << m_iconfile << "])\n"

      << "        [-verify_image_fname_sequence [image_fname_base]]\n"
    << "        when image_fname_base exist, verify the rendering image sequence.  (default: ["
      << m_verify_image_fname_sequence << "])\n"
      << "            This is for non-unit test mode.\n"

      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"
      << "            This is for unit test mode.\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mod. "
      << "(default: " << m_is_unittest << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

mi::neuraylib::Tag Create_icons::create_scene(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  // Create the main scene group which will contain all shapes
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Set up the transformation matrix
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  // Add a translation
  transform_mat.translate(mi::math::Vector<mi::Float32, 3>(5.0f, 0.0f, 0.0f));
  group_node->set_transform(transform_mat);

  // Create a texture attribute
  mi::math::Vector<mi::Sint32, 2> tex_resolution(0, 0);
```

```
  {
   mi::base::Handle<nv::index::ITexture> texture(scene_edit->create_attribute<nv::index::ITexture>
    check_success(texture.is_valid_interface());
    check_success(!m_iconfile.empty());

    // Load the texture from a file
    mi::base::Handle<nv::index::app::image::IImage_importer> image_importer(
    get_application_layer_interface()->get_api_component<nv::index::app::image::IImage_importer>(
    check_success(image_importer.is_valid_interface());
    mi::base::Handle<mi::neuraylib::ICanvas> canvas(
      image_importer->create_canvas_from_file(m_iconfile.c_str()));
    check_success(canvas.is_valid_interface());
    tex_resolution.x = static_cast<mi::Sint32>(canvas->get_resolution_x());
    tex_resolution.y = static_cast<mi::Sint32>(canvas->get_resolution_y());

    // Copy the pixel data
    check_success(texture->set_pixel_data(
             canvas.get(),
             nv::index::ITexture::RGBA_FLOAT32));

   mi::neuraylib::Tag texture_tag = dice_transaction->store_for_reference_counting(texture.get());
    check_success(texture_tag.is_valid());
    group_node->append(texture_tag, dice_transaction);
  }

  // Icon
  {
   mi::base::Handle<nv::index::IIcon_2D> icon(scene_edit->create_shape<nv::index::IIcon_2D>());
    check_success(icon.is_valid_interface());

    const mi::math::Vector<mi::Float32, 3> position(0.f, 0.f, 0.f);
    const mi::math::Vector<mi::Float32, 2> right(1.f, 0.f);
    const mi::math::Vector<mi::Float32, 2> up(0.f, 1.f);
    const mi::Float32 width  = 50.0f;
   const mi::Float32 height = width * (static_cast<mi::Float32>(tex_resolution.x) / static_cast<mi::I
    icon->set_geometry(position, right, up, width, height);

   mi::neuraylib::Tag icon_tag = dice_transaction->store_for_reference_counting(icon.get());
    check_success(icon_tag.is_valid());
    group_node->append(icon_tag, dice_transaction);
  }

  {
   mi::base::Handle<nv::index::IIcon_3D> icon(scene_edit->create_shape<nv::index::IIcon_3D>());
    check_success(icon.is_valid_interface());

    const mi::math::Vector<mi::Float32, 3> position(50.f, 50.f, 50.f);
    const mi::math::Vector<mi::Float32, 3> right(2.f, 0.f, 1.0f);
    const mi::math::Vector<mi::Float32, 3> up(0.f, 2.f, 0.f);
    const mi::Float32 width  = 100.0f;
   const mi::Float32 height = width * (static_cast<mi::Float32>(tex_resolution.x) / static_cast<mi::I
    icon->set_geometry(position, right, up, width, height);

   mi::neuraylib::Tag icon_tag = dice_transaction->store_for_reference_counting(icon.get());
    check_success(icon_tag.is_valid());
    group_node->append(icon_tag, dice_transaction);
  }
```

```
  // Append the scene group to the scene
 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  INFO_LOG << "Hierachical scene description creation complete.";
  return group_node_tag;
}

void Create_icons::move_scene(
  mi::neuraylib::Tag                group_node_tag,
  mi::Uint32                        frame_id,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    dice_transaction->edit<nv::index::ITransformed_scene_group>(group_node_tag));
  check_success(group_node.is_valid_interface());

  // Set up the transformation matrix
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  // Add a translation
 transform_mat.translate(mi::math::Vector<mi::Float32, 3>(5.f, 0.f, frame_id * 10.f));
  group_node->set_transform(transform_mat);
}

void Create_icons::setup_camera(
  nv::index::IPerspective_camera* cam)
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene.
  const mi::math::Vector<mi::Float32, 3> from(100.0f, 254.0f,  550.0f);
  const mi::math::Vector<mi::Float32, 3> to  (255.0f, 255.0f, -255.0f);
  const mi::math::Vector<mi::Float32, 3> up  (  0.0f,   1.0f,    0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_icons::render_frame(
  const std::string& output_fname)
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

 mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
```

```
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;

  // Application (Create_icon) settings
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("outfname", "frame_create_icons");            // output file base name
  sdict.insert("iconfile", "../create_icons/nvidia_logo.ppm"); // icon file
  sdict.insert("verify_image_fname_sequence", "");             // for non-unit test
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1");  // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");               // default: not call from make check.

  // DiCE settings
  sdict.insert("dice::network::mode", "OFF");

  // IndeX settings
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // Application_layer settings
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io");

  // Initialize application
  Create_icons create_icon;
  create_icon.initialize(argc, argv, sdict);
  check_success(create_icon.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_icon.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.11 create_line_set.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iindex.h>
#include <nv/index/iline_set.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/canvas_utility.h"
#include "utility/app_rendering_context.h"

#include <iostream>
#include <sstream>

class Create_line_set:
  public nv::index::app::Index_connect
{
public:
  Create_line_set()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_line_set() ctor";
  }

  virtual ~Create_line_set()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_line_set() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
```

```
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // get a color value from an integer
  // \param[in] attrib attribute value
  // \return a color associated with an attribute
  mi::math::Color_struct get_color_from_attribute(mi::Sint32 attrib) const;

  // get a width value from an integer
  // \param[in] attrib attribute value
  // \return a width value associated with an attribute
  mi::Float32 get_width_from_attribute(mi::Float32 attrib) const;

  // create line set in the scene.
  //
  // \param[in] scene_edit       IScene for the scene edit.
  // \param[in] dice_transaction dice transaction
  // \return true when success
  bool create_line_set(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  // setup camera to see this example scene
  //
  // \param[in] cam              a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;


  // This session tag
  mi::neuraylib::Tag                                                  m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                    m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                            m_outfname;
  bool                                                                m_is_unittest;
  std::string                                                      m_verify_image_fname;
};

mi::Sint32 Create_line_set::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    m_cluster_configuration =
      get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
```

```
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    // This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));
    {
      // DiCE database access
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        // Setup session information
        m_session_tag =
          m_index_session->create_session(dice_transaction.get());
        check_success(m_session_tag.is_valid());
        mi::base::Handle< nv::index::ISession const > session(
          dice_transaction->access< nv::index::ISession const >(
            m_session_tag));
        check_success(session.is_valid_interface());

        mi::base::Handle< nv::index::IScene > scene_edit(
          dice_transaction->edit<nv::index::IScene>(session->get_scene()));
        check_success(scene_edit.is_valid_interface());

        //------------------------------------------------------------------------
        // Scene setup: add line set shape, scene parameters, camera.
        //------------------------------------------------------------------------
        // Add line set shape to the scene
        check_success(create_line_set(scene_edit.get(), dice_transaction.get()));

        // Create and edit a camera. Data distribution is based on
        // the camera. (Because only visible massive data are
        // considered)
        mi::base::Handle< nv::index::IPerspective_camera > cam(
          scene_edit->create_camera<nv::index::IPerspective_camera>());
        check_success(cam.is_valid_interface());
        setup_camera(cam.get());
        const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
        check_success(camera_tag.is_valid());

        const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
        m_image_file_canvas->set_resolution(buffer_resolution);

        // Set up the scene
        mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
          {   0.0f,   0.0f,   0.0f, },
          { 500.0f, 500.0f, 500.0f, },
        };

        // set the region of interest
        const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
        check_success(xyz_roi.is_volume());
        scene_edit->set_clipped_bounding_box(xyz_roi_st);
```

```
        // Set the scene global transformation matrix.
        // only change the coordinate system
        mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
          1.0f,  0.0f,  0.0f,  0.0f,
          0.0f,  1.0f,  0.0f,  0.0f,
          0.0f,  0.0f, -1.0f,  0.0f,
          0.0f,  0.0f,  0.0f,  1.0f
          );
        scene_edit->set_transform_matrix(transform_mat);

        // Set the current camera to the scene.
        check_success(camera_tag.is_valid());
        scene_edit->set_camera(camera_tag);
      }
      dice_transaction->commit();
    }

    // Rendering
    {
      const mi::Sint32  frame_idx = 0;
      const std::string fname = get_output_file_name(m_outfname, frame_idx);
     mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      if (err_set->any_errors())
      {
        std::ostringstream os;
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
            << os.str();
        exit_code = 1;
      }


      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), m_verify_image_fname, get_options())))
      {
        exit_code = 1;
      }
    }
  }
  // index_connect shutdown
  return exit_code;
}

bool Create_line_set::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));
```

```
  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname           = sdict.get("outfname", "");
  m_verify_image_fname = sdict.get("verify_image_fname", "");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"
      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"
      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")"
      << std::endl;
    exit(1);
  }
  return true;
}

mi::math::Color_struct Create_line_set::get_color_from_attribute(mi::Sint32 attrib) const
{
  mi::math::Color_struct col;
  const mi::Float32 coef = 1.0f / 15.0f;
  col.r = coef * static_cast< mi::Float32 >(attrib & 15);
  col.g = coef * static_cast< mi::Float32 >((attrib >> 4) & 15);
  col.b = coef * static_cast< mi::Float32 >((attrib >> 8) & 15);
  col.a = 1.0;

  return col;
}
```

```
mi::Float32 Create_line_set::get_width_from_attribute(mi::Float32 attrib) const
{
  const mi::Float32 width =
    static_cast< mi::Float32 >(static_cast< mi::Sint32 >(fabs(attrib)) % 16);
  return width;
}

bool Create_line_set::create_line_set(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  // hierarchical scene description node for the point set
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Line coordinates and its attributes. According to the
  // attributes, color and width are defined.
  // Here, we create two line sets.
  std::vector< mi::math::Vector_struct< mi::Float32, 3> > line_seg_vec[2];
  std::vector< mi::math::Color_struct > color_vec[2];
  std::vector< mi::Float32 > width_vec[2];

  // Square shaped positions
  const mi::Sint32  column_count       = 20;
  const mi::Sint32  line_segment_count = 200;
  const mi::Sint32  line_set_count     = 2;
  const mi::Float32 x_org              =  0.0;
  const mi::Float32 y_org              =  0.0;
  const mi::Float32 x_mag              = 25.0;
  const mi::Float32 y_mag              = 25.0;
  const mi::Float32 z                  = 30.0;
  const mi::Float32 y_offset[2]        = { 0.0f, 280.0f };
  mi::Sint32 idx_p = 0; // point index

  for(mi::Sint32 i = 0; i < line_segment_count; ++i)
  {
    // one segment has two points
    for(mi::Sint32 k = 0; k < 2; ++k){
      mi::math::Vector_struct< mi::Float32, 3> pos[2];
      for(mi::Sint32 j = 0; j < line_set_count; ++j){
        pos[j].x = x_org + static_cast< mi::Float32 >(idx_p % column_count) * x_mag;
     pos[j].y = y_org + static_cast< mi::Float32 >(idx_p / column_count) * y_mag + y_offset[j];
        pos[j].z = z;
        line_seg_vec[j].push_back(pos[j]);
      }
      ++idx_p;
    }
    for(mi::Sint32 j = 0; j < line_set_count; ++j){
      color_vec[j].push_back(get_color_from_attribute(i));
      width_vec[j].push_back(get_width_from_attribute(static_cast<mi::Float32>(i)));
    }
  }
```

```
  // Create two line sets
  for(mi::Sint32 j = 0; j < 2; ++j)
  {
    // Create attribute_line_set scene element and add it to the scene
   mi::base::Handle<nv::index::ILine_set> line_set(scene_edit->create_shape<nv::index::ILine_set>(
    check_success(line_set.is_valid_interface());
    line_set->set_line_type(nv::index::ILine_set::LINE_TYPE_SEGMENTS);

    check_success((line_seg_vec[j].size()) > 0);
   line_set->set_lines(&(line_seg_vec[j][0]), line_seg_vec[j].size() / 2); // may cut the last point
    line_set->set_colors(&(color_vec[j][0]), color_vec[j].size());
    line_set->set_widths(&(width_vec[j][0]), width_vec[j].size());

    // Add the line segments to the database
   const mi::neuraylib::Tag line_set_tag = dice_transaction->store_for_reference_counting(line_set.
    // if you are not registered Attribute_line_set, the next check fails.
    check_success(line_set_tag.is_valid());
    // Add to the scene description
    group_node->append(line_set_tag, dice_transaction);
   INFO_LOG << "Added line_set (size: " << line_set_count << ") to the scene (tag id: "
        << line_set_tag.id << ").";
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  return true;
}

void Create_line_set::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene.
  mi::math::Vector< mi::Float32, 3 > const from( 254.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_line_set::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());
```

```
  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("outfname", "frame_create_line_set");         // output file base name
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // Initialize application
  Create_line_set create_line_set;
  create_line_set.initialize(argc, argv, sdict);
  check_success(create_line_set.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_line_set.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.12 create_path.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/icolormap.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/iindex.h>
#include <nv/index/iindex_debug_configuration.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/ipath.h>
#include <nv/index/iplane.h>
#include <nv/index/iscene.h>
#include <nv/index/iscene_group.h>
#include <nv/index/isession.h>
#include <nv/index/itexture_filter_mode.h>

#include <nv/index/ipath_query_results.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/idata_analysis_and_processing.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Test_tag_info
{
public:
  // default constructor
  Test_tag_info() :
    m_assoc_type("<not defined>"),
    m_num_points(-1)
  {
    // empty
  }
  // default constructor
  Test_tag_info(const std::string & type_name, mi::Sint32 num_points) :
    m_assoc_type(type_name),
    m_num_points(num_points)
  {
    // empty
  }
  // destructor
  ~Test_tag_info()
  {
    // empty
  }
```

```
  // copy constructor
  Test_tag_info(const Test_tag_info & tti)
  {
    m_assoc_type = tti.m_assoc_type;
    m_num_points = tti.m_num_points;
  }
  // operator=
  Test_tag_info & operator=(const Test_tag_info & rhs)
  {
    if(this != &rhs){
      this->m_assoc_type = rhs.m_assoc_type;
      this->m_num_points = rhs.m_num_points;
    }
    return (*this);
  }

public:
  // tag associated type name
  std::string        m_assoc_type;
  // number of points if the scene element has, otherwise, -1
  mi::Sint32         m_num_points;

private:
};

class Tag_comp {
public:
 bool operator()(const mi::neuraylib::Tag & lhs, const mi::neuraylib::Tag & rhs) const
  {
    return lhs.id < rhs.id;
  }
};

class Create_path:
  public nv::index::app::Index_connect
{
public:
  struct Path_test_params
  {
    mi::Uint32  num_points;
    bool        use_colormap;
    bool        use_rgba;
    bool        npr_mode;
    bool        test_transparency;
    bool        upsampling;
    mi::Uint32  up_factor;
    mi::Float32 up_tension;
  };

  // ctor
  Create_path()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_path() ctor";
  }
```

```
  virtual ~Create_path()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_path() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  mi::Float32 evaluate_path(mi::Float32 x) const;
 mi::math::Color_struct jetmap(mi::Float32 v, mi::Float32 vmin, mi::Float32 vmax) const;

 nv::index::IColormap* create_colormap(nv::index::IScene* scene, bool use_transparency) const;
  // Create scene.
  bool create_scene(
    nv::index::IScene*                                    scene_edit,
    std::vector<mi::neuraylib::Tag>&                      path_tags,
    std::map< mi::neuraylib::Tag, Test_tag_info, Tag_comp >& test_tag_info_map,
    mi::neuraylib::IDice_transaction*                     dice_transaction) const;

  // setup camera to see this example scene
  // \param[in] cam    a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;
  // Setup a far away camera for a extreme transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  // \param[in] cam    a camera
  void setup_extreme_transformed_camera(nv::index::IPerspective_camera* cam) const;

  // setup a large scene transform for the extreme
  // transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
```

```
  mi::math::Matrix<mi::Float32, 4, 4> get_extreme_transformed_matrix() const;
  // render a frame
  //
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                   m_is_unittest;
  std::string                                                         m_verify_image_fname;
  bool                                                                 m_supersampling;
  bool                                                                m_is_large_translate;
  Path_test_params                                                      m_path_test_params;
};


mi::Sint32 Create_path::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    mi::base::Handle<nv::index::IIndex_scene_query> iindex_query(
      get_index_interface()->get_api_component<nv::index::IIndex_scene_query>());
    check_success(iindex_query.is_valid_interface());

    m_cluster_configuration =
      get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    // This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));

    std::map< mi::neuraylib::Tag, Test_tag_info, Tag_comp > test_tag_info_map;
   std::vector<mi::neuraylib::Tag> path_tags; // used for later entry lookup queries.
    {
      // DiCE database access
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        // Setup session information
        m_session_tag = m_index_session->create_session(dice_transaction.get());
        check_success(m_session_tag.is_valid());
        mi::base::Handle<const nv::index::ISession> session(
```

```
      dice_transaction->access<const nv::index::ISession>(m_session_tag));
    check_success(session.is_valid_interface());

    mi::base::Handle< nv::index::IScene > scene_edit(
      dice_transaction->edit< nv::index::IScene >(session->get_scene()));
    check_success(scene_edit.is_valid_interface());

    // Enable supersampling
    if (m_supersampling)
    {
      INFO_LOG << "Enable full-screen supersampling.";
      mi::base::Handle<nv::index::IConfig_settings> config_settings(
    dice_transaction->edit<nv::index::IConfig_settings>(session->get_config()));
      check_success(config_settings.is_valid_interface());

      config_settings->set_rendering_samples(16);
    }

    //-------------------------------------------------------------------------
    // Scene setup: hierarchical scene description root node,
    // scene parameters, camera.
    //-------------------------------------------------------------------------
  check_success(create_scene(scene_edit.get(), path_tags, test_tag_info_map, dice_transaction.g
    check_success(test_tag_info_map.size() == path_tags.size());

    // Create and edit a camera. Data distribution is based on
    // the camera. (Because only visible massive data are considered)
    mi::base::Handle< nv::index::IPerspective_camera > cam(
      scene_edit->create_camera<nv::index::IPerspective_camera>());
    check_success(cam.is_valid_interface());

    mi::math::Vector<mi::Uint32, 2> buffer_resolution;
    if (m_is_large_translate)
    {
      INFO_LOG << "large translate test mode.";
      setup_extreme_transformed_camera(cam.get());
      buffer_resolution = mi::math::Vector<mi::Uint32, 2>(1602, 934);
    }
    else
    {
      setup_camera(cam.get());
      buffer_resolution = mi::math::Vector<mi::Uint32, 2>(1024, 1024);
    }
    const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
    check_success(camera_tag.is_valid());

    m_image_file_canvas->set_resolution(buffer_resolution);

    // Set up the scene
    const mi::math::Bbox_struct< mi::Float32, 3 > xyz_roi_st = {
      {   -1000.0f,   -1000.0f,   -1000.0f, },
      { 1000.0f, 1000.0f, 1000.0f, },
    };

    // set the region of interest
    const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
    check_success(xyz_roi.is_volume());
```

```
      scene_edit->set_clipped_bounding_box(xyz_roi_st);

      // Set the scene global transformation matrix.
      // only change the coordinate system
      mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
        1.0f,  0.0f,  0.0f,  0.0f,
        0.0f,  1.0f,  0.0f,  0.0f,
        0.0f,  0.0f, -1.0f,  0.0f,
        0.0f,  0.0f,  0.0f,  1.0f
        );

      if (m_is_large_translate)
      {
        transform_mat = get_extreme_transformed_matrix();
      }

      scene_edit->set_transform_matrix(transform_mat);

      INFO_LOG << "transformed: " << scene_edit->get_transform_matrix();

      // Set the current camera to the scene.
      check_success(camera_tag.is_valid());
      scene_edit->set_camera(camera_tag);
    }
    dice_transaction->commit();
  }

  // Rendering
  {
    // Render a frame and save the rendered image to a file.
    const mi::Sint32  frame_idx = 0;
    const std::string fname = get_output_file_name(m_outfname, frame_idx);
   mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
  const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;


      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }

   ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
          << os.str();
      exit_code = 1;
    }
  }

  // Picking
  {
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
```

```
check_success(dice_transaction.is_valid_interface());
{
  mi::math::Vector_struct<mi::Uint32, 2> pick_location;
  pick_location.x = 256;
  pick_location.y = 314;
  INFO_LOG << "Query pick location: " << pick_location;
  mi::base::Handle<nv::index::IScene_pick_results> scene_pick_results(
 iindex_query->pick(pick_location, m_image_file_canvas.get(), m_session_tag, dice_transaction
  const mi::Uint32 nb_results = scene_pick_results->get_nb_results();
  if(nb_results>0)
  {
    INFO_LOG << "Number of pick results: " << nb_results;
    for(mi::Uint32 i=0; i<nb_results; ++i)
    {
  const mi::base::Handle<nv::index::IScene_pick_result> result(scene_pick_results->get_resul
  const mi::math::Vector_struct<mi::Float32, 3>& intersection = result->get_intersection();
      INFO_LOG << "Intersection no. " << i << "\n"
      << "\t\t Element (tag)        " << result->get_scene_element().id << "\n"
     << "\t\t Sub index:          " << result->get_scene_element_sub_index() << "\n"
          << "\t\t Distance:               " << result->get_distance() << "\n"
          << "\t\t Position (local space): " << intersection << "\n"
          << "\t\t Color (evaluated):      " << result->get_color();

  if(result->get_intersection_info_class() == nv::index::IPath_pick_result::IID() )
      {
        mi::base::Handle<const nv::index::IPath_pick_result> path_pick_result(
          result->get_interface<const nv::index::IPath_pick_result>());
        if(path_pick_result.get())
        {
          INFO_LOG << "Path specific pick results:" << "\n"
           << "\t\t Segment id:           " << path_pick_result->get_segment();
        }
      }
    }
  }

  const mi::Size nb_tags = path_tags.size();
  for (mi::Size i = 0; i < nb_tags; ++i)
  {
    const mi::neuraylib::Tag path_scene_element_tag = path_tags[i];
 check_success(test_tag_info_map.find(path_scene_element_tag) != test_tag_info_map.end());
    const Test_tag_info tti = test_tag_info_map[path_scene_element_tag];

    // The loop is supposed to query every entry of the given path until
    // an entry for the given index value is not available anymore, i.e., the
    // length of the path has been exceeded.
   // Once the path has been exceeded, the example should continue with the next
    // path (outer loop).
    const mi::Uint32 LOOP_DELTA = 9;
    for(mi::Uint32 index = 0; index < 100; index += LOOP_DELTA)
    {
      if(tti.m_num_points <= static_cast< mi::Sint32 >(index)){
        WARN_LOG << "Note: expected the following warning: "
      << "The index " << index << " is greater than the number of entries in the path "
            << "(entry count: " << tti.m_num_points <<  "). for tag: "
            << path_scene_element_tag.id << ", type: " << tti.m_assoc_type
            << " with access index: " << index << " in this test.";
```

```
            }

            // Issue an entry lookup query
        mi::base::Handle<nv::index::IScene_lookup_result> lookup_result(iindex_query->entry_lookup
                                              index, // index
                                              path_scene_element_tag,
                                              m_session_tag,
                                              dice_transaction.get()));

            // Verify if the data entry available at path location.
            if(!lookup_result.is_valid_interface())
            {
              WARN_LOG << "No entry available at path index: " << index;
             INFO_LOG << "Please note, the path might simply have too few entries.";
              check_success(tti.m_num_points <= static_cast< mi::Sint32 >(index));
              break;
            }

            // Verify the query results
        if(lookup_result->get_intersection_info_class() == nv::index::IPath_lookup_result::IID())
            {
          mi::base::Handle<const nv::index::IPath_lookup_result> path_lookup_result(
              lookup_result->get_interface<const nv::index::IPath_lookup_result>());
              if(path_lookup_result.get())
              {
                INFO_LOG << "Path specific lookup results:";
                INFO_LOG << "\t Tag id:      " << path_scene_element_tag.id;
                INFO_LOG << "\t Entry index: " << index;
                INFO_LOG << "\t Vertex:      " << path_lookup_result->get_vertex();
                INFO_LOG << "\t Color index: " << path_lookup_result->get_radius();
                INFO_LOG << "\t Color:       " << path_lookup_result->get_color();
             INFO_LOG << "\t Color index: " << path_lookup_result->get_color_index();
              }
            }
          }
        }

        // verify the generated frame
        if (!(verify_canvas_result(get_application_layer_interface(),
                    m_image_file_canvas.get(), m_verify_image_fname, get_options())))
        {
          exit_code = 1;
        }

      }
      // Finish this transaction
      dice_transaction->commit();
    }
  }

  return exit_code;
}

bool Create_path::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));
```

```
 if (m_is_unittest)
 {
   if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
   {
     sdict.insert("is_dump_comparison_image_when_failed", "0");
   }
   sdict.insert("outfname", "");  // turn off file output in the unit test mode
   sdict.insert("dice::verbose", "2");
 }

 m_outfname           = sdict.get("outfname", "");
 m_verify_image_fname = sdict.get("verify_image_fname");
m_supersampling    = nv::index::app::get_bool(sdict.get("supersampling", "false"));
m_is_large_translate = nv::index::app::get_bool(sdict.get("is_large_translate", "false"));
 // Path_test_params
m_path_test_params.num_points     = nv::index::app::get_uint32(sdict.get("num_samples",     ""))
m_path_test_params.use_colormap    = nv::index::app::get_bool( sdict.get("use_colormap",    "fal
m_path_test_params.use_rgba       = nv::index::app::get_bool( sdict.get("use_rgba",       "false"
m_path_test_params.npr_mode       = nv::index::app::get_bool( sdict.get("npr_mode",       "false"
m_path_test_params.test_transparency = nv::index::app::get_bool( sdict.get("test_transparency", "
m_path_test_params.upsampling     = nv::index::app::get_bool( sdict.get("upsampling",     "false
m_path_test_params.up_factor      = nv::index::app::get_uint32(sdict.get("up_factor",      "2"))
m_path_test_params.up_tension     = nv::index::app::get_float32(sdict.get("up_tension",    "0")

 info_cout(std::string("running ") + com_name, sdict);
 info_cout(std::string("outfname = [") + m_outfname +
      "], verify_image_fname = [" + m_verify_image_fname +
      "], dice::verbose = " + sdict.get("dice::verbose"), sdict);

 // print help and exit if -h
 if(sdict.is_defined("h")){
   std::cout
     << "info: Usage: " << com_name << " [option]\n"
     << "Option: [-h]\n"
     << "            printout this message\n"
     << "        [-dice::verbose severity_level]\n"
   << "         verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
     << ")\n"

     << "        [-supersampling bool]\n"
     << "            when true then full screen supersampling is enabled."
     << "(default: " << m_supersampling << ")\n"

     << "        [-num_samples int]\n"
     << "            set the number of path samples for the test."
     << "(default: " << m_path_test_params.num_points << ")\n"

     << "        [-use_colormap bool]\n"
     << "            enable/disable colormap source."
     << "(default: " << m_path_test_params.use_colormap << ")\n"

     << "        [-use_rgba bool]\n"
     << "            enable/disable rgba array source."
     << "(default: " << m_path_test_params.use_rgba << ")\n"

     << "        [-npr_mode bool]\n"
```

```
      << "            enable/disable non-photorealistic mode."
      << "(default: " << m_path_test_params.npr_mode << ")\n"

      << "        [-test_transparency bool]\n"
      << "            enable/disable transparency test."
      << "(default: " << m_path_test_params.test_transparency << ")\n"

      << "        [-upsampling bool]\n"
      << "            enable/disable upsampling."
      << "(default: " << m_path_test_params.upsampling << ")\n"

      << "        [-up_factor int]\n"
      << "            upsampling factor. Increase of the sampling rate factor."
      << "(default: " << m_path_test_params.up_factor << ")\n"

      << "        [-up_tension float]\n"
      << "            upsampling tension. tension of the fitting curve [0.0, 1.0]."
      << "(default: " << m_path_test_params.up_tension << ")\n"

      << "        [-is_large_translate bool]\n"
      << "            on/off large translation mode."
      << "(default: " << m_is_large_translate << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")"
      << std::endl;
    exit(1);
  }
  return true;
}

mi::Float32 Create_path::evaluate_path(mi::Float32 x) const
{
  return expf(-x) + 0.15f*cosf(3.f*static_cast<mi::Float32>(MI_PI)*x/4.f);
}

mi::math::Color_struct Create_path::jetmap(mi::Float32 v, mi::Float32 vmin, mi::Float32 vmax) const
{
  mi::math::Color_struct c = {1.f, 1.f, 1.f, 1.f};
  mi::Float32 dv;

  if (v < vmin)
    v = vmin;
  if (v > vmax)
    v = vmax;
  dv = vmax - vmin;

  if (v < (vmin + 0.25f * dv))
```

```
  {
    c.r = 0.f;
    c.g = 4.f * (v - vmin) / dv;
  }
  else if (v < (vmin + 0.5f * dv))
  {
    c.r = 0.f;
    c.b = 1.f + 4.f * (vmin + 0.25f * dv - v) / dv;
  }
  else if (v < (vmin + 0.75f * dv))
  {
    c.r = 4.f * (v - vmin - 0.5f * dv) / dv;
    c.b = 0.f;
  }
  else
  {
    c.g = 1.f + 4.f * (vmin + 0.75f * dv - v) / dv;
    c.b = 0.f;
  }

  return(c);
}

nv::index::IColormap* Create_path::create_colormap(nv::index::IScene* scene, bool use_transparency)
{
  const mi::Float32 MIN_TRANS = 0.2f;
  const mi::Float32 MAX_TRANS = 0.9f;

  nv::index::IColormap* colormap = scene->create_attribute<nv::index::IColormap>();
  check_success(colormap != 0);
  std::vector<mi::math::Color_struct> colormap_entries;
  colormap_entries.resize(256);
  for(mi::Uint32 i=0; i<256; ++i)
  {
   colormap_entries[i] = jetmap((static_cast<mi::Float32>(i)+0.5f)/256.f, 0.f, 1.f);
    if(use_transparency)
    {
      mi::Float32 t = (static_cast<mi::Float32>(i) + 0.5f)/256.f;
      colormap_entries[i].a = MIN_TRANS + (MAX_TRANS - MIN_TRANS)*t;
    }
  }

  // Set the the colormap values.
  colormap->set_colormap(&colormap_entries[0], colormap_entries.size());
  return colormap;
}

bool Create_path::create_scene(
  nv::index::IScene*                                    scene_edit,
  std::vector<mi::neuraylib::Tag>&                      path_tags,
  std::map< mi::neuraylib::Tag, Test_tag_info, Tag_comp >& test_tag_info_map,
  mi::neuraylib::IDice_transaction*                     dice_transaction) const
{
  check_success(dice_transaction != 0);
  check_success(scene_edit       != 0);

  {
```

```
  // Set the default light source
  mi::base::Handle<nv::index::IDirectional_light> light_global(
    scene_edit->create_attribute<nv::index::IDirectional_light>());
  check_success(light_global.is_valid_interface());
  light_global->set_direction(mi::math::Vector<mi::Float32, 3>(0.5f, 0.f, 1.f));
 const mi::neuraylib::Tag light_global_tag = dice_transaction->store_for_reference_counting(light
  check_success(light_global_tag.is_valid());
  scene_edit->append(light_global_tag, dice_transaction);

  // Set the default material
 mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
  check_success(phong_1.is_valid_interface());
  if(m_path_test_params.npr_mode)
  {
    phong_1->set_ambient(mi::math::Color(0.75f, 0.75f, 0.75f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.0f, 0.0f, 0.0f, 1.0f));
    phong_1->set_specular(mi::math::Color(0.0f, 0.0f, 0.0f, 1.0f));
    phong_1->set_shininess(0);
  }
  else
  {
    phong_1->set_ambient(mi::math::Color(0.15f, 0.15f, 0.15f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.95f, 0.95f, 0.95f, 1.0f));
    phong_1->set_specular(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    phong_1->set_shininess(85);
  }
 const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
  check_success(phong_1_tag.is_valid());
  scene_edit->append(phong_1_tag, dice_transaction);

  // The the colormap to demonstrate color index color mapping.
 mi::base::Handle<nv::index::IColormap> colormap(create_colormap(scene_edit, m_path_test_params.
  check_success(colormap.is_valid_interface());
 const mi::neuraylib::Tag colormap_tag = dice_transaction->store_for_reference_counting(colormap.
  check_success(colormap_tag.is_valid());
  scene_edit->append(colormap_tag, dice_transaction);
}

mi::math::Bbox< mi::Float32, 3 > path_bbox;

{
  const mi::Uint32 num_points = m_path_test_params.num_points;

  // get the color source style from parameters
  nv::index::IPath_style::Color_source color_source;
  if(m_path_test_params.use_colormap && m_path_test_params.use_rgba)
    color_source = nv::index::IPath_style::COLOR_SOURCE_BOTH;
  else if(m_path_test_params.use_colormap)
    color_source = nv::index::IPath_style::COLOR_SOURCE_COLORMAP_ONLY;
  else if(m_path_test_params.use_rgba)
    color_source = nv::index::IPath_style::COLOR_SOURCE_RGBA_ONLY;
  else
    color_source = nv::index::IPath_style::COLOR_SOURCE_NONE;


  //create rgba array. A grayscale color scale with a "V" shape
  std::vector<mi::math::Color_struct> colors;
```

```
  colors.resize(num_points);
  for(mi::Uint32 i=0; i<num_points; i++)
  {
  mi::Float32 g = (static_cast<mi::Float32>(i) + 0.5f)/static_cast<mi::Float32>(num_points);
    mi::Float32 f = fabsf(2.0f*g - 1.0f);
    colors[i].r = g;
    colors[i].g = 1.f - f;
    colors[i].b = 1.f - f;
    colors[i].a = 1.f;
  }

 // Set up the transformation matrix, define a translation and create and store the scene group
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.f);
  transform_mat.translate(mi::math::Vector<mi::Float32, 3>(120.f, 0.f, 0.f));
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());
  group_node->set_transform(transform_mat);

  // set up points
  std::vector< mi::math::Vector_struct<mi::Float32, 3> > points;
  points.resize(num_points);
  for(mi::Uint32 i=0; i<num_points; i++)
  {
  points[i].y = ((static_cast<mi::Float32>(i) + 0.5f)/static_cast<mi::Float32>(num_points))*6.f;
    points[i].x = evaluate_path(points[i].y);
    points[i].z = 1.f;

    path_bbox.insert(points[i]);

    points[i].x = points[i].x*100.f - 80.f;
    points[i].y = points[i].y*100.f + 10.f;
    points[i].z *= 100.f;

    path_bbox.insert(points[i]);
  }

  // set up colormap ids
  std::vector<mi::Uint32> colormap_ids;
  colormap_ids.resize(num_points);
  for(mi::Uint32 i=0; i<num_points; i++){
    colormap_ids[i] = (mi::Uint32) (255.f*((static_cast<mi::Float32>(i)+0.5f)/
                        static_cast<mi::Float32>(num_points)));
  }

  {
    //create a path style with linear interpolation
  mi::base::Handle<nv::index::IPath_style> path_style1(scene_edit->create_attribute<nv::index::I
    check_success(path_style1.is_valid_interface());
    path_style1->set_interpolation(nv::index::IPath_style::INTERPOLATION_LINEAR);
    path_style1->set_color_source(color_source);
  path_style1->set_upsampling(m_path_test_params.upsampling, m_path_test_params.up_factor, m_pat
  const mi::neuraylib::Tag path_style1_tag = dice_transaction->store_for_reference_counting(path_
    check_success(path_style1_tag.is_valid());
    group_node->append(path_style1_tag, dice_transaction);

  mi::base::Handle<nv::index::IPath_3D> path1(scene_edit->create_shape<nv::index::IPath_3D>());
```

```
  check_success(path1.is_valid_interface());

  // set radius
  path1->set_radius(15.f);

  // set points
  path1->set_points(&points[0], num_points);

  // set colormap ids
  path1->set_color_map_indexes(&colormap_ids[0], num_points);
  path1->set_colors(&colors[0], num_points);

  // append path to the group node
const mi::neuraylib::Tag path1_tag = dice_transaction->store_for_reference_counting(path1.get()
  check_success(path1_tag.is_valid());
  group_node->append(path1_tag, dice_transaction);

  path_tags.push_back(path1_tag);
test_tag_info_map[path1_tag] = Test_tag_info("IPath_3D", num_points); // for test purpose
}

{
  //create a path style with segment interpolation
mi::base::Handle<nv::index::IPath_style> path_style2(scene_edit->create_attribute<nv::index::I
  check_success(path_style2.is_valid_interface());
  path_style2->set_interpolation(nv::index::IPath_style::INTERPOLATION_SEGMENT);
  path_style2->set_color_source(color_source);
path_style2->set_upsampling(m_path_test_params.upsampling, m_path_test_params.up_factor, m_pat
  const mi::neuraylib::Tag path2_style_tag =
    dice_transaction->store_for_reference_counting(path_style2.get());
  check_success(path2_style_tag.is_valid());
  group_node->append(path2_style_tag, dice_transaction);

mi::base::Handle<nv::index::IPath_3D> path2(scene_edit->create_shape<nv::index::IPath_3D>());
  check_success(path2.is_valid_interface());

  // set radius
  path2->set_radius(12.0f);

  // set points
  for(mi::Uint32 i=0; i<num_points; i++){
    points[i].x += 200;
    path_bbox.insert(points[i]);
  }

  path2->set_points(&points[0], num_points);

  // set colormap ids
  path2->set_color_map_indexes(&colormap_ids[0], num_points);
  path2->set_colors(&colors[0], num_points);

  // append path to the group node
const mi::neuraylib::Tag path2_tag = dice_transaction->store_for_reference_counting(path2.get()
  check_success(path2_tag.is_valid());
  group_node->append(path2_tag, dice_transaction);

  path_tags.push_back(path2_tag);
```

```
test_tag_info_map[path2_tag] = Test_tag_info("IPath_3D", num_points); // for test purpose
}

{
  //create a path style with nearest interpolation
mi::base::Handle<nv::index::IPath_style> path_style3(scene_edit->create_attribute<nv::index::I
  check_success(path_style3.is_valid_interface());
  path_style3->set_interpolation(nv::index::IPath_style::INTERPOLATION_NEAREST);
  path_style3->set_color_source(color_source);
path_style3->set_upsampling(m_path_test_params.upsampling, m_path_test_params.up_factor, m_pat
const mi::neuraylib::Tag path3_style_tag = dice_transaction->store_for_reference_counting(path_
  check_success(path3_style_tag.is_valid());
  group_node->append(path3_style_tag, dice_transaction);

mi::base::Handle<nv::index::IPath_3D> path3(scene_edit->create_shape<nv::index::IPath_3D>());
  check_success(path3.is_valid_interface());

  // set radius
  path3->set_radius(7.0f);

  // set points
  for(mi::Uint32 i=0; i<num_points; i++){
    points[i].x += 200;
    path_bbox.insert(points[i]);
  }

  path3->set_points(&points[0], num_points);

  // set colormap ids
  path3->set_color_map_indexes(&colormap_ids[0], num_points);
  path3->set_colors(&colors[0], num_points);

  // append path to the group node
const mi::neuraylib::Tag path3_tag = dice_transaction->store_for_reference_counting(path3.get()
  check_success(path3_tag.is_valid());
  group_node->append(path3_tag, dice_transaction);

  path_tags.push_back(path3_tag);
test_tag_info_map[path3_tag] = Test_tag_info("IPath_3D", num_points); // for test purpose
}

if(m_path_test_params.test_transparency)
{
  // background plane
  mi::math::Vector<mi::Float32, 3> plane_point(-500.f, -100.f, 150.0f);
  mi::math::Vector<mi::Float32, 3> plane_normal(0.f, 0.f, 1.f);
  mi::math::Vector<mi::Float32, 3> plane_up(0.f, 1.f, 0.f);
  mi::math::Vector<mi::Float32, 2> plane_extent(1200.0f, 800.0f);

  mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
 scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
  check_success(tex_filter.is_valid_interface());
mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.
  check_success(tex_filter_tag.is_valid());
  group_node->append(tex_filter_tag, dice_transaction);

// Access an application layer component that provides some sample techniques such as the checkerb
```

```
    mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing>
      get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_proce
      check_success(processing.is_valid_interface());
    // Create a checkerboard technique and add it to the scene. For more details on how to implement th
    // source that was shipped with the application layer component 'nv::index::app::data_analysis_ar
      mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
        processing->get_sample_tool_set()->create_checker_board_2d_technique(
          plane_extent,
       nv::index::IDistributed_compute_destination_buffer_2d_texture::FORMAT_RGBA_FLOAT32,
          mi::math::Color(0.9f, 0.2f, 0.15f, 0.7f),
          mi::math::Color(0.2f, 0.2f, 0.8f,  1.0f)));
      check_success(mapping.is_valid_interface());

    mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.get())
      check_success(mapping_tag.is_valid());

      group_node->append(mapping_tag, dice_transaction);

    mi::base::Handle<nv::index::IPlane> plane(scene_edit->create_shape<nv::index::IPlane>());
      check_success(plane.is_valid_interface());
      plane->set_point(plane_point);
      plane->set_normal(plane_normal);
      plane->set_up(plane_up);
      plane->set_extent(plane_extent);

    mi::neuraylib::Tag plane_tag = dice_transaction->store_for_reference_counting(plane.get());
      check_success(plane_tag.is_valid());
      group_node->append(plane_tag, dice_transaction);

      path_tags.push_back(plane_tag);
      test_tag_info_map[plane_tag] = Test_tag_info("IPlane", -1); // for test purpose
    }

   const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_n
    check_success(group_node_tag.is_valid());
    scene_edit->append(group_node_tag, dice_transaction);

    INFO_LOG << "Bounding box of the vertices that create the paths: " << path_bbox;
  }

  return true;
}

void Create_path::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  const mi::math::Vector< mi::Float32, 3 > from( 234.0f, 604.0f,  500.0f);
  const mi::math::Vector< mi::Float32, 3 > to  ( 255.0f, 255.0f, -255.0f);
  const mi::math::Vector< mi::Float32, 3 > up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
```

```
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

void Create_path::setup_extreme_transformed_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // *** Camera:
  // index::camera::eye_point = 1808409.125 9981818 10948.5126953125
  // index::camera::view_direction = (1808409.125 9981818 -1500) - eye_point
  // index::camera::up_direction = 0 1 0
  // index::camera::aspect = 1.7152034261242
  // index::camera::aperture = 0.033
  // index::camera::focal = 0.03
  // index::camera::clip_min = 124.485130310059
  // index::camera::clip_max = 17585.455078125
  // index::canvas_resolution = 1602 934

  // Adjusted the camera position for p = 23+1.
 mi::math::Vector< mi::Float32, 3 > const from( 1805060.125f + 8192.0f, 9978520.0f + 16384.0f,  16384
 mi::math::Vector< mi::Float32, 3 > const to  ( 1805060.125f + 8192.0f, 9978520.0f + 8292.0f,  -256.0f
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.71520f); // not 1.7152034261242f, see IEEE754
  cam->set_focal(0.03f);
  cam->set_clip_min(124.485f); // not 124.485130310059f, see IEEE754
  cam->set_clip_max(17585.6f); // not 17585.55078125f,   see IEEE754

  INFO_LOG << "A camera set up using a large matrix.";
}

mi::math::Matrix<mi::Float32, 4, 4> Create_path::get_extreme_transformed_matrix() const
{
  // transform =
  //  [ 20.6100006103516   0                 0       1805060
  //    0                 20.6100006103516  0       9978520
  //    0                  0               -4       0
  //    0                  0                0       1       ]

  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    20.61f,      0.0f,         0.0f,  0.0f,
    0.0f,        20.61f,       0.0f,  0.0f,
    0.0f,        0.0f,        -4.0f,  0.0f,
    1805060.0f,  9978520.0f,   0.0f,  1.0f
    );
  return transform_mat;
}

nv::index::IFrame_results* Create_path::render_frame(const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());
```

```
  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("outfname", "frame_create_path");             // output file base name
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("supersampling", "0");                 // disable supersampling (default)
  sdict.insert("num_samples", "40");                           // 40 samples (default)
  sdict.insert("use_colormap", "1");                         // use color map (default)
 sdict.insert("use_rgba", "0");                       // don't use rgba array (default)
 sdict.insert("npr_mode", "0");                         // don't use npr mode (default)
 sdict.insert("test_transparency", "0");                // don't test transparency (default)
 sdict.insert("upsampling", "0");                      // don't use upsampling (default)
  sdict.insert("up_factor", "2");                              // 2x (default)
  sdict.insert("up_tension", "0.0");                           // 0.0 (default)
 sdict.insert("is_large_translate", "0");               // large translation mode (default 0)
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                // default: not call from make check.

  // DiCE settings
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io data_analysis_and_processing");
```

```
  // Initialize application
  Create_path create_path;
  create_path.initialize(argc, argv, sdict);
  check_success(create_path.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_path.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.13 create_plane.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iplane.h>
#include <nv/index/iscene.h>
#include <nv/index/iscene_group.h>
#include <nv/index/isession.h>
#include <nv/index/itexture_filter_mode.h>

#include "utility/canvas_utility.h"

#include <nv/index/app/idata_analysis_and_processing.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>

#include <iostream>
#include <sstream>

class Create_plane:
  public nv::index::app::Index_connect
{
public:
  Create_plane()
    :
    Index_connect(),
    m_is_unittest(false),
    m_supersampling(false),
    m_is_large_translate(false)
  {
    // INFO_LOG << "DEBUG: Create_plane() ctor";
  }

  virtual ~Create_plane()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_plane() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
```

```
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // create plane instances
  void create_planes(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction);

  // setup camera to see this example scene
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // setup a far away camera for a extreme transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  void setup_extreme_transformed_camera(nv::index::IPerspective_camera* cam) const;

  // setup a large scene transform for the extreme
  // transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  mi::math::Matrix<mi::Float32, 4, 4> get_extreme_transformed_matrix() const;

  // render a frame
  //
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                   m_is_unittest;
  std::string                                                         m_verify_image_fname;
  bool                                                                    m_supersampling;
```

```
  bool                                                    m_is_large_translate;
};

mi::Sint32 Create_plane::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    m_cluster_configuration =
      get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    const int max_retry = 3;
    for (int retry = 0; retry < max_retry; ++retry)
    {
      if (m_cluster_configuration->get_number_of_hosts() == 0)
      {
        INFO_LOG << "no host joined yet, retry "
            << (retry + 1) << "/" << max_retry;
        nv::index::app::util::time::sleep(0.3f);
      }
    }
    check_success(m_cluster_configuration->get_number_of_hosts() != 0);

    {
      // Obtain a DiCE transaction
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        // Setup session information
        m_session_tag = m_index_session->create_session(dice_transaction.get());
        check_success(m_session_tag.is_valid());

     mi::base::Handle<const nv::index::ISession> session(dice_transaction->access<nv::index::ISess
        check_success(session.is_valid_interface());

        mi::base::Handle< nv::index::IScene > scene_edit(
          dice_transaction->edit< nv::index::IScene >(session->get_scene()));
        check_success(scene_edit.is_valid_interface());

        // Enable supersampling
        if (m_supersampling)
        {
          INFO_LOG << "Enable supersampling.";

          mi::base::Handle<nv::index::IConfig_settings> config_settings(
        dice_transaction->edit<nv::index::IConfig_settings>(session->get_config()));
          check_success(config_settings.is_valid_interface());

          config_settings->set_rendering_samples(8);
```

```
}

//------------------------------------------------------------------------
// Scene setup: add textured planes
//------------------------------------------------------------------------
create_planes(scene_edit.get(), dice_transaction.get());

mi::base::Handle< nv::index::IPerspective_camera > cam(
  scene_edit->create_camera<nv::index::IPerspective_camera>());
check_success(cam.is_valid_interface());

mi::math::Vector<mi::Uint32, 2> buffer_resolution;
if (m_is_large_translate)
{
  INFO_LOG << "large translate test mode.";
  setup_extreme_transformed_camera(cam.get());
  buffer_resolution = mi::math::Vector<mi::Uint32, 2>(1602, 934);
}
else
{
  setup_camera(cam.get());
  buffer_resolution = mi::math::Vector<mi::Uint32, 2>(512, 512);
}
const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
check_success(camera_tag.is_valid());

m_image_file_canvas->set_resolution(buffer_resolution);

// Set up the scene and define the region of interest
const mi::math::Bbox_struct<mi::Float32, 3> xyz_roi_st = {
  { -1000.0f, -1000.0f, -1000.0f, },
  {  1000.0f,  1000.0f,  1000.0f, },
};

// set the region of interest
const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
check_success(xyz_roi.is_volume());
scene_edit->set_clipped_bounding_box(xyz_roi_st);

// Set the scene global transformation matrix.
// only change the coordinate system
mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
  1.0f,  0.0f,  0.0f,  0.0f,
  0.0f,  1.0f,  0.0f,  0.0f,
  0.0f,  0.0f, -1.0f,  0.0f,
  0.0f,  0.0f,  0.0f,  1.0f
  );

if(m_is_large_translate)
{
  transform_mat = get_extreme_transformed_matrix();
}

scene_edit->set_transform_matrix(transform_mat);

// Set the current camera to the scene.
check_success(camera_tag.is_valid());
```

```
        scene_edit->set_camera(camera_tag);
      }
      // Commit the transaction used for initializing
      dice_transaction->commit();
    }

    // Rendering
    {
      // Render the frame to a file
      const mi::Sint32  frame_idx = 0;
      const std::string fname = get_output_file_name(m_outfname, frame_idx);
     mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      if (err_set->any_errors())
      {
        std::ostringstream os;
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        exit_code = 1;
      }

      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), m_verify_image_fname, get_options())))
      {
        exit_code = 1;
      }
    }
  }

  return exit_code;
}

bool Create_plane::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  // set own options
```

```
  m_outfname           = sdict.get("outfname");
  m_verify_image_fname = sdict.get("verify_image_fname");
 m_supersampling    = nv::index::app::get_bool(sdict.get("supersampling", "false"));
 m_is_large_translate = nv::index::app::get_bool(sdict.get("is_large_translate", "false"));

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["       + sdict.get("outfname") +
       "], dice::verbose = " + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-supersampling bool]\n"
      << "            when true then supersampling is enabled."
      << "(default: " << m_supersampling << ")\n"

      << "        [-is_large_translate bool]\n"
      << "            on/off large translation mode."
      << "(default: " << m_is_large_translate << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode. "
      << m_is_unittest << "])"
      << std::endl;
    exit(1);
  }
  return true;
}

void Create_plane::create_planes(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(scene_edit != 0);

  // Add a scene group where the shapes should be added
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());
```

```
// Add a light and a material
{
  // Add a light
  mi::base::Handle<nv::index::IDirectional_headlight> headlight(
    scene_edit->create_attribute<nv::index::IDirectional_headlight>());
  check_success(headlight.is_valid_interface());
  const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
  headlight->set_intensity(color_intensity);
  headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
 const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
  check_success(headlight_tag.is_valid());
  group_node->append(headlight_tag, dice_transaction);

  // Add a fully ambient material for the planes, so that lighting doesn't matter
  mi::base::Handle<nv::index::IPhong_gl> phong_1(
    scene_edit->create_attribute<nv::index::IPhong_gl>());
  check_success(phong_1.is_valid_interface());
  phong_1->set_ambient(mi::math::Color(1.0f, 1.0f, 1.0f));
  phong_1->set_diffuse(mi::math::Color(0.0f));
  phong_1->set_specular(mi::math::Color(0.0f));

 mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.get());
  check_success(phong_1_tag.is_valid());
  group_node->append(phong_1_tag, dice_transaction);
}

//
// Plane 1: RGBA mode, axis aligned, no filtering
//
{
  // Place plane centered in the origin, normal pointing along the z-axis
  mi::math::Vector<mi::Float32, 3> plane_point(-400.f, -250.f, 1.0f);
  mi::math::Vector<mi::Float32, 3> plane_normal(0.f, 0.f, 1.f);
  mi::math::Vector<mi::Float32, 3> plane_up(0.f, 1.f, 0.f);
  mi::math::Vector<mi::Float32, 2> plane_extent(800.0f, 500.0f);

  mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
  scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
  check_success(tex_filter.is_valid_interface());
 mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.ge
  check_success(tex_filter_tag.is_valid());
  group_node->append(tex_filter_tag, dice_transaction);

 // Access an application layer component that provides some sample techniques such as the mandelbro
 mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> p
  get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_proces
  check_success(processing.is_valid_interface());
 // Create a mandelbrot technique and add it to the scene. For more details on how to implement the ma
 // source that was shipped with the application layer component 'nv::index::app::data_analysis_and
  mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
    processing->get_sample_tool_set()->create_mandelbrot_2d_technique(
   plane_extent, nv::index::IDistributed_compute_destination_buffer_2d_texture::FORMAT_RGBA_FLO

 mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.get());
  check_success(mapping_tag.is_valid());
  group_node->append(mapping_tag, dice_transaction);
```

```
  mi::base::Handle<nv::index::IPlane> plane(
    scene_edit->create_shape<nv::index::IPlane>());
  check_success(plane.is_valid_interface());
  plane->set_point(plane_point);
  plane->set_normal(plane_normal);
  plane->set_up(plane_up);
  plane->set_extent(plane_extent);

 mi::neuraylib::Tag plane_tag = dice_transaction->store_for_reference_counting(plane.get());
  check_success(plane_tag.is_valid());
  group_node->append(plane_tag, dice_transaction);
}

//
// Plane 2: Color map mode, arbitrary orientation, use texture filtering
//
{
  mi::math::Vector<mi::Float32, 3> plane_point(-50.0f, 50.0f, 700.0f);
  mi::math::Vector<mi::Float32, 3> plane_normal(1.f, 1.f, 1.f);
  mi::math::Vector<mi::Float32, 3> plane_up(-1.f, 1.f, 0.f);
  mi::math::Vector<mi::Float32, 2> plane_extent(400.0f, 400.0f);

  const bool enable_texture_filtering = true;
 const mi::Sint32 colormap_entry_id = 40; // same as demo application's colormap file 40
  mi::neuraylib::Tag colormap_tag =
    create_colormap(colormap_entry_id, scene_edit, dice_transaction);
  check_success(colormap_tag.is_valid());
  group_node->append(colormap_tag, dice_transaction);

  mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter;

  if (enable_texture_filtering)
  {
  tex_filter = scene_edit->create_attribute<nv::index::ITexture_filter_mode_linear>();
  }
  else
  {
  tex_filter = scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>();
  }
  check_success(tex_filter.is_valid_interface());
 mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.ge
  check_success(tex_filter_tag.is_valid());
  group_node->append(tex_filter_tag, dice_transaction);

 // Access an application layer component that provides some sample techniques such as the mandelbro
 mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> p
  get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_proces
  check_success(processing.is_valid_interface());
 // Create a mandelbrot technique and add it to the scene. For more details on how to implement the ma
 // source that was shipped with the application layer component 'nv::index::app::data_analysis_and
  mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
    processing->get_sample_tool_set()->create_mandelbrot_2d_technique(
   plane_extent, nv::index::IDistributed_compute_destination_buffer_2d_texture::FORMAT_SCALAR_U
  check_success(mapping.is_valid_interface());
 mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.get());
  check_success(mapping_tag.is_valid());
  group_node->append(mapping_tag, dice_transaction);
```

```
    mi::base::Handle<nv::index::IPlane> plane(
      scene_edit->create_shape<nv::index::IPlane>());
    check_success(plane.is_valid_interface());
    plane->set_point(plane_point);
    plane->set_normal(plane_normal);
    plane->set_up(plane_up);
    plane->set_extent(plane_extent);

   mi::neuraylib::Tag plane_tag = dice_transaction->store_for_reference_counting(plane.get());
    check_success(plane_tag.is_valid());
    group_node->append(plane_tag, dice_transaction);
  }

  // Finally append everything to the root of the hierachical scene description
 mi::neuraylib::Tag group_tag = dice_transaction->store_for_reference_counting(group_node.get());
  check_success(group_tag.is_valid());
  scene_edit->append(group_tag, dice_transaction);
}

void Create_plane::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene.
  const mi::math::Vector<mi::Float32, 3> from(100.0f, 0.0f, -1000.0f);
  const mi::math::Vector<mi::Float32, 3> to  (0.0f, 250.0f, 0.0f);
  const mi::math::Vector<mi::Float32, 3> up  (0.0f, 1.0f, 0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(80.0f);
  cam->set_clip_max(5000.0f);
}

void Create_plane::setup_extreme_transformed_camera(
  nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);
  // *** Camera:
  // index::camera::eye_point = 1808409.125 9981818 10948.5126953125
  // index::camera::view_direction = (1808409.125 9981818 -1500) - eye_point
  // index::camera::up_direction = 0 1 0
  // index::camera::aspect = 1.7152034261242
  // index::camera::aperture = 0.033
  // index::camera::focal = 0.03
  // index::camera::clip_min = 124.485130310059
  // index::camera::clip_max = 17585.455078125
  // index::canvas_resolution = 1602 934

  // Adjusted the camera position for p = 23+1.
 mi::math::Vector< mi::Float32, 3 > const from( 1805060.125f + 5120.0f, 9978520.0f + 0.0f,   -30720.0
 mi::math::Vector< mi::Float32, 3 > const to  ( 1805060.125f + 0.0f,   9978520.0f + 2560.0f,    0.0f)
```

```
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.71520f); // not 1.7152034261242f, see IEEE754
  cam->set_focal(0.03f);
  cam->set_clip_min(124.485f); // not 124.485130310059f, see IEEE754
  cam->set_clip_max(17585.6f); // not 17585.55078125f,   see IEEE754

  INFO_LOG << "Set camera extreme mode";
}

mi::math::Matrix<mi::Float32, 4, 4> Create_plane::get_extreme_transformed_matrix() const
{
  // transform =
  //  [ 20.6100006103516   0                0       1805060
  //    0                 20.6100006103516  0       9978520
  //    0                  0               -4       0
  //    0                  0                0       1       ]

  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    20.0f,       0.0f,       0.0f,  0.0f,
    0.0f,        20.0f,      0.0f,  0.0f,
    0.0f,        0.0f,      -20.0f,  0.0f,
    1805060.0f,  9978520.0f, 0.0f,  1.0f
    );
  return transform_mat;
}

nv::index::IFrame_results* Create_plane::render_frame(const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
```

```
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("outfname", "frame_create_plane");             // output file base name
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("supersampling", "0");                  // disable supersampling (default)
 sdict.insert("is_large_translate", "0");              // large translation mode (default 0)
 sdict.insert("is_dump_comparison_image_when_failed", "1");  // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");               // default: not call from make check.

  // dice setting
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io data_analysis_and_processing");

  // Initialize application
  Create_plane create_plane;
  create_plane.initialize(argc, argv, sdict);
  check_success(create_plane.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_plane.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.14 create_point_set.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/ipoint_set.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Create_point_set:
  public nv::index::app::Index_connect
{
public:
  Create_point_set()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_point_set() ctor";
  }

  virtual ~Create_point_set()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_point_set() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
```

```
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // attribute to color map example
  // \param[in] attrib attribute value
  // \return color defined by an attribute
  mi::math::Color_struct get_color_st_from_attribute(mi::Sint32 attrib) const;

  // attribute to radius map example
  // \param[in] attrib attribute value
  // \return a radius corresponding to the attribute value
  mi::Float32 get_radius_from_attribute(mi::Sint32 attrib) const;

  // create point set in the scene.
  // \param[in] scene_edit       IScene for scene editing
  // \param[in] dice_transaction dice transaction
  // \return true when success
  bool create_point_set(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  // setup camera to see this example scene
  // \param[in] cam    a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // render a frame
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
 mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
 mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
 mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
 std::string                                                              m_outfname;
 bool                                                                   m_is_unittest;
 std::string                                                         m_verify_image_fname;
};

mi::Sint32 Create_point_set::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    m_cluster_configuration =
```

```
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  {
    // DiCE database access
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());
    {
      // Setup session information
      m_session_tag =
        m_index_session->create_session(dice_transaction.get());
      check_success(m_session_tag.is_valid());
      mi::base::Handle< nv::index::ISession const > session(
        dice_transaction->access< nv::index::ISession const >(
          m_session_tag));
      check_success(session.is_valid_interface());

      mi::base::Handle< nv::index::IScene > scene_edit(
        dice_transaction->edit< nv::index::IScene >(session->get_scene()));
      check_success(scene_edit.is_valid_interface());

      //------------------------------------------------------------------------
      // Scene setup: add point set shape, scene parameters, camera.
      //------------------------------------------------------------------------
      // Add an point set shape to the scene
      check_success(create_point_set(scene_edit.get(), dice_transaction.get()));

      // Create and edit a camera. Data distribution is based on
      // the camera. (Because only visible massive data are
      // considered)
      mi::base::Handle< nv::index::IPerspective_camera > cam(
        scene_edit->create_camera<nv::index::IPerspective_camera>());
      check_success(cam.is_valid_interface());
      setup_camera(cam.get());
      const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
      check_success(camera_tag.is_valid());

      const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
      m_image_file_canvas->set_resolution(buffer_resolution);

      // Set up the scene
      mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
        {   0.0f,   0.0f,   0.0f, },
        { 500.0f, 500.0f, 500.0f, },
      };

      // set the region of interest
      const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
```

```
        check_success(xyz_roi.is_volume());
        scene_edit->set_clipped_bounding_box(xyz_roi_st);

        // Set the scene global transformation matrix.
        // only change the coordinate system
        mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
          1.0f,  0.0f,  0.0f,  0.0f,
          0.0f,  1.0f,  0.0f,  0.0f,
          0.0f,  0.0f, -1.0f,  0.0f,
          0.0f,  0.0f,  0.0f,  1.0f
          );
        scene_edit->set_transform_matrix(transform_mat);

        // Set the current camera to the scene.
        check_success(camera_tag.is_valid());
        scene_edit->set_camera(camera_tag);
      }
      dice_transaction->commit();
    }

    // Rendering
    {
      // Render a frame and save the rendered image to a file.
      const mi::Sint32  frame_idx = 0;
      const std::string fname = get_output_file_name(m_outfname, frame_idx);
     mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      if (err_set->any_errors())
      {
        std::ostringstream os;
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        exit_code = 1;
      }


      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), m_verify_image_fname, get_options())))
      {
        exit_code = 1;
      }
    }
  }

  return exit_code;
}

bool Create_point_set::evaluate_options(nv::index::app::String_dict& sdict)
```

```
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname = sdict.get("outfname", "");
  m_verify_image_fname = sdict.get("verify_image_fname", "");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

mi::math::Color_struct Create_point_set::get_color_st_from_attribute(mi::Sint32 attrib) const
{
  const mi::Sint32 intval = abs(attrib);
  mi::math::Color_struct col;
  const mi::Float32 coef = 1.0f / 15.0f;
```

```
  col.r = coef * static_cast< mi::Float32 >(intval & 15);
  col.g = coef * static_cast< mi::Float32 >((intval >> 4) & 15);
  col.b = coef * static_cast< mi::Float32 >((intval >> 8) & 15);
  col.a = 1.0;

  return col;
}

mi::Float32 Create_point_set::get_radius_from_attribute(mi::Sint32 attrib) const
{
  const mi::Float32 rad = static_cast< mi::Float32 >(abs(attrib) % 12);
  return rad;
}

bool Create_point_set::create_point_set(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(scene_edit != 0);
  check_success(dice_transaction != 0);

  // hierarchical scene description node for the point set
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Add a light and a material
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
    headlight->set_intensity(color_intensity);
    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());
    group_node->append(headlight_tag, dice_transaction);

    // add material for 3D points shape (the material is only effective for 3D shape)
   mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(mi::math::Color(0.3f, 0.3f, 0.3f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    phong_1->set_specular(mi::math::Color(0.4f));
    phong_1->set_shininess(100.f);
   const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
    check_success(phong_1_tag.is_valid());
    group_node->append(phong_1_tag, dice_transaction);
  }

  // Point coordinates and its attributes. According to the
  // attributes, color and radius are defined.
  // Here, we create two point sets.
  std::vector< mi::math::Vector_struct< mi::Float32, 3> > point_pos_vec[2];
  std::vector< mi::math::Color_struct >                   point_col_vec[2];
  std::vector< mi::Float32 >                              point_rad_vec[2];
```

```
  // square shaped positions
  const mi::Sint32  column_count= 20;
  const mi::Sint32  point_count = 200;
  const mi::Float32 x_mag = 25.0;
  const mi::Float32 y_mag = 25.0;
  const mi::Float32 z     = 30.0;
  const mi::Float32 y_offset[2] = { 0.0f, 280.0f };
  for(mi::Sint32 i = 0; i < point_count; ++i){
    mi::math::Vector_struct< mi::Float32, 3> pos[2];
    for(mi::Sint32 j = 0; j < 2; ++j){
      pos[j].x = static_cast< mi::Float32 >(i % column_count) * x_mag;
      pos[j].y = static_cast< mi::Float32 >(i / column_count) * y_mag + y_offset[j];
      pos[j].z = z;
      point_pos_vec[j].push_back(pos[j]);
    point_col_vec[j].push_back(get_color_st_from_attribute(i)); // calculate the same value twice he
    point_rad_vec[j].push_back(get_radius_from_attribute(i));  // but this overhead is small in this
    }
  }

  // Create two point sets
  for(mi::Sint32 j = 0; j < 2; ++j){

    // Create attribute_point_set scene element and add it to the scene
    nv::index::IPoint_set::Point_style style =
    (j==0) ? nv::index::IPoint_set::FLAT_CIRCLE : nv::index::IPoint_set::SHADED_CIRCLE;

   mi::base::Handle<nv::index::IPoint_set> point_set(scene_edit->create_shape<nv::index::IPoint_se
    check_success(point_set.is_valid_interface());
    point_set->set_point_style(style);
    point_set->set_vertices(&point_pos_vec[j][0], point_pos_vec[j].size());
    point_set->set_colors(  &point_col_vec[j][0], point_col_vec[j].size());
    point_set->set_radii(   &point_rad_vec[j][0], point_rad_vec[j].size());

    // Add the points to the database
   const mi::neuraylib::Tag point_set_scene_element_tag = dice_transaction->store_for_reference_cou
    // if you are not registered Attribute_point_set, the next check fails.
    check_success(point_set_scene_element_tag.is_valid());
    // Add to the scene description
    group_node->append(point_set_scene_element_tag, dice_transaction);
    std::stringstream sstr;
    sstr << "Added point_set (size: " << point_count << ") to the scene (tag id: "
      << point_set_scene_element_tag.id << ").";
    INFO_LOG << sstr.str();
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  return true;
}

void Create_point_set::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);
```

```
  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from( 254.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_point_set::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("outfname", "frame_create_point_set");        // output file base name
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                    // default: not call from make check.

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");
```

```
  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // Initialize application
  Create_point_set create_point_set;
  create_point_set.initialize(argc, argv, sdict);
  check_success(create_point_set.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_point_set.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.15 create_polygons.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/iindex.h>
#include <nv/index/isession.h>
#include <nv/index/iscene.h>
#include <nv/index/icamera.h>
#include <nv/index/ipolygon.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Create_polygons:
  public nv::index::app::Index_connect
{
public:
  Create_polygons()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_polygons() ctor";
  }

  virtual ~Create_polygons()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_polygons() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
```

```
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Create polygons in the scene.
  bool create_polygons(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  // setup camera to see this example scene
  //
  // \param[in] cam    a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // render a frame
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                        m_verify_image_fname;
};

mi::Sint32 Create_polygons::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    m_cluster_configuration =
      get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    // This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));

    {
      // DiCE database access
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
```

```
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    // Setup session information
    m_session_tag =
      m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle< nv::index::ISession const > session(
      dice_transaction->access< nv::index::ISession const >(
        m_session_tag));
    check_success(session.is_valid_interface());

    mi::base::Handle< nv::index::IScene > scene_edit(
      dice_transaction->edit< nv::index::IScene >(session->get_scene()));
    check_success(scene_edit.is_valid_interface());

    //------------------------------------------------------------------------
    // Scene setup: add polygon shape, scene parameters, camera.
    //------------------------------------------------------------------------
    // Add polygon shapes to the scene
    check_success(create_polygons(scene_edit.get(), dice_transaction.get()));

    // Create and edit a camera. Data distribution is based on
    // the camera. (Because only visible massive data are
    // considered)
    mi::base::Handle< nv::index::IPerspective_camera > cam(
      scene_edit->create_camera<nv::index::IPerspective_camera>());
    check_success(cam.is_valid_interface());
    setup_camera(cam.get());
    const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
    check_success(camera_tag.is_valid());

    const mi::math::Vector<mi::Uint32, 2> buffer_resolution(1025, 1024);
    m_image_file_canvas->set_resolution(buffer_resolution);

    // Set up the scene
    mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
      {   0.0f,   0.0f,   0.0f, },
      { 500.0f, 500.0f, 500.0f, },
    };

    // set the region of interest
    const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
    check_success(xyz_roi.is_volume());
    scene_edit->set_clipped_bounding_box(xyz_roi_st);

    // Set the scene global transformation matrix.
    // only change the coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f,
      0.0f,  0.0f,  0.0f,  1.0f
      );
    scene_edit->set_transform_matrix(transform_mat);

    // Set the current camera to the scene.
```

```
        check_success(camera_tag.is_valid());
        scene_edit->set_camera(camera_tag);
      }
      dice_transaction->commit();
    }

    // Rendering
    {
      const std::string fname = get_output_file_name(m_outfname, 0);
     mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      if (err_set->any_errors())
      {
        std::ostringstream os;
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
            << os.str();
        exit_code = 1;
      }


      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), m_verify_image_fname, get_options())))
      {
        exit_code = 1;
      }
    }
  }

  return exit_code;
}

bool Create_polygons::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }


  // Set own options
```

```
  m_outfname = sdict.get("outfname", "");
  m_verify_image_fname = sdict.get("verify_image_fname", "");

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << sdict.get("outfname") << "])\n"
      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")\n"

      << "        [-is_dump_comparison_image_when_failed bool]\n"
    << "         when true, dump the comparison images and generate the diff image for debug.\n"
      << "            You can also force to dump and set the filename.\n"
    << "         (default: " << sdict.get("is_dump_comparison_image_when_failed") << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

bool Create_polygons::create_polygons(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // create a pentagone
  {
   mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
    check_success(poly.is_valid_interface());
```

```
    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = 40.f;
    pos.y = 70.f;
    pos.z = 30.f;

    mi::Uint32 num_vertices = 5;
    mi::math::Vector_struct< mi::Float32, 2> vertices[5];

   const mi::Float32 delta_phi = (2.0f*static_cast<mi::Float32>(M_PI))/static_cast<mi::Float32>(num
    const mi::Float32 radius = 120.f;

    for (mi::Uint32 i=0; i<num_vertices; i++)
    {
      mi::math::sincos(delta_phi*i, vertices[i].x, vertices[i].y);
      vertices[i].x *= radius;
      vertices[i].y *= radius;
    }

    // create polygon geometry
    if (poly->set_geometry(vertices, num_vertices, pos))
    {
      INFO_LOG << "Pentagone created";

      mi::math::Color_struct fill_color = {0.5f, 1.0f, 0.f, 1.f};

      poly->set_fill_style(fill_color, nv::index::IPolygon::FILL_SOLID);

    const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
      check_success(poly_tag.is_valid());
      group_node->append(poly_tag, dice_transaction);
    }
    else
    {
      INFO_LOG << "Pentagone not created, bad shaped";
    }
  }

  // create a star
  {
   mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
    check_success(poly.is_valid_interface());
    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = 250.f;
    pos.y = 70.f;
    pos.z = 30.f;

    mi::Uint32 num_vertices = 5;
    mi::math::Vector_struct< mi::Float32, 2> vertices[10];

   const mi::Float32 delta_phi = (2.0f * static_cast<mi::Float32>(M_PI)) / static_cast<mi::Float32 >(
    const mi::Float32 radius = 120.f;
    const mi::Float32 inner_radius = 80.f;

    for (mi::Uint32 i=0; i<num_vertices; i++)
    {
      mi::math::sincos(delta_phi*i, vertices[2*i].x, vertices[2*i].y);
      vertices[2*i].x *= radius;
```

```
      vertices[2*i].y *= radius;

     mi::math::sincos(delta_phi*(i+0.5f), vertices[2*i + 1].x, vertices[2*i + 1].y);
      vertices[2*i + 1].x *= inner_radius;
      vertices[2*i + 1].y *= inner_radius;
    }

    // create polygon geometry
    if (poly->set_geometry(vertices, 2*num_vertices, pos))
    {
      INFO_LOG << "Star created";

      mi::math::Color_struct fill_color = {0.39f, 0.58f, 0.93f, 1.f};

      poly->set_fill_style(fill_color, nv::index::IPolygon::FILL_SOLID);

    const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
      check_success(poly_tag.is_valid());
      group_node->append(poly_tag, dice_transaction);
    }
    else
    {
      INFO_LOG << "Star not created, bad shaped";
    }
  }

  // create a 8 spikes star
  {
   mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
    check_success(poly.is_valid_interface());
    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = 460.f;
    pos.y = 70.f;
    pos.z = 30.f;

    mi::Uint32 num_vertices = 8;
    mi::math::Vector_struct< mi::Float32, 2> vertices[16];

   const mi::Float32 delta_phi = (2.0f* static_cast<mi::Float32>(M_PI)) / static_cast<mi::Float32>(n
    const mi::Float32 radius = 120.f;
    const mi::Float32 inner_radius = 60.f;

    for (mi::Uint32 i=0; i<num_vertices; i++)
    {
      mi::math::sincos(delta_phi*i, vertices[2*i].x, vertices[2*i].y);
      vertices[2*i].x *= radius;
      vertices[2*i].y *= radius;

     mi::math::sincos(delta_phi*(i+0.5f), vertices[2*i + 1].x, vertices[2*i + 1].y);
      vertices[2*i + 1].x *= inner_radius;
      vertices[2*i + 1].y *= inner_radius;
    }

    // create polygon geometry
    if (poly->set_geometry(vertices, 2*num_vertices, pos))
    {
      INFO_LOG << "Star created";
```

```
    mi::math::Color_struct fill_color = {0.39f, 0.58f, 0.93f, 1.f};

    poly->set_fill_style(fill_color, nv::index::IPolygon::FILL_SOLID);

  const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
    check_success(poly_tag.is_valid());
    group_node->append(poly_tag, dice_transaction);
  }
  else
  {
    INFO_LOG << "Star not created, bad shaped";
  }
}

// create a free shape: Cross
{
 mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
  check_success(poly.is_valid_interface());
  mi::math::Vector_struct< mi::Float32, 3> pos;
  pos.x = 40.f;
  pos.y = 280.f;
  pos.z = 30.f;

  mi::Uint32 num_vertices = 12;
  mi::math::Vector_struct< mi::Float32, 2> vertices[12];

  vertices[0].x = -60.f;
  vertices[0].y = 120.f;
  vertices[1].x = -60.f;
  vertices[1].y = 60.f;
  vertices[2].x = -120.f;
  vertices[2].y = 60.f;
  vertices[3].x = -120.f;
  vertices[3].y = -30.f;
  vertices[4].x = -60.f;
  vertices[4].y = -30.f;
  vertices[5].x = -60.f;
  vertices[5].y = -120.f;
  vertices[6].x = 60.f;
  vertices[6].y = -120.f;
  vertices[7].x = 60.f;
  vertices[7].y = -30.f;
  vertices[8].x = 120.f;
  vertices[8].y = -30.f;
  vertices[9].x = 120.f;
  vertices[9].y = 60.f;
  vertices[10].x = 60.f;
  vertices[10].y = 60.f;
  vertices[11].x = 60.f;
  vertices[11].y = 120.f;

  // create polygon geometry
  if (poly->set_geometry(vertices, num_vertices, pos))
  {
    INFO_LOG << "Cross created";
```

```
      mi::math::Color_struct fill_color = {0.8f, 0.8f, 0.8f, 1.f};

      poly->set_fill_style(fill_color, nv::index::IPolygon::FILL_SOLID);

    const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
      check_success(poly_tag.is_valid());
      group_node->append(poly_tag, dice_transaction);
    }
    else
    {
      INFO_LOG << "Cross not created, bad shaped";
    }
  }

  // create a free shape: N
  {
   mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
    check_success(poly.is_valid_interface());
    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = 250.f;
    pos.y = 280.f;
    pos.z = 30.f;

    mi::Uint32 num_vertices = 10;
    mi::math::Vector_struct< mi::Float32, 2> vertices[10];

    vertices[0].x = -100.f;
    vertices[0].y = -100.f;
    vertices[1].x = -20.f;
    vertices[1].y = -100.f;
    vertices[2].x = -20.f;
    vertices[2].y = -20.f;
    vertices[3].x = 20.f;
    vertices[3].y = -100.f;
    vertices[4].x = 100.f;
    vertices[4].y = -100.f;
    vertices[5].x = 100.f;
    vertices[5].y = 100.f;
    vertices[6].x = 20.f;
    vertices[6].y = 100.f;
    vertices[7].x = 20.f;
    vertices[7].y = 20.f;
    vertices[8].x = -20.f;
    vertices[8].y = 100.f;
    vertices[9].x = -100.f;
    vertices[9].y = 100.f;

    // create polygon geometry
    if (poly->set_geometry(vertices, num_vertices, pos))
    {
      INFO_LOG << "N created";

      mi::math::Color_struct fill_color = {0.0f, 1.0f, 0.f, 1.f};

      poly->set_fill_style(fill_color, nv::index::IPolygon::FILL_SOLID);

    const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
```

```
    check_success(poly_tag.is_valid());
    group_node->append(poly_tag, dice_transaction);
  }
  else
  {
    INFO_LOG << "N not created, bad shaped";
  }
}

// create a free shape: Flash
{
 mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
  check_success(poly.is_valid_interface());
  mi::math::Vector_struct< mi::Float32, 3> pos;
  pos.x = 460.f;
  pos.y = 280.f;
  pos.z = 30.f;

  mi::Uint32 num_vertices = 6;
  mi::math::Vector_struct< mi::Float32, 2> vertices[6];

  vertices[0].x = 0.f;
  vertices[0].y = 0.f;
  vertices[1].x = 0.f;
  vertices[1].y = 40.f;
  vertices[2].x = -80.f;
  vertices[2].y = -40.f;
  vertices[3].x = -20.f;
  vertices[3].y = 0.f;
  vertices[4].x = 0.f;
  vertices[4].y = -40.f;
  vertices[5].x = 80.f;
  vertices[5].y = 60.f;

  // create polygon geometry
  if (poly->set_geometry(vertices, num_vertices, pos))
  {
    INFO_LOG << "Flash created";

    mi::math::Color_struct fill_color = {0.0f, 1.0f, 0.f, 1.f};

    poly->set_fill_style(fill_color, nv::index::IPolygon::FILL_SOLID);

  const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
    check_success(poly_tag.is_valid());
    group_node->append(poly_tag, dice_transaction);
  }
  else
  {
    INFO_LOG << "Flash not created, bad shaped";
  }
}

// create a free shape: irregular shape
{
 mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
  check_success(poly.is_valid_interface());
```

```
    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = 240.f;
    pos.y = 430.f;
    pos.z = 30.f;

    mi::Uint32 num_vertices = 26;
    mi::math::Vector_struct< mi::Float32, 2> vertices[26];

    vertices[0].x = 0.f;
    vertices[0].y = -40.f;
    vertices[1].x = 120.f;
    vertices[1].y = -60.f;
    vertices[2].x = 120.f;
    vertices[2].y = -20.f;
    vertices[3].x = 100.f;
    vertices[3].y = -20.f;
    vertices[4].x = 120.f;
    vertices[4].y = 40.f;
    vertices[5].x = 80.f;
    vertices[5].y = 40.f;
    vertices[6].x = 60.f;
    vertices[6].y = -20.f;
    vertices[7].x = 40.f;
    vertices[7].y = -20.f;
    vertices[8].x = 50.f;
    vertices[8].y = 140.f;
    vertices[9].x = 20.f;
    vertices[9].y = 160.f;
    vertices[10].x = -10.f;
    vertices[10].y = 140.f;
    vertices[11].x = 0.f;
    vertices[11].y = 40.f;
    vertices[12].x = -40.f;
    vertices[12].y = 50.f;
    vertices[13].x = -60.f;
    vertices[13].y = 20.f;
    vertices[14].x = -60.f;
    vertices[14].y = -20.f;
    vertices[15].x = -80.f;
    vertices[15].y = -20.f;
    vertices[16].x = -80.f;
    vertices[16].y = 40.f;
    vertices[17].x = -100.f;
    vertices[17].y = 50.f;
    vertices[18].x = -100.f;
    vertices[18].y = -20.f;
    vertices[19].x = -120.f;
    vertices[19].y = -20.f;
    vertices[20].x = -140.f;
    vertices[20].y = -20.f;
    vertices[21].x = -120.f;
    vertices[21].y = 80.f;
    vertices[22].x = -180.f;
    vertices[22].y = 80.f;
    vertices[23].x = -160.f;
    vertices[23].y = -20.f;
    vertices[24].x = -180.f;
```

```
    vertices[24].y = -20.f;
    vertices[25].x = -180.f;
    vertices[25].y = -40.f;


    // create polygon geometry
    if (poly->set_geometry(vertices, num_vertices, pos))
    {
      INFO_LOG << "Irregular shape created";

      mi::math::Color_struct fill_color = {1.0f, 0.25f, 0.f, 1.f};

      poly->set_fill_style(fill_color, nv::index::IPolygon::FILL_SOLID);

    const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
      check_success(poly_tag.is_valid());
      group_node->append(poly_tag, dice_transaction);
    }
    else
    {
      INFO_LOG << "Irregular shape not created, bad shaped";
    }
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  return true;
}

void Create_polygons::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from( 254.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Create_polygons::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());
```

```
  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("outfname", "frame_create_polygons");         // output file base name
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // Initialize application
  Create_polygons create_polygons;
  create_polygons.initialize(argc, argv, sdict);
  check_success(create_polygons.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_polygons.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.16 create_stylized_points_and_lines.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/iline_set.h>
#include <nv/index/imaterial.h>
#include <nv/index/ipoint_set.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Create_stylized_point_and_lines:
  public nv::index::app::Index_connect
{
public:
  Create_stylized_point_and_lines()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_stylized_point_and_lines() ctor";
  }

  virtual ~Create_stylized_point_and_lines()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_stylized_point_and_lines() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);
```

```
    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Create point set in the scene.
  //
  // \param[in] scene_edit       IScene for the scene edit.
  // \param[in] pick_line_style  pick line style
  // \param[in] pick_point_style pick point style
  // \param[in] transparency     true when transparency is on.
  // \param[in] dice_transaction dice transaction
  // \return true when success
  bool create_stylized_lines_and_points(
    nv::index::IScene*                scene_edit,
    mi::Uint32                        pick_line_style,
    mi::Uint32                        pick_point_style,
    bool                              transparency,
    mi::neuraylib::IDice_transaction* dice_transaction);

  // setup camera to see this example scene
  // \param[in] cam    a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // render a frame
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
 mi::neuraylib::Tag                                                  m_session_tag;
  // NVIDIA IndeX cluster configuration
 mi::base::Handle<nv::index::ICluster_configuration>                    m_cluster_configuration;
  // Application layer image file canvas (a render target)
 mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                           m_outfname;
  bool                                                                m_is_unittest;
  std::string                                                      m_verify_image_fname;
  mi::Uint32                                                           m_line_style;
  mi::Uint32                                                          m_point_style;
  bool                                                               m_transparency;
};

mi::Sint32 Create_stylized_point_and_lines::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
```

```
{
  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));
  {
    // DiCE database access
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());
    {
      // Setup session information
      m_session_tag =
        m_index_session->create_session(dice_transaction.get());
      check_success(m_session_tag.is_valid());
      mi::base::Handle< nv::index::ISession const > session(
        dice_transaction->access< nv::index::ISession const >(
          m_session_tag));
      check_success(session.is_valid_interface());

      mi::base::Handle< nv::index::IScene > scene_edit(
        dice_transaction->edit<nv::index::IScene>(session->get_scene()));
      check_success(scene_edit.is_valid_interface());

      //------------------------------------------------------------------------
      // Scene setup: add stylized points and lines, scene parameters, camera.
      //------------------------------------------------------------------------
      // Add stylized point and lines to the scene
      check_success(create_stylized_lines_and_points(scene_edit.get(),
                            m_line_style, m_point_style, m_transparency,
                            dice_transaction.get()));

      // Create and edit a camera. Data distribution is based on
      // the camera. (Because only visible massive data are
      // considered)
      mi::base::Handle< nv::index::IPerspective_camera > cam(
        scene_edit->create_camera<nv::index::IPerspective_camera>());
      check_success(cam.is_valid_interface());
      setup_camera(cam.get());
      const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
      check_success(camera_tag.is_valid());

      const mi::math::Vector<mi::Uint32, 2> buffer_resolution(1025, 1024);
      m_image_file_canvas->set_resolution(buffer_resolution);

      // Set up the scene
      mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
        {   0.0f,   0.0f,   0.0f, },
        { 500.0f, 500.0f, 500.0f, },
      };
```

```
      // set the region of interest
      const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
      check_success(xyz_roi.is_volume());
      scene_edit->set_clipped_bounding_box(xyz_roi_st);

      // Set the scene global transformation matrix.
      // only change the coordinate system
      mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
        1.0f,  0.0f,  0.0f,  0.0f,
        0.0f,  1.0f,  0.0f,  0.0f,
        0.0f,  0.0f, -1.0f,  0.0f,
        0.0f,  0.0f,  0.0f,  1.0f
        );
      scene_edit->set_transform_matrix(transform_mat);

      // Set the current camera to the scene.
      check_success(camera_tag.is_valid());
      scene_edit->set_camera(camera_tag);
    }
    dice_transaction->commit();
  }

  // Rendering
  {
    const std::string fname = m_transparency ?
      get_output_file_name(m_outfname, m_point_style) :
      get_output_file_name(m_outfname, m_line_style);
   mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
  const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;


      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }

   ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
          << os.str();
      exit_code = 1;
    }

    // verify the generated frame
    if (!(verify_canvas_result(get_application_layer_interface(),
                m_image_file_canvas.get(), m_verify_image_fname, get_options())))
    {
      exit_code = 1;
    }
  }
}
```

```
  return exit_code;
}

bool Create_stylized_point_and_lines::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname           = sdict.get("outfname");
  m_verify_image_fname = sdict.get("verify_image_fname");
  m_line_style         = nv::index::app::get_uint32(sdict.get("line_style"));
  m_point_style        = nv::index::app::get_uint32(sdict.get("point_style"));
 m_transparency      = nv::index::app::get_bool(sdict.get("transparency", "false"));

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"

      << "        [-transparency bool]\n"
      << "            to render transparent points. (default: ["
      << m_transparency << "])\n"

      << "        [-line_style unsigned int]\n"
      << "            to choose a line style [0..8]. (default: ["
      << m_line_style << "])\n"
```

```
      << "        [-point_style unsigned int]\n"
      << "            to choose a point style [0..3]. (default: ["
      << m_point_style << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")"
      << std::endl;
    exit(1);
  }

  return true;
}

bool Create_stylized_point_and_lines::create_stylized_lines_and_points(
  nv::index::IScene*                scene_edit,
  mi::Uint32                        pick_line_style,
  mi::Uint32                        pick_point_style,
  bool                              transparency,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(scene_edit       != 0);
  check_success(dice_transaction != 0);

  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Add a light and a material
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
    headlight->set_intensity(color_intensity);
    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());
    group_node->append(headlight_tag, dice_transaction);

    // add material for 3D points shape (the material is only effective for 3D shape)
   mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(mi::math::Color(0.3f, 0.3f, 0.3f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    phong_1->set_specular(mi::math::Color(0.4f));
    phong_1->set_shininess(100.f);
   const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
    check_success(phong_1_tag.is_valid());
    group_node->append(phong_1_tag, dice_transaction);
  }

  // square shaped positions
  const mi::Sint32  column_count = 20;
  const mi::Sint32  point_count = 200;
  const mi::Float32 x_mag = 25.f;
```

```
const mi::Float32 y_mag = 25.f;
const mi::Float32 z     = 30.f;
const mi::Float32 y_offset_points = 0.f;
const mi::Float32 y_offset_lines  = 255.f;

// Point coordinates and its attributes. According to the
// attributes, color and radius are defined.
// Here, we create two point sets.
std::vector< mi::math::Vector_struct< mi::Float32, 3> > vertices;
std::vector< mi::math::Color_struct>                    colors;
std::vector< mi::Float32 >                              radii;
for(mi::Sint32 i = 0; i < point_count; ++i)
{
  mi::math::Vector_struct< mi::Float32, 3> pos;
  pos.x = static_cast< mi::Float32 >(i % column_count) * x_mag;
  pos.y = static_cast< mi::Float32 >(i / column_count) * y_mag + y_offset_points;
  pos.z = z;
  vertices.push_back(pos);

  mi::math::Color_struct color;
  color.r = static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
  color.g = static_cast<mi::Float32>(0.1);
 color.b = 1.f - static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
  color.a = static_cast<mi::Float32>(1.f);
  colors.push_back(color);

 mi::Float32 radius = (static_cast<mi::Float32>(i%(column_count))/static_cast<mi::Float32>(colum
  radii.push_back(static_cast<mi::Float32>(1+radius));
}
if(transparency)
{
  for(mi::Sint32 i = 0; i < point_count*2; ++i)
  {
    mi::math::Vector_struct< mi::Float32, 3> pos;
    pos.x = static_cast< mi::Float32 >(i % column_count) * x_mag;
    pos.y = static_cast< mi::Float32 >(i / column_count) * y_mag + y_offset_points;
    pos.z = z-3;
    vertices.push_back(pos);

    mi::math::Color_struct color;
  color.r = 1.f - static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
    color.g = static_cast<mi::Float32>(i)/static_cast< mi::Float32 >(point_count);
    color.b = static_cast<mi::Float32>(0.3);
    color.a = static_cast<mi::Float32>(0.7f);
    colors.push_back(color);

  mi::Float32 radius = (static_cast<mi::Float32>(i%(column_count))/static_cast<mi::Float32>(colu
    radii.push_back(static_cast<mi::Float32>(14-radius));
  }
}

// Create a point sets
nv::index::IPoint_set::Point_style available_point_styles[] =
{
  nv::index::IPoint_set::FLAT_CIRCLE,
  nv::index::IPoint_set::SHADED_CIRCLE,
  nv::index::IPoint_set::FLAT_SQUARE,
```

```
    nv::index::IPoint_set::FLAT_TRIANGLE
  };
 nv::index::IPoint_set::Point_style chosen_point_style = available_point_styles[pick_point_style];

  // Create point set using the scene's factory
 mi::base::Handle<nv::index::IPoint_set> point_set(scene_edit->create_shape<nv::index::IPoint_set>
  check_success(point_set.is_valid_interface());
  // ... set the point styles (see IPoint_set)
  point_set->set_point_style(chosen_point_style);
  // ... and the vertices with colors and radii.
  point_set->set_vertices(&vertices[0], vertices.size());
  point_set->set_colors(&colors[0], colors.size());
  point_set->set_radii(&radii[0], radii.size());

  // Add the point set to the database.
 const mi::neuraylib::Tag point_set_scene_element_tag = dice_transaction->store_for_reference_count
  check_success(point_set_scene_element_tag.is_valid());

  // Add to the scene description
  group_node->append(point_set_scene_element_tag, dice_transaction);

 INFO_LOG << "Added point_set (size: " << vertices.size() << ") to the scene (tag id: " << point_set_sc

  nv::index::ILine_set::Line_style available_line_styles[] =
  {
    nv::index::ILine_set::LINE_STYLE_SOLID,
    nv::index::ILine_set::LINE_STYLE_DASHED,
    nv::index::ILine_set::LINE_STYLE_DOTTED,
    nv::index::ILine_set::LINE_STYLE_CENTER,
    nv::index::ILine_set::LINE_STYLE_HIDDEN,
    nv::index::ILine_set::LINE_STYLE_PHANTOM,
    nv::index::ILine_set::LINE_STYLE_DASHDOT,
    nv::index::ILine_set::LINE_STYLE_BORDER,
    nv::index::ILine_set::LINE_STYLE_DIVIDE
  };

 nv::index::ILine_set::Line_style chosen_line_style = available_line_styles[pick_line_style];

  // Create a line sets
  const mi::Uint32 nb_lines = 10;
  for(mi::Uint32 j=0; j<nb_lines; ++j)
  {
    std::vector< mi::math::Vector_struct< mi::Float32, 3> > segment_vertices;
    std::vector< mi::math::Color_struct>                    color_per_segment;
    std::vector< mi::Float32 >                              width_per_segment;

    mi::math::Vector_struct< mi::Float32, 3> v0;
    v0.x = static_cast< mi::Float32 >(0.f);
    v0.y = static_cast< mi::Float32 >(j) * y_mag + y_offset_lines;
    v0.z = z;

    mi::math::Vector_struct< mi::Float32, 3> v1;
    v1.x = static_cast< mi::Float32 >(column_count) * x_mag;
    v1.y = static_cast< mi::Float32 >(j) * y_mag + y_offset_lines;
    v1.z = z;

    segment_vertices.push_back(v0);
```

```
    segment_vertices.push_back(v1);

    mi::math::Color_struct color;
    color.r = static_cast<mi::Float32>(j)/static_cast< mi::Float32 >(nb_lines);
    color.g = static_cast<mi::Float32>(0.1);
    color.b = 1.f - static_cast<mi::Float32>(j)/static_cast< mi::Float32 >(nb_lines);
    color.a = static_cast<mi::Float32>(1.f);
    color_per_segment.push_back(color);

    mi::Float32 width = static_cast<mi::Float32>(1.3*nb_lines+1-j);
    width_per_segment.push_back(static_cast<mi::Float32>(width));

    // Create line set using the scene's factory
   mi::base::Handle<nv::index::ILine_set> line_set(scene_edit->create_shape<nv::index::ILine_set>(
    check_success(line_set.is_valid_interface());
   // ... set the line style such a dashed or dotted or solid linestyle (see ILine_set)
    line_set->set_line_style(chosen_line_style);
   // ... set the line type. Currently only line segments are supported (see ILine_set)
    line_set->set_line_type(nv::index::ILine_set::LINE_TYPE_SEGMENTS);
    // ... and the lines with colors and widths all in line segment order.
    line_set->set_lines(&segment_vertices[0], segment_vertices.size());
    line_set->set_colors(&color_per_segment[0], color_per_segment.size());
    line_set->set_widths(&width_per_segment[0], width_per_segment.size());

    // Add the lines to the database
   const mi::neuraylib::Tag line_set_scene_element_tag = dice_transaction->store_for_reference_coun
    // If you haven't registered the Line_set, the next check fails.
    check_success(line_set_scene_element_tag.is_valid());
    // Add to the scene description
    group_node->append(line_set_scene_element_tag, dice_transaction);
   INFO_LOG << "Added line_set with line style " << pick_line_style << " to the scene (tag id: " << line
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  return true;
}

void Create_stylized_point_and_lines::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from( 254.0f, 254.0f,  550.0f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 255.0f, 255.0f, -255.0f);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(10.0f);
  cam->set_clip_max(5000.0f);
```

```
}

nv::index::IFrame_results* Create_stylized_point_and_lines::render_frame(const std::string& output_
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  const std::string com_name(argv[0]);

  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
 sdict.insert("outfname", "frame_create_stylized_points_and_lines"); // output file base name
  sdict.insert("verify_image_fname", "");                      // for unit test
  sdict.insert("line_style", "0");                             //
  sdict.insert("point_style", "0");                            //
  sdict.insert("transparency", "0");                           //
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1");  // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");
```

```
  // Initialize application
  Create_stylized_point_and_lines create_stylized_point_and_lines;
  create_stylized_point_and_lines.initialize(argc, argv, sdict);
  check_success(create_stylized_point_and_lines.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_stylized_point_and_lines.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.17 create_synthetic_heightfield.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/itexture_filter_mode.h>

#include <nv/index/app/idata_analysis_and_processing.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/canvas_utility.h"

#include <sstream>
#include <iostream>

class Create_synthetic_heightfield:
  public nv::index::app::Index_connect
{
public:
  Create_synthetic_heightfield()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Create_synthetic_heightfield() ctor";
  }

  virtual ~Create_synthetic_heightfield()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Create_synthetic_heightfield() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);
```

```
    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Setup camera to see this example scene
  // \param[in] cam    a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;

  // setup a far away camera for a extreme transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  //
  // \param[in] cam    a camera
  void setup_extreme_transformed_camera(nv::index::IPerspective_camera* cam) const;

  // setup a large scene transform for the extreme
  // transformation handling test
  //
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  mi::math::Matrix<mi::Float32, 4, 4> get_extreme_transformed_matrix() const;

  // render a frame
  //
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
 mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
 mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
 mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                             m_outfname;
  bool                                                                   m_is_unittest;
  std::string                                                         m_verify_image_fname;
  bool                                                                        m_seeds;
  bool                                                                  m_depth_buffer;
  bool                                                                 m_supersampling;
  std::string                                                               m_texture;
  bool                                                                  m_use_texture;
  bool                                                                m_is_large_translate;
};

mi::Sint32 Create_synthetic_heightfield::launch()
```

```
{
  mi::Sint32 exit_code = 0;
  {
   m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configura
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

   // Verifying that local host has joined. This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));

    {
      // Create our DiCE transaction
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());

      // Create an IndeX session.
      m_session_tag = m_index_session->create_session(dice_transaction.get());
      check_success(m_session_tag.is_valid());
      {
        // Change the global configuration.
        {
          mi::base::Handle<const nv::index::ISession> the_session(
            dice_transaction->access<const nv::index::ISession>(m_session_tag));
          check_success(the_session.is_valid_interface());

          mi::base::Handle<nv::index::IConfig_settings> config_settings(
        dice_transaction->edit<nv::index::IConfig_settings>(the_session->get_config()));
          check_success(config_settings.is_valid_interface());

          // Enable supersampling.
          if (m_supersampling)
          {
            INFO_LOG << "Enable supersampling.";
            config_settings->set_rendering_samples(8);
          }
        }

        // Create a scene that contains a synthetically generated heightfield.
        {
          // Access the session instance from the database.
          mi::base::Handle<const nv::index::ISession> session(
            dice_transaction->access<const nv::index::ISession>(m_session_tag));
          check_success(session.is_valid_interface());

          // Access (edit mode) the scene instance from the database.
          mi::base::Handle<nv::index::IScene> scene_edit(
            dice_transaction->edit<nv::index::IScene>(session->get_scene()));
          check_success(scene_edit.is_valid_interface());

          // Create static group node for large data
          mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
            scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
          check_success(static_group_node.is_valid_interface());
```

```
    // Details for creating a synthetic heightfield.
    const mi::math::Vector<mi::Uint32, 2> heightfield_size(500, 500);

    nv::index::app::String_dict heightfield_opt;
    heightfield_opt.insert("args::type",           "heightfield");
heightfield_opt.insert("args::importer",     "nv::index::plugin::legacy_importer.Synthetic_
    heightfield_opt.insert("args::synthetic_type", "i");
    heightfield_opt.insert("args::size",           "500 500");
    heightfield_opt.insert("args::range",          "0.1 1000");
    nv::index::IDistributed_data_import_callback* importer_callback =
      get_importer_from_application_layer(
        get_application_layer_interface(),
        "nv::index::plugin::legacy_importer.Synthetic_heightfield_generator",
        heightfield_opt);

    // Create heightfield scene element and add it to the scene
    const mi::Float32 rotate_k = 0.0f;
    const mi::math::Vector<mi::Float32, 3> translate(0.0f, 0.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 3> scale(1.0f, 1.0f, 1.0f);
    const mi::math::Vector<mi::Float32, 2> elevation_range(50.0f, 256.0f);

  mi::base::Handle<nv::index::IRegular_heightfield> heightfield_scene_element(
      scene_edit->create_regular_heightfield(
        scale, rotate_k, translate,
        heightfield_size,
        elevation_range,
        importer_callback,
        dice_transaction.get()));
    check_success(heightfield_scene_element.is_valid_interface());

const mi::math::Bbox<mi::Float32, 3> heightfield_bbox = heightfield_scene_element->get_IJK_bo

    // Set the name of the heightfield scene element
    const std::string heightfield_name = "synthetic heightfield";
    heightfield_scene_element->set_name(heightfield_name.c_str());

    // Render the heightfield with additional seed points and lines
    if (m_seeds)
    {
      // Add a line
      {
        std::vector<mi::math::Vector<mi::Float32, 3> > seed_line;
    seed_line.push_back(mi::math::Vector<mi::Float32, 3>(400.f, 100.f, 100.f));
    seed_line.push_back(mi::math::Vector<mi::Float32, 3>(500.f, 100.f, 200.f));
        check_success(!seed_line.empty());
    heightfield_scene_element->add_seed_line(&seed_line[0], seed_line.size());
      }

      // Add a few points
      {
        std::vector<mi::math::Vector<mi::Float32, 3> > seed_points;
        for (int i = 0; i < 100; ++i)
        {
          seed_points.push_back(mi::math::Vector<mi::Float32, 3>(
                      360.0f, static_cast<mi::Float32>(i) * 10.0f, 100.0f));
        }
```

```
        check_success(!seed_points.empty());
    heightfield_scene_element->add_seed_points(&seed_points[0], seed_points.size());
      }
      //
  // Enable the rendering of seed points and lines by adding attributes to the scene
      //

      // Settings for seed points
      {
    mi::base::Handle<nv::index::IHeightfield_geometry_settings> point_settings(
     scene_edit->create_attribute<nv::index::IHeightfield_geometry_settings>());
        check_success(point_settings.is_valid_interface());
        // Apply this attribute only to seed points
    point_settings->set_type_mask(nv::index::IHeightfield_geometry_settings::TYPE_SEED_POINT
        // Use a fixed color with no lighting
    point_settings->set_color_mode(nv::index::IHeightfield_geometry_settings::MODE_FIXED);
        // Blue points please
        point_settings->set_color(mi::math::Color(0.f, 0.f, 1.f));
        // Enable rendering of seed points
        point_settings->set_visible(true);

        const mi::neuraylib::Tag point_settings_tag
       = dice_transaction->store_for_reference_counting(point_settings.get());
        check_success(point_settings_tag.is_valid());
        static_group_node->append(point_settings_tag, dice_transaction.get());
      }

      // Settings for seed lines
      {
     mi::base::Handle<nv::index::IHeightfield_geometry_settings> line_settings(
      scene_edit->create_attribute<nv::index::IHeightfield_geometry_settings>());
        check_success(line_settings.is_valid_interface());
        // Apply this attribute only to seed lines
    line_settings->set_type_mask(nv::index::IHeightfield_geometry_settings::TYPE_SEED_LINES)
        // Use a fixed color with no lighting
    line_settings->set_color_mode(nv::index::IHeightfield_geometry_settings::MODE_FIXED);
        // Red lines please
        line_settings->set_color(mi::math::Color(1.f, 0.f, 0.f));
        // Enable rendering of seed lines
        line_settings->set_visible(true);

        const mi::neuraylib::Tag line_settings_tag
        = dice_transaction->store_for_reference_counting(line_settings.get());
        check_success(line_settings_tag.is_valid());
        static_group_node->append(line_settings_tag, dice_transaction.get());
      }
     }

  // ... and store the heightfield scene element in the distributed database ...
      const mi::neuraylib::Tag heightfield_tag =
  dice_transaction->store_for_reference_counting(heightfield_scene_element.get());
      check_success(heightfield_tag.is_valid());

      // Add a light and a material to the static group node
      {
        // Add a light
        mi::base::Handle<nv::index::IDirectional_headlight> headlight(
```

```
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color color_intensity(1.0f, 1.0f, 1.0f, 1.0f);
    headlight->set_intensity(color_intensity);
headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(hea
    check_success(headlight_tag.is_valid());
    static_group_node->append(headlight_tag, dice_transaction.get());

    // Material for the heightfield
mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPh
    check_success(phong_1.is_valid_interface());

    if (m_use_texture)
    {
      // Use just ambient white with texture
      phong_1->set_ambient(mi::math::Color(1.f));
      phong_1->set_diffuse(mi::math::Color(0.f));
      phong_1->set_specular(mi::math::Color(0.f));
    }
    else
    {
      // Define a more interesting greenish material
      phong_1->set_ambient(mi::math::Color(0.f, 0.3f, 0.0f, 0.3f));
      phong_1->set_diffuse(mi::math::Color(0.f, 0.8f, 0.2f, 0.3f));
      phong_1->set_specular(mi::math::Color(0.6f));
      phong_1->set_shininess(100.f);
    }

    const mi::neuraylib::Tag phong_1_tag
      = dice_transaction->store_for_reference_counting(phong_1.get());
    check_success(phong_1_tag.is_valid());
    static_group_node->append(phong_1_tag, dice_transaction.get());
  }

  // Optionally map a computed texture onto the heightfield
  if (m_use_texture)
  {
    // Mandelbrot texture
    if (m_texture == "mandelbrot")
    {
      mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
   scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
      check_success(tex_filter.is_valid_interface());
  mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_fi
      check_success(tex_filter_tag.is_valid());
      static_group_node->append(tex_filter_tag, dice_transaction.get());

      // Create the computed texture, with a size fitting the heightfield
  mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_proces
  get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_p
      check_success(processing.is_valid_interface());
  // Create a mandelbrot technique and add it to the scene. For more details on how to implement
  // source that was shipped with the application layer component 'nv::index::app::data_analys
      mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
   processing->get_sample_tool_set()->create_mandelbrot_2d_technique(mi::math::Vector<mi::
```

```
    // Add the mapping to the scene before the heightfield
mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.g
    check_success(mapping_tag.is_valid());
    static_group_node->append(mapping_tag, dice_transaction.get());
  }
  // Map height values in the heightfield to colors
  else if (m_texture == "height_color")
  {
 // Create the computed texture, passing the height range of the heightfield
    const mi::math::Vector<mi::Float32, 2> height_range(
      heightfield_bbox.min.z, heightfield_bbox.max.z);

    mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
 scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
    check_success(tex_filter.is_valid_interface());
mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_fi
    check_success(tex_filter_tag.is_valid());
    static_group_node->append(tex_filter_tag, dice_transaction.get());

    // Create the computed texture, with a size fitting the heightfield
mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_proces
get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_p
    check_success(processing.is_valid_interface());
// Create a mandelbrot technique and add it to the scene. For more details on how to implement
// source that was shipped with the application layer component 'nv::index::app::data_analys
    const mi::math::Color color0(0.f, 1.f, 0.f, 1.f); // green
    const mi::math::Color color1(1.f, 0.f, 0.f, 1.f); // red
    mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
 processing->get_sample_tool_set()->create_color_encoded_elevation_technique(height_rang
    check_success(mapping.is_valid_interface());

    // Add the mapping to the scene before the heightfield
mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.g
    static_group_node->append(mapping_tag, dice_transaction.get());
  }
  else
  {
    ERROR_LOG << "Unknown texture mode '" << m_texture << "'";
    check_success(false);
  }
}

// Append the heightfield to the scene group
static_group_node->append(heightfield_tag, dice_transaction.get());
mi::neuraylib::Tag static_group_node_tag =
  dice_transaction->store_for_reference_counting(static_group_node.get());
check_success(static_group_node_tag.is_valid());

// Append the static scene group to the scene.
scene_edit->append(static_group_node_tag, dice_transaction.get());

std::stringstream sstr;
sstr << "Created a synthetic heightfield: size = "
  << heightfield_size << ", tag = " << heightfield_tag.id;
INFO_LOG << sstr.str();

// Create a camera and adjust the camera parameter.
```

```
    mi::base::Handle< nv::index::IPerspective_camera > cam(
      scene_edit->create_camera<nv::index::IPerspective_camera>());
    check_success(cam.is_valid_interface());

    mi::math::Vector<mi::Uint32,2> buffer_resolution;
    if (m_is_large_translate)
    {
      INFO_LOG << "large translate test mode.";
      setup_extreme_transformed_camera(cam.get());
      buffer_resolution = mi::math::Vector<mi::Uint32,2>(1602, 934);
    }
    else
    {
      setup_camera(cam.get());
      buffer_resolution = mi::math::Vector<mi::Uint32,2>(512, 512);
    }
    const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());

    m_image_file_canvas->set_resolution(buffer_resolution);

    // Define a region of interest for the entire scene in the
    // scene's global coordinate system.
    const mi::math::Bbox<mi::Float32, 3> region_of_interest(
      0.f, 0.f, 0.f,
      static_cast< mi::Float32 >(heightfield_size.x),
      static_cast< mi::Float32 >(heightfield_size.y),
  static_cast< mi::Float32 >(heightfield_size.x)  // Note: using the x dimension to define the z
      );
    scene_edit->set_clipped_bounding_box(region_of_interest);

    // Finally, optionally adjust the scene's coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f, // adjust for coordinate system
      0.0f,  0.0f,  0.0f,  1.0f
      );

    if (m_is_large_translate)
    {
      transform_mat = get_extreme_transformed_matrix();
    }

    scene_edit->set_transform_matrix(transform_mat);

    // ... and add the camera to the scene.
    scene_edit->set_camera(camera_tag);
  }
  // Finish the initialization transaction
  dice_transaction->commit();
}

// Rendering
{
  // Render a frame and save the rendered image to a file.
  const mi::Sint32  frame_idx = 0;
  const std::string fname = get_output_file_name(m_outfname, frame_idx);
```

```
        if(m_depth_buffer)
        {
          // The layer is added by automatically when set the depth_file_name
          const std::string depth_file_name = m_outfname + "_depth_buffer";
       m_image_file_canvas->set_depth_file_name(get_output_file_name(depth_file_name, frame_idx).c
        }

      mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
      const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
        if (err_set->any_errors())
        {
          std::ostringstream os;
          const mi::Uint32 nb_err = err_set->get_nb_errors();
          for (mi::Uint32 e = 0; e < nb_err; ++e)
          {
            if (e != 0) os << '\n';
            const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
            os << err->get_error_string();
          }

        ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
                << os.str();
          exit_code = 1;
        }

        // if(layer_id != invalid_layer_id)
        // {
        //     arc.m_canvas.detach_layer(layer_id);
        // }

        // Verify the generated frame
        if (!verify_canvas_result(get_application_layer_interface(),
                    m_image_file_canvas.get(), m_verify_image_fname, get_options()))
        {
          exit_code = 1;
        }
       }
    }
  }

  return exit_code;
}

bool Create_synthetic_heightfield::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }
```

```
 m_outfname           = sdict.get("outfname");
 m_verify_image_fname = sdict.get("verify_image_fname");
 m_seeds              = nv::index::app::get_bool(sdict.get("seeds", "false"));
m_depth_buffer      = nv::index::app::get_bool(sdict.get("depth_buffer", "false"));
m_supersampling    = nv::index::app::get_bool(sdict.get("supersampling", "false"));
 m_texture            = sdict.get("texture");
 m_use_texture        = m_texture != "none";
m_is_large_translate = nv::index::app::get_bool(sdict.get("is_large_translate", "false"));

 info_cout("running " + com_name, sdict);
 info_cout("outfname = ["              + m_outfname +
      "], verify_image_fname = [" + m_verify_image_fname +
      "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

 // print help and exit if -h
 if (sdict.is_defined("h"))
 {
   std::cout
     << "info: Usage: " << com_name << " [option]\n"
     << "Option: [-h]\n"
     << "            print out this message\n"
     << "        [-dice::verbose severity_level]\n"
   << "         verbose severity level (3 is info). (default: " + sdict.get("dice::verbose")
     << ")\n"

     << "        [-seeds bool]\n"
    << "          render with additional 'seed' points and lines. (default: " + m_seeds
     << ")\n"

     << "        [-depth_buffer bool]\n"
   << "         render depth buffer and store depth buffer as image (ppm). (default: " + m_depth_buffe
     << ")\n"

     << "        [-supersampling bool]\n"
     << "            when true then supersampling is enabled."
     << "(default: " << m_supersampling << ")\n"

     << "        [-texture mode]\n"
     << "            map a computed texture onto the heightfield.\n"
     << "            Available modes: none, mandelbrot, height_color\n"
     << "            (default: " << m_texture << ")\n"

     << "        [-is_large_translate bool]\n"
     << "            on/off large translation mode."
     << "(default: " << m_is_large_translate << ")\n"

     << "        [-outfname string]\n"
     << "            output ppm file base name. When empty, no output.\n"
     << "            A frame number and extension (.ppm) will be added.\n"
     << "            (default: [" << m_outfname << "])\n"

     << "        [-verify_image_fname image_fname]\n"
     << "            when image_fname exist, verify the rendering image.\n"
     << "            (default: [" << m_verify_image_fname << "])\n"

     << "        [-unittest bool]\n"
```

```
      << "            when true, unit test mode. "
      << "(default: [" << m_is_unittest << "])"
      << std::endl;
    exit(1);
  }

  return true;
}

void Create_synthetic_heightfield::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  const mi::math::Vector<mi::Float32, 3> from(700, 700, 200);
  const mi::math::Vector<mi::Float32, 3> to  ( 250.f, 250.f, -200.f);
  const mi::math::Vector<mi::Float32, 3> up  ( 0.f, 0.f,  1.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(1000.0f);
}

void Create_synthetic_heightfield::setup_extreme_transformed_camera(nv::index::IPerspective_camera
{
  check_success(cam != 0);

  // *** Camera:
  // index::camera::eye_point = 1808409.125 9981818 10948.5126953125
  // index::camera::view_direction = (1808409.125 9981818 -1500) - eye_point
  // index::camera::up_direction = 0 1 0
  // index::camera::aspect = 1.7152034261242
  // index::camera::aperture = 0.033
  // index::camera::focal = 0.03
  // index::camera::clip_min = 124.485130310059
  // index::camera::clip_max = 17585.455078125
  // index::canvas_resolution = 1602 934

  // Adjusted the camera position for p = 23+1.
 // mi::math::Vector< mi::Float32, 3 > const from( 1805060.125f + 1024.0f, 9978520.0f + 1024.0f,  256.
 // mi::math::Vector< mi::Float32, 3 > const to  ( 1805060.125f + 256.0f,  9978520.0f + 256.0f,  -256.0
 mi::math::Vector< mi::Float32, 3 > const from( 1805060.125f + 20480.0f, 9978520.0f + 20480.0f,  2048
 mi::math::Vector< mi::Float32, 3 > const to  ( 1805060.125f + 5120.0f,  9978520.0f + 5120.0f,  -1024.0
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   0.0f,   1.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.71520f); // not 1.7152034261242f, see IEEE754
  cam->set_focal(0.03f);
  cam->set_clip_min(124.485f); // not 124.485130310059f, see IEEE754
```

```
  cam->set_clip_max(17585.6f); // not 17585.55078125f,   see IEEE754

  INFO_LOG << "Set camera extreme mode";
}

mi::math::Matrix<mi::Float32, 4, 4> Create_synthetic_heightfield::get_extreme_transformed_matrix()
{
  // transform =
  //  [ 20.6100006103516   0                0       1805060
  //    0                 20.6100006103516  0       9978520
  //    0                  0               -4       0
  //    0                  0                0       1       ]
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    20.0f,      0.0f,          0.0f,  0.0f,
    0.0f,        20.0,         0.0f,  0.0f,
    0.0f,        0.0f,        -20.0f,  0.0f,
    1805060.0f,  9978520.0f,   0.0f,  1.0f
    );

  return transform_mat;
}

nv::index::IFrame_results* Create_synthetic_heightfield::render_frame(const std::string& output_fna
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                              // log level
 sdict.insert("seeds", "0");                               // enable seeds (default off)
 sdict.insert("depth_buffer", "0");                       // write depth buffer to image file
 sdict.insert("outfname", "frame_create_synthetic_heightfield"); // output file base name
```

```
  sdict.insert("verify_image_fname", "");                          // for unit test
  sdict.insert("unittest", "0");                                   // unit test mode
 sdict.insert("supersampling", "0");                     // disable supersampling (default)
 sdict.insert("texture", "none");                            // texture (default none)
 sdict.insert("is_large_translate", "0");                 // large translation mode (default 0)
 sdict.insert("is_dump_comparison_image_when_failed", "1");    // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                  // default: not call from make check.

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io data_analysis_and_processing");
  sdict.insert("index::app::plugins::base_importer::enabled",   "true");
  sdict.insert("index::app::plugins::legacy_importer::enabled", "true");

  // Initialize application
  Create_synthetic_heightfield create_synthetic_heightfield;
  create_synthetic_heightfield.initialize(argc, argv, sdict);
  check_success(create_synthetic_heightfield.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = create_synthetic_heightfield.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.18 distributed_sparse_volume_data.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <iostream>
#include <limits>
#include <sstream>
#include <vector>

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/idistributed_data_access.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/isparse_volume_subset.h>
#include <nv/index/isparse_volume_rendering_properties.h>

#include <nv/index/app/forwarding_logger.h>
#include <nv/index/app/idata_analysis_and_processing.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

class Distributed_sparse_volume_data:
  public nv::index::app::Index_connect
{
public:
  Distributed_sparse_volume_data()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Distributed_sparse_volume_data() ctor";
  }

  virtual ~Distributed_sparse_volume_data()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Distributed_sparse_volume_data() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
```

```
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
 mi::Size calc_offset(const mi::math::Vector<mi::Sint32, 3>& p, const mi::math::Vector<mi::Uint32, 3
  mi::Float32 analyze_sparse_volume_data(
    mi::neuraylib::IDice_transaction* dice_transaction) const;
  // Edit the data in the volume
  // \param[in] dice_transaction transaction
  void edit_sparse_volume_data(
    mi::neuraylib::IDice_transaction* dice_transaction) const;
  // Export the data in the volume
  // \param[in] dice_transaction transaction
  // \returns average of the data in the volume
  mi::Float32 export_sparse_volume_data(
    mi::neuraylib::IDice_transaction* dice_transaction) const;
  void render_frame(const std::string& output_filename) const;

  // Create a synthetic sparse volume to the scene.
  mi::neuraylib::Tag create_synthetic_volume(
    const nv::index::ISession*             session,
    const mi::math::Vector<mi::Uint32, 3>& volume_size,
    mi::neuraylib::IDice_transaction*      dice_transaction) const;
  std::vector<mi::Uint32> evaluate_sparse_volume_data_locality(
    const nv::index::IDistributed_data_locality* data_locality,
    const mi::math::Bbox<mi::Sint32, 3>&         query_bound,
    const mi::neuraylib::Tag&                    scene_element_tag,
    const std::string&                           scene_element_type) const;
  // Get brick's clamped bboxes
  //
  // \param[in]  svol_data_subset_desc
  // \param[in]  brick_idx
  // \param[in]  brick_position
  // \param[in]  query_bbox_s32
  // \param[out] out_brick_box_subdata_clamped
  // \param[out] out_brick_box_clamped
  void get_brick_clamped_bbox(
    const nv::index::ISparse_volume_subset_data_descriptor* svol_data_subset_desc,
    const mi::Uint32                                        brick_idx,
    const mi::math::Vector<mi::Sint32, 3>&                  brick_position,
    const mi::math::Bbox<mi::Sint32, 3>&                    query_bbox_s32,
   mi::math::Bbox<mi::Sint32, 3>&                     out_brick_box_subdata_clamped,
   mi::math::Bbox<mi::Sint32, 3>&                      out_brick_box_clamped) const;

  // This session tag
```

```
  mi::neuraylib::Tag                                                    m_session_tag;
   // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
   // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
   // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                    m_is_unittest;
  std::string                                                            m_voxel_format;
  mi::neuraylib::Tag                                                  m_sparse_volume_tag;
};

mi::Sint32 Distributed_sparse_volume_data::launch()
{
 m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configurat
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  {
   mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
    check_success(dice_transaction.is_valid_interface());
    {
      // Setup session information
      m_session_tag = m_index_session->create_session(dice_transaction.get());
      check_success(m_session_tag.is_valid());

    mi::base::Handle<const nv::index::ISession> session(dice_transaction->access<nv::index::ISessi
      check_success(session.is_valid_interface());

      // Setup the session's configuration
      {
        mi::base::Handle<nv::index::IConfig_settings> edit_config_settings(
        dice_transaction->edit<nv::index::IConfig_settings>(session->get_config()));
        check_success(edit_config_settings.is_valid_interface());

      // All data should be uploaded immediately, even when it is not initially visible
        edit_config_settings->set_force_data_upload(true);

        // Set smaller subcube size in unit test mode
        if (m_is_unittest)
        {
          const mi::math::Vector<mi::Uint32, 3> subcube_size(30);
          mi::math::Vector<mi::Float32, 3> minimal_volume_scaling;
          edit_config_settings->get_minimal_volume_scaling(minimal_volume_scaling);
          edit_config_settings->set_subcube_configuration(
            subcube_size,
            edit_config_settings->get_subcube_border_size(),
            edit_config_settings->get_continuous_volume_translation_supported(),
            edit_config_settings->get_volume_rotation_supported(),
            minimal_volume_scaling);
```

```
      }
    }

    //------------------------------------------------------------------------
    // Scene setup: add volume data, scene parameters
    //------------------------------------------------------------------------

    // the Size of the volume and the global scene transformation used below.
    mi::math::Vector<mi::Uint32, 3> volume_size(1000, 1000, 1000);
    mi::math::Matrix<mi::Float32, 4, 4> transform(
      0.02f, 0.0f,  0.0f,  0.0f,
      0.0f,  0.02f, 0.0f,  0.0f,
      0.0f,  0.0f, -0.02f, 0.0f, // adjust for coordinate system
      0.0f,  0.0f,  0.0f,  1.0f
      );

  // Use a smaller volume but a larger global scene scaling in unit test mode, to speed things up.
    if (m_is_unittest)
    {
      volume_size = mi::math::Vector<mi::Uint32, 3>(100, 100, 100);
      transform = mi::math::Matrix<mi::Float32, 4, 4>(
        0.2f, 0.0f,  0.0f,  0.0f,
        0.0f,  0.2f, 0.0f,  0.0f,
        0.0f,  0.0f, -0.2f, 0.0f, // adjust for coordinate system
        0.0f,  0.0f,  0.0f,  1.0f
        );
    }

  m_sparse_volume_tag = create_synthetic_volume(session.get(), volume_size, dice_transaction.get(
    check_success(m_sparse_volume_tag.is_valid());
    INFO_LOG << "Created a synthetic sparse volume dataset: size = "
     << "[" << volume_size.x << " " << volume_size.y << " " << volume_size.z << "], " << "tag = " << m_

    // Set up the scene and the camera
  mi::base::Handle<nv::index::IScene> scene_edit(dice_transaction->edit<nv::index::IScene>(sessi
    check_success(scene_edit.is_valid_interface());

    // Set region of interest to fit the volume in this example
    const mi::math::Bbox<mi::Float32, 3> global_roi(
      0.0f, 0.0f, 0.0f,
      static_cast<mi::Float32>(volume_size.x),
      static_cast<mi::Float32>(volume_size.y),
      static_cast<mi::Float32>(volume_size.z));
    scene_edit->set_clipped_bounding_box(global_roi);
    scene_edit->set_transform_matrix(transform);

    // Create a camera and set the camera parameters to see the whole volume data.
    mi::base::Handle< nv::index::IPerspective_camera > cam(
      scene_edit->create_camera<nv::index::IPerspective_camera>());
    check_success(cam.is_valid_interface());

    const mi::math::Vector< mi::Float32, 3 > from(-6.0f, 35.0f,   7.0f);
    const mi::math::Vector< mi::Float32, 3 > to  ( 9.0f, 14.f,  -11.0f);
    const mi::math::Vector< mi::Float32, 3 > up  ( 0.0f, -0.10f,  1.0f);
    mi::math::Vector<mi::Float32, 3> viewdir = to - from;
    viewdir.normalize();
```

```
      cam->set(from, viewdir, up);
      cam->set_aperture(0.033f);
      cam->set_aspect(1.0f);
      cam->set_focal(0.03f);
      cam->set_clip_min(2.0f);
      cam->set_clip_max(400.0f);

      const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
      check_success(camera_tag.is_valid());

      scene_edit->set_camera(camera_tag);

      const mi::math::Vector<mi::Uint32, 2> canvas_resolution(1024, 1024);
      m_image_file_canvas->set_resolution(canvas_resolution);
    }
    dice_transaction->commit();
  }

  // The following block invokes the example code/functionality:
  {
    INFO_LOG << "Render the initial sparse 3D volume ...";
    mi::Sint32 frame_idx = 0;
    {
      const std::string outfname = get_output_file_name(m_outfname, frame_idx);
      ++frame_idx;
      render_frame(outfname);
    }

    // Applying operations to the volume now ...
    INFO_LOG << "Analyzing the sparse volume data ...";
    {
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
      check_success(dice_transaction.is_valid_interface());
      {
        mi::Float32 value = analyze_sparse_volume_data(dice_transaction.get());
     INFO_LOG << "voxel_format: " << m_voxel_format << ", unittest: " << m_is_unittest << ", generated

        if (m_voxel_format == "uint8")
        {
          if (m_is_unittest)
          {
        check_success(fabs(value - 225.47f) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
          }
          else
          {
        check_success(fabs(value - 237.931f) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
          }
        }
        else if (m_voxel_format == "rgba8")
        {
          if (m_is_unittest)
          {
        check_success(fabs(value - 126.25) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
          }
          else
          {
        check_success(fabs(value - 50.625) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
```

```
        }
      }
      else if (m_voxel_format == "sint16")
      {
        if (m_is_unittest)
        {
      check_success(fabs(value - 29044.3f) < (65535.0f/1000.0f)); // (65535.0f/1000.0f) 0.1%
        }
        else
        {
      check_success(fabs(value - 30577.9f) < (65535.0f/1000.0f)); // (65535.0f/1000.0f) 0.1%
        }
      }
      else if (m_voxel_format == "float32")
      {
        if (m_is_unittest)
        {
      check_success(fabs(value - 0.886378f) < (1.0f/1000.0f)); // (1.0f/1000.0f) 0.1%
        }
        else
        {
      check_success(fabs(value - 0.935178) < (1.0f/1000.0f)); // (1.0f/1000.0f) 0.1%
        }
      }
      else
      {
        ERROR_LOG << "Unknown voxel format: " << m_voxel_format;
      }
    }
    dice_transaction->commit();
  }

  INFO_LOG << "Editing the sparse volume data ...";
  {
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
    check_success(dice_transaction.is_valid_interface());
    {
      edit_sparse_volume_data(dice_transaction.get());
    }
    dice_transaction->commit();
  }

  INFO_LOG << "Render the edited sparse 3D volume ...";
  {
    const std::string outfname = get_output_file_name(m_outfname, frame_idx);
    ++frame_idx;
    render_frame(outfname);
  }

  INFO_LOG << "Analyzing the sparse volume data after editing ...";
  {
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
    check_success(dice_transaction.is_valid_interface());
    {
      mi::Float32 value = analyze_sparse_volume_data(dice_transaction.get());
      // Set all values to 42.0
      check_success(value == 42.0f);
```

```
    }
    dice_transaction->commit();
  }

  INFO_LOG << "Exporting subset of the sparse volume data ...";
  {
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
    check_success(dice_transaction.is_valid_interface());
    {
      mi::Float32 value = export_sparse_volume_data(dice_transaction.get());

   INFO_LOG << "voxel_format: " << m_voxel_format << ", unittest: " << m_is_unittest << ", export va

      if (m_voxel_format == "uint8")
      {
        if (m_is_unittest)
        {
      check_success(fabs(value - 165.244f) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
        }
        else
        {
      check_success(fabs(value - 171.237f) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
        }
      }
      else if (m_voxel_format == "rgba8")
      {
        if (m_is_unittest)
        {
      check_success(fabs(value - 118.402f) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
        }
        else
        {
      check_success(fabs(value - 2.40677f) < (255.0f/1000.0f)); // (255.0f/1000.0f) 0.1%
        }
      }
      else if (m_voxel_format == "sint16")
      {
        if (m_is_unittest)
        {
      check_success(fabs(value - 19539.6f) < (65535.0f/1000.0f)); // (65535.0f/1000.0f) 0.1%
        }
        else
        {
      check_success(fabs(value - 20579.5f) < (65535.0f/1000.0f)); // (65535.0f/1000.0f) 0.1%
        }
      }
      else if (m_voxel_format == "float32")
      {
        if (m_is_unittest)
        {
      check_success(fabs(value - 13.6316f) < (1.0f/1000.0f)); // (1.0f/1000.0f) 0.1%
        }
        else
        {
      check_success(fabs(value - 11.4886f) < (1.0f/1000.0f)); // (1.0f/1000.0f) 0.1%
        }
      }
```

```
        else
        {
          ERROR_LOG << "Unknown voxel format: " << m_voxel_format;
        }
      }
      dice_transaction->commit();
    }
  }

  return 0;
}

bool Distributed_sparse_volume_data::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("dice::verbose", "2");
    sdict.insert("outfname", "");
  }

  m_outfname     = sdict.get("outfname");
  m_voxel_format = sdict.get("voxel_format");

  info_cout(std::string("Running ") + com_name, sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name <<" [option]\n"
      << "Option: [-h]\n"
      << "            print this message\n"

      << "        [-dice::network::multicast_address address]\n"
      << "            set multicast address. (default: ["
      << sdict.get("dice::network::multicast_address") + "])\n"

      << "        [-dice::network::cluster_interface]\n"
      << "            set cluster interface. (default: ["
      << sdict.get("dice::network::cluster_interface") + "])\n"

      << "        [-voxel_format VOXEL_FORMAT]\n"
      << "            the voxel format. 'uint8', 'rgba8', 'sint16', 'float32' \n"
      << "            (default: [" << m_voxel_format << "])\n"

      // Note: subcube_size is [30 30 30], the generator shows
      // multiple same generate bounds. Because that generate
      // bounds has a subcube. These subcubes are shared in a
      // brick. Thus, they show multiple times.
      << "        [-unittest bool]\n"
```

```
    << "        when true, unit test mode (create smaller volume). Also small subcube_size."
      << sdict.get("unittest") + "])\n"

      << std::endl;
    exit(1);
  }

  return true;
}

mi::Size Distributed_sparse_volume_data::calc_offset(const mi::math::Vector<mi::Sint32, 3>& p, cons
{
  const mi::Size o =   static_cast<mi::Size>(p.x + b)
            + static_cast<mi::Size>(p.y + b) * s.x
            + static_cast<mi::Size>(p.z + b) * s.x * s.y;
  return o;
}

mi::neuraylib::Tag Distributed_sparse_volume_data::create_synthetic_volume(
  const nv::index::ISession*             session,
  const mi::math::Vector<mi::Uint32, 3>& volume_size,
  mi::neuraylib::IDice_transaction*      dice_transaction) const
{
  check_success(dice_transaction != 0);
  check_success(volume_size.x > 0 && volume_size.y > 0 && volume_size.z > 0);

 const mi::math::Bbox<mi::Uint32, 3> volume_bbox(mi::math::Vector<mi::Uint32, 3>(0), volume_size);

 // A synthetic volume generator relies on the data import functionality of NVIDIA IndeX. Instead of re
  // the data is created on the fly.
  // sparse volume creation parameter
  nv::index::app::String_dict sparse_volume_opt;
  sparse_volume_opt.insert("args::type",     "sparse_volume");
 sparse_volume_opt.insert("args::importer", "nv::index::plugin::base_importer.Sparse_volume_genera
  {
    std::stringstream sstr;
   sstr << "0 0 0 " << volume_size.x << " " << volume_size.y << " " << volume_size.z;
    sparse_volume_opt.insert("args::bbox", sstr.str());
  }
  sparse_volume_opt.insert("args::voxel_format",                m_voxel_format);
  sparse_volume_opt.insert("args::synthetic_type",              "perlin_noise");
  sparse_volume_opt.insert("args::parameter::cube_unit",        "1024");
  sparse_volume_opt.insert("args::parameter::time",             "0");
  sparse_volume_opt.insert("args::parameter::terms",            "2");
  sparse_volume_opt.insert("args::parameter::turbulence_weight", "1 1 1 1");
  sparse_volume_opt.insert("args::parameter::abs_noise",        "false");
  sparse_volume_opt.insert("args::parameter::ridged",           "true");
  nv::index::IDistributed_data_import_callback* generator =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::base_importer.Sparse_volume_generator_synthetic",
      sparse_volume_opt);

 // Create the volume scene element using the above importer and parameters and the scene (that is part
 mi::base::Handle<nv::index::IScene> scene(dice_transaction->edit<nv::index::IScene>(session->get_
  check_success(scene.is_valid_interface());
```

```
  mi::math::Matrix<mi::Float32, 4, 4>    transform_mat(1.0f); // Identity matrix
  const bool                             is_rendering_enabled = true;
  const mi::math::Bbox<mi::Float32, 3>   ijk_bbox(volume_bbox);
  mi::base::Handle<nv::index::ISparse_volume_scene_element> svol_scene_element(
   scene->create_sparse_volume(ijk_bbox, transform_mat, generator, dice_transaction));
  check_success(svol_scene_element.is_valid_interface());
  svol_scene_element->set_enabled(is_rendering_enabled);

  if(svol_scene_element.is_valid_interface())
  {
   const mi::neuraylib::Tag svol_scene_element_tag = dice_transaction->store_for_reference_counting
    check_success(svol_scene_element_tag.is_valid());

    // Create static group node for large data and append the volume
   mi::base::Handle<nv::index::IStatic_scene_group> static_group(scene->create_scene_group<nv::ind
    check_success(static_group.is_valid_interface());
    {
      // Add a sparse_volume_render_properties to the scene.
    mi::base::Handle<nv::index::ISparse_volume_rendering_properties> sparse_render_prop(
        scene->create_attribute<nv::index::ISparse_volume_rendering_properties>());
      sparse_render_prop->set_filter_mode(nv::index::SPARSE_VOLUME_FILTER_NEAREST);
      sparse_render_prop->set_sampling_distance(             1.0);
      sparse_render_prop->set_reference_sampling_distance(   1.0);
    sparse_render_prop->set_voxel_offsets(           mi::math::Vector<mi::Float32, 3>(0.0f, 0.0f,
      sparse_render_prop->set_preintegrated_volume_rendering(false);
      sparse_render_prop->set_lod_rendering_enabled(         false);
      sparse_render_prop->set_lod_pixel_threshold(           2.0);
      sparse_render_prop->set_debug_visualization_option(    0);
      const mi::neuraylib::Tag sparse_render_prop_tag
        = dice_transaction->store_for_reference_counting(sparse_render_prop.get());
      check_success(sparse_render_prop_tag.is_valid());
      static_group->append(sparse_render_prop_tag, dice_transaction);
     INFO_LOG << "Created a sparse_render_prop_tag: tag = " << sparse_render_prop_tag;

      // Add a colormap to the scene.
    const mi::Sint32 colormap_entry_id = 1; // same as demo application's colormap file 1
      const mi::neuraylib::Tag colormap_tag =
        create_colormap(colormap_entry_id, scene.get(), dice_transaction);
      check_success(colormap_tag.is_valid());
      static_group->append(colormap_tag, dice_transaction);
    }

    static_group->append(svol_scene_element_tag, dice_transaction);
   const mi::neuraylib::Tag static_group_tag = dice_transaction->store_for_reference_counting(stati
    check_success(static_group_tag.is_valid());

    // Add the new volume to the hierarchical scene description
    scene->append(static_group_tag, dice_transaction);

    return svol_scene_element_tag;
  }

  return mi::neuraylib::NULL_TAG;
}

std::vector<mi::Uint32> Distributed_sparse_volume_data::evaluate_sparse_volume_data_locality(
  const nv::index::IDistributed_data_locality* svol_data_locality,
```

```
  const mi::math::Bbox<mi::Sint32, 3>&         query_bound,
  const mi::neuraylib::Tag&                    scene_element_tag,
  const std::string&                           scene_element_type) const
{
 // Determine all host in the cluster that store part of the distributed data (here: sparse volume).
  std::vector<mi::Uint32> cluster_host_ids;
  std::ostringstream hosts;
  std::ostringstream locality_report;
  for(mi::Uint32 i = 0; i < svol_data_locality->get_nb_cluster_nodes(); ++i)
  {
   // A host id=0 indicates that an issue occurred, e.g., no data has been loaded or no distribution sch
    check_success(svol_data_locality->get_cluster_node(i) != 0);

    const mi::Uint32 host_id = svol_data_locality->get_cluster_node(i);
    cluster_host_ids.push_back(host_id);

    // The remaining part is just for reporting/logging.
    hosts << host_id << " ";

    const mi::Size nb_subregions = svol_data_locality->get_nb_bounding_box(host_id);
   locality_report << "Host " << host_id << " stores part of the distributed data in the following " <<
    for(mi::Uint32 j = 0; j < nb_subregions; ++j)
    {
    const mi::math::Bbox<mi::Sint32, 3> bbox = svol_data_locality->get_bounding_box(host_id, j);
      locality_report << "  " << (j+1) << ")\t" << bbox;
      if(j < nb_subregions-1)
      {
        locality_report << "\n";
      }
    }
  }
  INFO_LOG << "\n"
      << "Data locality for a " << scene_element_type
      << " with tag id "        << scene_element_tag.id
      << " and query region "   << query_bound << "\n"
    << "The distributed data is located on the following hosts (ids): [ " << hosts.str() << "]." << "\n
      << locality_report.str()
      << "\n";

  return cluster_host_ids;
}

void Distributed_sparse_volume_data::get_brick_clamped_bbox(
  const nv::index::ISparse_volume_subset_data_descriptor* svol_data_subset_desc,
  const mi::Uint32                                        brick_idx,
  const mi::math::Vector<mi::Sint32, 3>&                  brick_position,
  const mi::math::Bbox<mi::Sint32, 3>&                    query_bbox_s32,
  mi::math::Bbox<mi::Sint32, 3>&                       out_brick_box_subdata_clamped,
  mi::math::Bbox<mi::Sint32, 3>&                         out_brick_box_clamped) const
{
  check_success(svol_data_subset_desc != 0);

  const mi::math::Vector<mi::Sint32, 3> query_bbox_ext_s32 = query_bbox_s32.extent();

 const mi::math::Vector<mi::Uint32,3> svol_data_brick_dim = svol_data_subset_desc->get_subset_data_
 const mi::Uint32 svol_data_brick_border            = svol_data_subset_desc->get_subset_data_brick
```

```
  const mi::math::Vector<mi::Sint32, 3> brick_pos_vol       = mi::math::Vector<mi::Sint32, 3>(brick_

  const mi::math::Bbox<mi::Sint32, 3> brick_box_global      = mi::math::Bbox<mi::Sint32, 3>(brick_p
                                brick_pos_vol + mi::math::Vector<mi::Sint32, 3>(svol_data_brick_di
                                - 2 * mi::math::Vector<mi::Sint32, 3>(svol_data_brick_border));
  const mi::math::Bbox<mi::Sint32, 3> brick_box_subdata     = mi::math::Bbox<mi::Sint32, 3>(brick_b
                                      brick_box_global.max - query_bbox_s32.min);
   // max clamp with [[0,0,0], ext.size()]
  const mi::math::Bbox<mi::Sint32, 3> brick_box_subdata_clamped = mi::math::Bbox<mi::Sint32, 3>(mi::
                                mi::math::clamp(brick_box_subdata.max, mi::math::Vector<mi::Sint3
  const mi::math::Bbox<mi::Sint32, 3> brick_box_global_clamped  = mi::math::Bbox<mi::Sint32, 3>(bric
                                brick_box_subdata_clamped.max + query_bbox_s32.min);
  const mi::math::Bbox<mi::Sint32, 3> brick_box_clamped     = mi::math::Bbox<mi::Sint32, 3>(brick_b
                                     brick_box_global_clamped.max - brick_pos_vol);

   out_brick_box_subdata_clamped = brick_box_subdata_clamped;
   out_brick_box_clamped         = brick_box_clamped;
}

mi::Float32 Distributed_sparse_volume_data::analyze_sparse_volume_data(
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  mi::base::Handle<const nv::index::ISession> the_session(dice_transaction->access<nv::index::ISess
   check_success(the_session.is_valid_interface());

  mi::base::Handle<const nv::index::ISparse_volume_scene_element> sparse_volume(dice_transaction->a
   check_success(sparse_volume.is_valid_interface());

   const mi::math::Vector<mi::Sint32, 2> trace(50, 50);
   const mi::math::Bbox<mi::Sint32, 3> query_bbox_s32(
     trace.x, trace.y, 0,
    trace.x + 1, trace.y + 1, static_cast<mi::Uint32>(sparse_volume->get_bounding_box().max.z));
   const mi::math::Bbox<mi::Float32, 3> query_bbox_f32(query_bbox_s32);

   // Access the distribution scheme
   const mi::neuraylib::Tag dist_layout_tag = the_session->get_distribution_layout();
   check_success(dist_layout_tag.is_valid());

  mi::base::Handle<const nv::index::IData_distribution> distribution_layout(dice_transaction->acces
   check_success(distribution_layout.is_valid_interface());

   // Find out on which hosts the data is located and print this information
   mi::base::Handle<nv::index::IDistributed_data_locality> svol_data_locality(
    distribution_layout->get_data_locality<nv::index::ISparse_volume_scene_element>(m_sparse_volume
   check_success(svol_data_locality.is_valid_interface());

   // Determine all host in the cluster that store part of the distributed data.
  std::vector<mi::Uint32> cluster_host_ids = evaluate_sparse_volume_data_locality(svol_data_localit

   // Create the access factory
   const mi::neuraylib::Tag data_access_tag = the_session->get_data_access_factory();
   check_success(data_access_tag.is_valid());

  mi::base::Handle<const nv::index::IDistributed_data_access_factory> svol_access_factory(
    dice_transaction->access<nv::index::IDistributed_data_access_factory>(data_access_tag));
   check_success(svol_access_factory.is_valid_interface());
```

```
 mi::base::Handle<nv::index::IDistributed_data_access> svol_data_access(
  svol_access_factory->create_distributed_data_access<nv::index::ISparse_volume_scene_element>(m_
 check_success(svol_data_access.is_valid_interface());

 // Now retrieve the data
 check_success(svol_data_access->access(query_bbox_f32, dice_transaction) >= 0);

const mi::math::Bbox<mi::Float32, 3> effective_bbox_f32 = svol_data_access->get_bounding_box();
 check_success(effective_bbox_f32 == query_bbox_f32);

 mi::Float32 avg = -1.0f;

mi::base::Handle<const nv::index::IDistributed_data_subset>           data_subset(svol_data_acce
mi::base::Handle<const nv::index::ISparse_volume_subset>             svol_data_subset(data_subse
 check_success(svol_data_subset.is_valid_interface());

mi::base::Handle<const nv::index::ISparse_volume_attribute_set_descriptor> svol_data_attrib_desc
mi::base::Handle<const nv::index::ISparse_volume_subset_data_descriptor>   svol_data_subset_desc(

 const mi::Uint32 svol_attrib_index_0 = 0u;
nv::index::ISparse_volume_attribute_set_descriptor::Attribute_parameters svol_attrib_param_0;
check_success(svol_data_attrib_desc->get_attribute_parameters(svol_attrib_index_0, svol_attrib_pa

const nv::index::Sparse_volume_voxel_format svol_voxel_fmt    = svol_attrib_param_0.format;
const mi::Sint32                   svol_voxel_fmt_size = nv::index::get_sizeof(svol_voxel_fmt);

const mi::math::Vector<mi::Sint32, 3> query_bbox_ext_s32 = query_bbox_s32.extent(); // size (= quer

 const mi::Size svol_query_buffer_size =
   static_cast<mi::Size>(query_bbox_ext_s32.x) *
   static_cast<mi::Size>(query_bbox_ext_s32.y) *
   static_cast<mi::Size>(query_bbox_ext_s32.z) *
   svol_voxel_fmt_size;

 mi::Uint8* svol_query_buffer = new mi::Uint8[svol_query_buffer_size];

const mi::math::Vector<mi::Uint32,3> svol_data_brick_dim = svol_data_subset_desc->get_subset_data_
const mi::Uint32 svol_data_brick_border            = svol_data_subset_desc->get_subset_data_brick
const mi::Uint32 svol_subset_nb_bricks             = svol_data_subset_desc->get_subset_number_of_
 for (mi::Uint32 b = 0u; b < svol_subset_nb_bricks; ++b)
 {
  const nv::index::ISparse_volume_subset_data_descriptor::Data_brick_info brick_info    = svol_data
  const nv::index::ISparse_volume_subset::Data_brick_buffer_info       brick_data_nfo = svol_data_

  const mi::Uint8* svol_brick_data_raw = reinterpret_cast<mi::Uint8*>(brick_data_nfo.data);
   if (svol_brick_data_raw == NULL)
   {
     ERROR_LOG << "error accessing brick data pointer "
          << "(brick_idx: "    << b
          << ", brick_level: " << brick_info.brick_lod_level << ").";
     delete [] svol_query_buffer;
     check_success(false);
   }

   mi::math::Bbox<mi::Sint32, 3> brick_box_subdata_clamped;
   mi::math::Bbox<mi::Sint32, 3> brick_box_clamped;
  get_brick_clamped_bbox(svol_data_subset_desc.get(), b, brick_info.brick_position, query_bbox_s32
```

```
 for (mi::Uint32 z = 0u; z < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().z); ++z)
  {
  for (mi::Uint32 y = 0u; y < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().y); ++y)
    {
      const mi::math::Vector<mi::Sint32, 3> line_pos_dst =
        mi::math::Vector<mi::Sint32, 3>(brick_box_subdata_clamped.min.x,
                        brick_box_subdata_clamped.min.y + y,
                        brick_box_subdata_clamped.min.z + z);

    const mi::Size dst_off = calc_offset(line_pos_dst, mi::math::Vector<mi::Uint32, 3>(query_bbox_

      const mi::math::Vector<mi::Sint32, 3> line_pos_src =
        mi::math::Vector<mi::Sint32, 3>(brick_box_clamped.min.x,
                        brick_box_clamped.min.y + y,
                        brick_box_clamped.min.z + z);
    const mi::Size src_off =  calc_offset(line_pos_src, svol_data_brick_dim, svol_data_brick_borde

    const mi::Size line_size = static_cast<mi::Size>(brick_box_subdata_clamped.extent().x) * svol_

      memcpy(svol_query_buffer   + dst_off * svol_voxel_fmt_size,
          svol_brick_data_raw + src_off * svol_voxel_fmt_size,
          line_size);
    }
  }
}

if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_UINT8)
{
  // Average the contents of the trace
  const mi::Uint8* voxel_data = svol_query_buffer;

  mi::Sint64 sum = 0;
 mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min.z);
  for (mi::Sint64 k = 0; k < count; ++k)
  {
    sum += static_cast<mi::Sint64>(voxel_data[k]);
  }

  avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(count);
  INFO_LOG << "Tracing 8-bit volume: [" << trace.x << ", " << trace.y
      << ", " << query_bbox_s32.min.z << ":" << query_bbox_s32.max.z - 1 << "]: "
      << "count=" << count << ", sum=" << sum << ", avg=" << avg;
}
else if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_UINT8_4)
{
  // Average the contents of the trace
  const mi::math::Vector_struct<mi::Uint8, 4>* voxel_data =
    reinterpret_cast< mi::math::Vector_struct<mi::Uint8, 4>*>(svol_query_buffer);

  mi::Sint64 sum = 0;
 mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min.z);
  for (mi::Sint64 k = 0; k < count; ++k)
  {
    mi::math::Vector<mi::Uint8, 4> voxel_v(voxel_data[k]);
    mi::math::Vector<mi::Float32, 4> voxel_f32_4(voxel_v);
  sum += static_cast<mi::Sint64>(((voxel_f32_4.x + voxel_f32_4.y + voxel_f32_4.z) / 3.0f));
```

```
    }

    avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(count);
    INFO_LOG << "Tracing rgba8 volume: [" << trace.x << ", " << trace.y
        << ", " << query_bbox_s32.min.z << ":" << query_bbox_s32.max.z - 1 << "]: "
        << "count=" << count << ", sum=" << sum << ", avg=" << avg;
  }
  else if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_SINT16)
  {
    // Average the contents of the trace
    const mi::Sint16* voxel_data = reinterpret_cast<mi::Sint16*>(svol_query_buffer);

    mi::Sint64 sum = 0;
   mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min.z);
    for (mi::Sint64 k = 0; k < count; ++k)
    {
      sum += voxel_data[k];
    }

    avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(count);
    INFO_LOG << "Tracing sint16 volume: [" << trace.x << ", " << trace.y
        << ", " << query_bbox_s32.min.z << ":" << query_bbox_s32.max.z - 1 << "]: "
        << "count=" << count << ", sum=" << sum << ", avg=" << avg;
  }
  else if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_FLOAT32)
  {
    // Average the contents of the trace
   const mi::Float32* voxel_data = reinterpret_cast<mi::Float32*>(svol_query_buffer);

    mi::Float32 sum = 0;
   mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min.z);
    for (mi::Sint64 k = 0; k < count; ++k)
    {
      sum += voxel_data[k];
    }

    avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(count);
    INFO_LOG << "Tracing uint16 volume: [" << trace.x << ", " << trace.y
        << ", " << query_bbox_s32.min.z << ":" << query_bbox_s32.max.z - 1 << "]: "
        << "count=" << count << ", sum=" << sum << ", avg=" << avg;
  }
  else
  {
    ERROR_LOG << "Analyze volume data failed: data access on not-known volume type.";
  }

  return avg;
}

void Distributed_sparse_volume_data::edit_sparse_volume_data(
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  mi::base::Handle<const nv::index::ISession> the_session(
    dice_transaction->access<nv::index::ISession>(m_session_tag));
  check_success(the_session.is_valid_interface());

  mi::base::Handle<const nv::index::ISparse_volume_scene_element> sparse_volume(
```

```
   dice_transaction->access<nv::index::ISparse_volume_scene_element>(m_sparse_volume_tag));
  check_success(sparse_volume.is_valid_interface());

  const mi::math::Vector<mi::Sint32, 2> trace(50, 50);
  const mi::math::Bbox<mi::Sint32, 3> query_bbox_s32(
    trace.x, trace.y, 0,
   trace.x + 1, trace.y + 1, static_cast<mi::Sint32>(sparse_volume->get_bounding_box().max.z));
 INFO_LOG << "The sparse volume will be edited inside the following 3D bounding box: " << query_bbox_s3
  const mi::math::Bbox<mi::Float32, 3> query_bbox_f32(query_bbox_s32);

  // Access the distribution scheme
  const mi::neuraylib::Tag dist_layout_tag = the_session->get_distribution_layout();
  check_success(dist_layout_tag.is_valid());

 mi::base::Handle<const nv::index::IData_distribution> distribution_layout(dice_transaction->acces
  check_success(distribution_layout.is_valid_interface());

  // Distribution layout
  mi::base::Handle<nv::index::IDistributed_data_locality> svol_data_locality(
   distribution_layout->get_data_locality<nv::index::ISparse_volume_scene_element>(m_sparse_volume
  check_success(svol_data_locality.is_valid_interface());

 // Determine all host in the cluster that store part of the distributed data (here: sparse volume).
  std::vector<mi::Uint32> cluster_host_ids = evaluate_sparse_volume_data_locality(
   svol_data_locality.get(), query_bbox_s32, m_sparse_volume_tag, sparse_volume->get_class_name());

  // Set up distributed algorithm and start the job
  const mi::Sint32 voxel_value   = 42;    // just some "arbitrary" number

  // Access an application layer component that provides some sample distributed jobs:
 mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> pro
   get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_processi
  check_success(processing.is_valid_interface());
  // Create a job that distributed to all nodes/GPUs and add edits the volume data.
  // For more details on how to implement the mandelbrot, please review the provided
 // source that was shipped with the application layer component 'nv::index::app::data_analysis_and_p
 mi::base::Handle<nv::index::IDistributed_data_job> algo(processing->get_sample_tool_set()->create
      m_sparse_volume_tag,
      voxel_value,
      query_bbox_s32));
  distribution_layout->create_scheduler()->execute(algo.get(), dice_transaction);
}

mi::Float32 Distributed_sparse_volume_data::export_sparse_volume_data(
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
 mi::base::Handle<const nv::index::ISession> the_session(dice_transaction->access<nv::index::ISess
  check_success(the_session.is_valid_interface());

 mi::base::Handle<const nv::index::ISparse_volume_scene_element> sparse_volume(dice_transaction->a
  check_success(sparse_volume.is_valid_interface());

  // Select what subset of the volume should be exported
  mi::math::Bbox<mi::Sint32, 3> query_bbox_s32(sparse_volume->get_bounding_box());
  query_bbox_s32.min += (query_bbox_s32.max - query_bbox_s32.min) / 4;
  query_bbox_s32.max -= (query_bbox_s32.max - query_bbox_s32.min) / 4;
  const mi::math::Bbox<mi::Float32, 3> query_bbox_f32(query_bbox_s32);
```

```
 // Access the distribution scheme
 const mi::neuraylib::Tag dist_layout_tag = the_session->get_distribution_layout();
 check_success(dist_layout_tag.is_valid());

mi::base::Handle<const nv::index::IData_distribution> distribution_layout(dice_transaction->acces
 check_success(distribution_layout.is_valid_interface());

 // Find out on which hosts the data is located and print this information
 mi::base::Handle<nv::index::IDistributed_data_locality> svol_data_locality(
  distribution_layout->get_data_locality<nv::index::ISparse_volume_scene_element>(m_sparse_volume
 check_success(svol_data_locality.is_valid_interface());

 // Determine all host in the cluster that store part of the distributed data.
std::vector<mi::Uint32> cluster_host_ids = evaluate_sparse_volume_data_locality(svol_data_localit

 // Create the access factory
 const mi::neuraylib::Tag data_access_tag = the_session->get_data_access_factory();
 check_success(data_access_tag.is_valid());

mi::base::Handle<const nv::index::IDistributed_data_access_factory> svol_access_factory(
  dice_transaction->access<nv::index::IDistributed_data_access_factory>(data_access_tag));
 check_success(svol_access_factory.is_valid_interface());

 mi::base::Handle<nv::index::IDistributed_data_access> svol_data_access(
  svol_access_factory->create_distributed_data_access<nv::index::ISparse_volume_scene_element>(m_
 check_success(svol_data_access.is_valid_interface());

 // Now retrieve the data
 check_success(svol_data_access->access(query_bbox_f32, dice_transaction) >= 0);

const mi::math::Bbox<mi::Float32, 3> effective_bbox_f32 = svol_data_access->get_bounding_box();
 check_success(effective_bbox_f32 == query_bbox_f32);

 mi::Float32 avg = -1.0f;

mi::base::Handle<const nv::index::IDistributed_data_subset>           data_subset(svol_data_acce
mi::base::Handle<const nv::index::ISparse_volume_subset>             svol_data_subset(data_subse
 check_success(svol_data_subset.is_valid_interface());

mi::base::Handle<const nv::index::ISparse_volume_attribute_set_descriptor>  svol_data_attrib_desc
mi::base::Handle<const nv::index::ISparse_volume_subset_data_descriptor>   svol_data_subset_desc(

 const mi::Uint32 svol_attrib_index_0 = 0u;
nv::index::ISparse_volume_attribute_set_descriptor::Attribute_parameters svol_attrib_param_0;
check_success(svol_data_attrib_desc->get_attribute_parameters(svol_attrib_index_0, svol_attrib_pa

const nv::index::Sparse_volume_voxel_format svol_voxel_fmt    = svol_attrib_param_0.format;
const mi::Sint32                   svol_voxel_fmt_size = nv::index::get_sizeof(svol_voxel_fmt);

// const mi::math::Vector<mi::Sint32, 3> query_bbox_ext_s32 = query_bbox_s32.extent(); // size (= qu
const mi::math::Vector<mi::Uint32,3> svol_data_brick_dim = svol_data_subset_desc->get_subset_data_
const mi::Uint32 svol_data_brick_border            = svol_data_subset_desc->get_subset_data_brick
const mi::Uint32 svol_subset_nb_bricks             = svol_data_subset_desc->get_subset_number_of_

 mi::Sint64 sum       = 0;
 mi::Sint64 nb_voxels = 0;
```

```
if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_UINT8)
{
  for (mi::Uint32 b = 0u; b < svol_subset_nb_bricks; ++b)
  {
  const nv::index::ISparse_volume_subset_data_descriptor::Data_brick_info brick_info    = svol_da
  const nv::index::ISparse_volume_subset::Data_brick_buffer_info       brick_data_nfo = svol_data
  const mi::Uint8* svol_brick_data_raw = reinterpret_cast<mi::Uint8*>(brick_data_nfo.data);
    check_success (svol_brick_data_raw != 0);

    mi::math::Bbox<mi::Sint32, 3> brick_box_subdata_clamped;
    mi::math::Bbox<mi::Sint32, 3> brick_box_clamped;
  get_brick_clamped_bbox(svol_data_subset_desc.get(), b, brick_info.brick_position, query_bbox_s
  for (mi::Uint32 z = 0u; z < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().z); ++z)
    {
   for (mi::Uint32 y = 0u; y < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().y); ++y)
      {
        const mi::math::Vector<mi::Sint32, 3> line_pos_src =
          mi::math::Vector<mi::Sint32, 3>(brick_box_clamped.min.x,
                          brick_box_clamped.min.y + y,
                          brick_box_clamped.min.z + z);
     const mi::Size src_off =  calc_offset(line_pos_src, svol_data_brick_dim, svol_data_brick_bord

        // Average the contents with trace by trace
     const mi::Uint8* voxel_data = svol_brick_data_raw + src_off * svol_voxel_fmt_size;
     mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min
        for (mi::Sint64 k = 0; k < count; ++k)
        {
          sum += static_cast<mi::Sint64>(voxel_data[k]);
        }
        nb_voxels += count;
      }
    }
  }
  avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(nb_voxels);
 INFO_LOG << "Average voxel value in exported data: " << avg << " of " << nb_voxels;
}
else if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_UINT8_4)
{
  for (mi::Uint32 b = 0u; b < svol_subset_nb_bricks; ++b)
  {
  const nv::index::ISparse_volume_subset_data_descriptor::Data_brick_info brick_info    = svol_da
  const nv::index::ISparse_volume_subset::Data_brick_buffer_info       brick_data_nfo = svol_data
  const mi::Uint8* svol_brick_data_raw = reinterpret_cast<mi::Uint8*>(brick_data_nfo.data);
    check_success (svol_brick_data_raw != 0);

    mi::math::Bbox<mi::Sint32, 3> brick_box_subdata_clamped;
    mi::math::Bbox<mi::Sint32, 3> brick_box_clamped;
  get_brick_clamped_bbox(svol_data_subset_desc.get(), b, brick_info.brick_position, query_bbox_s
  for (mi::Uint32 z = 0u; z < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().z); ++z)
    {
   for (mi::Uint32 y = 0u; y < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().y); ++y)
      {
        const mi::math::Vector<mi::Sint32, 3> line_pos_src =
          mi::math::Vector<mi::Sint32, 3>(brick_box_clamped.min.x,
                          brick_box_clamped.min.y + y,
                          brick_box_clamped.min.z + z);
```

```
      const mi::Size src_off =  calc_offset(line_pos_src, svol_data_brick_dim, svol_data_brick_bord

         // Average the contents with trace by trace
         const mi::math::Vector_struct<mi::Uint8, 4>* voxel_data =
       reinterpret_cast<const  mi::math::Vector_struct<mi::Uint8, 4>* >(svol_brick_data_raw + src_
      mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min
         for (mi::Sint64 k = 0; k < count; ++k)
         {
           sum += (static_cast<mi::Sint64>(voxel_data[k].x) +
               static_cast<mi::Sint64>(voxel_data[k].y) +
               static_cast<mi::Sint64>(voxel_data[k].z)) / 3;
         }
         nb_voxels += count;
       }
      }
    }
    avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(nb_voxels);
   INFO_LOG << "Average voxel value in exported data: " << avg << " of " << nb_voxels;
  }
  else if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_SINT16)
  {
    for (mi::Uint32 b = 0u; b < svol_subset_nb_bricks; ++b)
    {
    const nv::index::ISparse_volume_subset_data_descriptor::Data_brick_info brick_info    = svol_da
    const nv::index::ISparse_volume_subset::Data_brick_buffer_info       brick_data_nfo = svol_data
    const mi::Uint8* svol_brick_data_raw = reinterpret_cast<mi::Uint8*>(brick_data_nfo.data);
      check_success (svol_brick_data_raw != 0);

      mi::math::Bbox<mi::Sint32, 3> brick_box_subdata_clamped;
      mi::math::Bbox<mi::Sint32, 3> brick_box_clamped;
    get_brick_clamped_bbox(svol_data_subset_desc.get(), b, brick_info.brick_position, query_bbox_s
    for (mi::Uint32 z = 0u; z < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().z); ++z)
      {
     for (mi::Uint32 y = 0u; y < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().y); ++y)
        {
          const mi::math::Vector<mi::Sint32, 3> line_pos_src =
            mi::math::Vector<mi::Sint32, 3>(brick_box_clamped.min.x,
                            brick_box_clamped.min.y + y,
                            brick_box_clamped.min.z + z);
      const mi::Size src_off =  calc_offset(line_pos_src, svol_data_brick_dim, svol_data_brick_bord

         // Average the contents with trace by trace
      const mi::Sint16* voxel_data = reinterpret_cast<const mi::Sint16*>(svol_brick_data_raw + src_
      mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min
         for (mi::Sint64 k = 0; k < count; ++k)
         {
           sum += voxel_data[k];
         }
         nb_voxels += count;
       }
      }
    }
    avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(nb_voxels);
   INFO_LOG << "Average voxel value in exported data: " << avg << " of " << nb_voxels;
  }
  else if(svol_voxel_fmt == nv::index::SPARSE_VOLUME_VOXEL_FORMAT_FLOAT32)
  {
```

```
    for (mi::Uint32 b = 0u; b < svol_subset_nb_bricks; ++b)
    {
    const nv::index::ISparse_volume_subset_data_descriptor::Data_brick_info brick_info    = svol_da
    const nv::index::ISparse_volume_subset::Data_brick_buffer_info       brick_data_nfo = svol_data
    const mi::Uint8* svol_brick_data_raw = reinterpret_cast<mi::Uint8*>(brick_data_nfo.data);
      check_success (svol_brick_data_raw != 0);

      mi::math::Bbox<mi::Sint32, 3> brick_box_subdata_clamped;
      mi::math::Bbox<mi::Sint32, 3> brick_box_clamped;
    get_brick_clamped_bbox(svol_data_subset_desc.get(), b, brick_info.brick_position, query_bbox_s
    for (mi::Uint32 z = 0u; z < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().z); ++z)
      {
     for (mi::Uint32 y = 0u; y < static_cast<mi::Uint32>(brick_box_subdata_clamped.extent().y); ++y)
        {
          const mi::math::Vector<mi::Sint32, 3> line_pos_src =
            mi::math::Vector<mi::Sint32, 3>(brick_box_clamped.min.x,
                            brick_box_clamped.min.y + y,
                            brick_box_clamped.min.z + z);
       const mi::Size src_off =  calc_offset(line_pos_src, svol_data_brick_dim, svol_data_brick_bord

          // Average the contents with trace by trace
          const mi::Float32* voxel_data =
        reinterpret_cast<const mi::Float32*>(svol_brick_data_raw + src_off * svol_voxel_fmt_size);
       mi::Sint64 count = static_cast<mi::Sint64>(effective_bbox_f32.max.z - effective_bbox_f32.min
          for (mi::Sint64 k = 0; k < count; ++k)
          {
            sum += static_cast<mi::Sint64>(voxel_data[k]);
          }
          nb_voxels += count;
        }
      }
    }
    avg = static_cast<mi::Float32>(sum) / static_cast<mi::Float32>(nb_voxels);
   INFO_LOG << "Average voxel value in exported data: " << avg << " of " << nb_voxels;
  }
  else
  {
    ERROR_LOG << "unknown voxel_format: " << svol_voxel_fmt;
  }
  return avg;
}

void Distributed_sparse_volume_data::render_frame(
  const std::string& output_filename) const
{
  // Rendering to a file by means of the file canvas
 m_image_file_canvas->set_rgba_file_name(output_filename.c_str()); // No output if empty string

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
```

```
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                                  // log level
  sdict.insert("unittest", "0");                                       // default mode
 sdict.insert("outfname", "frame_distributed_sparse_volume_data"); // output file base name
  sdict.insert("verify_image_fname", "");                              // for unit test
 sdict.insert("dice::network::multicast_address", "224.1.3.2");   // default multicast address
 sdict.insert("dice::network::cluster_interface", "");         // default cluster interface address
 sdict.insert("voxel_format", "uint8");                       // default voxel_format
 sdict.insert("is_dump_comparison_image_when_failed", "1");     // default: dump images when failed
 sdict.insert("is_call_from_test", "0");                  // default: not call from make check.

  // DiCE settings
  sdict.insert("dice::network::mode", "OFF");

  // IndeX settings
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // Application_layer settings
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io data_analysis_and_processing");
  sdict.insert("index::app::plugins::base_importer::enabled", "true");

  // Initialize the NVIDIA IndeX library
  Distributed_sparse_volume_data distributed_data_example;
  distributed_data_example.initialize(argc, argv, sdict);
  check_success(distributed_data_example.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = distributed_data_example.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.19 dynamic_plane.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/iscene_group.h>
#include <nv/index/isession.h>
#include <nv/index/iplane.h>
#include <nv/index/itexture_filter_mode.h>

#include "utility/canvas_utility.h"

#include <nv/index/app/idata_analysis_and_processing.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>

#include <iostream>
#include <sstream>

class Dynamic_plane:
  public nv::index::app::Index_connect
{
public:
  Dynamic_plane()
    :
    Index_connect(),
    m_is_unittest(false),
    m_max_iter(0)
  {
    // INFO_LOG << "DEBUG: Dynamic_plane() ctor";
  }

  virtual ~Dynamic_plane()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Dynamic_plane() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
```

```
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // create plane instances
  void create_planes(nv::index::IScene*                scene_edit,
            mi::neuraylib::IDice_transaction* dice_transaction);
  // move planes
  // \param[in] dice_transaction dice transaction
  void move_planes(mi::neuraylib::IDice_transaction* dice_transaction) const;
  // setup camera to see this example scene
  // \param[in] cam    a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;
  // render a frame
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                  m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                    m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                            m_outfname;
  bool                                                                m_is_unittest;
  std::string                                                  m_verify_image_fname;
  mi::Sint32                                                             m_max_iter;
  // plane tags
  std::vector<mi::neuraylib::Tag>                                   m_plane_tag_vec;
};

mi::Sint32 Dynamic_plane::launch()
{
  mi::Sint32 exit_code = 0;

  // Get DiCE database components
  {
    m_cluster_configuration =
      get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
```

```
m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
 check_success(m_image_file_canvas.is_valid_interface());

 // Verifying that local host has joined
 const mi::Sint32 max_retry = 3;
 for (mi::Sint32 retry = 0; retry < max_retry; ++retry)
 {
   if (m_cluster_configuration->get_number_of_hosts() == 0)
   {
     INFO_LOG << "no host joined yet, retry "
         << (retry + 1) << "/" << max_retry;
     nv::index::app::util::time::sleep(0.3f);
   }
 }
 check_success(m_cluster_configuration->get_number_of_hosts() != 0);

 {
   // Obtain a DiCE transaction
   mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
     m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
   check_success(dice_transaction.is_valid_interface());
   {
     // Setup session information
     m_session_tag =
       m_index_session->create_session(dice_transaction.get());
     check_success(m_session_tag.is_valid());

     mi::base::Handle<const nv::index::ISession> session(
       dice_transaction->access<nv::index::ISession>(
         m_session_tag));
     check_success(session.is_valid_interface());

     mi::base::Handle< nv::index::IScene > scene_edit(
       dice_transaction->edit<nv::index::IScene>(session->get_scene()));
     check_success(scene_edit.is_valid_interface());

     //------------------------------------------------------------------------
     // Scene setup: add textured planes
     //------------------------------------------------------------------------
     create_planes(scene_edit.get(), dice_transaction.get());

     mi::base::Handle< nv::index::IPerspective_camera > cam(
       scene_edit->create_camera<nv::index::IPerspective_camera>());
     check_success(cam.is_valid_interface());
     setup_camera(cam.get());
     const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
     check_success(camera_tag.is_valid());

     const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
     m_image_file_canvas->set_resolution(buffer_resolution);

     // Set up the scene and define the region of interest
     const mi::math::Bbox_struct<mi::Float32, 3> xyz_roi_st = {
       { -1000.0f, -1000.0f, -1000.0f, },
       {  1000.0f,  1000.0f,  1000.0f, },
     };
```

```
      // scope for scene edit
      {
        mi::base::Handle< nv::index::IScene > scene(
          dice_transaction->edit< nv::index::IScene >(session->get_scene()));
        check_success(scene.is_valid_interface());

        // set the region of interest
        const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
        check_success(xyz_roi.is_volume());
        scene->set_clipped_bounding_box(xyz_roi_st);

        // Set the scene global transformation matrix.
        // only change the coordinate system
        mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
          1.0f,  0.0f,  0.0f,  0.0f,
          0.0f,  1.0f,  0.0f,  0.0f,
          0.0f,  0.0f, -1.0f,  0.0f,
          0.0f,  0.0f,  0.0f,  1.0f
          );
        scene->set_transform_matrix(transform_mat);

        // Set the current camera to the scene.
        check_success(camera_tag.is_valid());
        scene->set_camera(camera_tag);
      }
    }
    dice_transaction->commit();
  }

  check_success(m_max_iter > 0);

  // Render as many frame as requested by the user
  for (mi::Sint32 i = 0; i < m_max_iter; ++i)
  {
    // Render the frame to a file
    const std::string fname = get_output_file_name(m_outfname, i);
   mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
  const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    if (err_set->any_errors())
    {
      std::ostringstream os;
      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }

   ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
          << os.str();
      exit_code = 1;
      break;
    }

    // verify the generated frame
    if (!(verify_canvas_result(get_application_layer_interface(),
```

```
                    m_image_file_canvas.get(), m_verify_image_fname, get_options())))
      {
        exit_code = 1;
        break;
      }

      // Move the plane a bit
      {
        mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
          m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
        check_success(dice_transaction.is_valid_interface());
        {
          move_planes(dice_transaction.get());
        }
        dice_transaction->commit();
      }
    }
  }
  return exit_code;
}

bool Dynamic_plane::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
    sdict.insert("max_iter", "1");
  }

  m_outfname = sdict.get("outfname");
  m_max_iter = nv::index::app::get_sint32(sdict.get("max_iter"));

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["       + m_outfname +
       "], max_iter = "     + nv::index::app::to_string(m_max_iter) +
       ", dice::verbose = " + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-max_iter number_of_iteration]\n"
```

```
      << "            set rendering loop iterations. (default: "
      << m_max_iter << " frames)\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_fname [image_fname]]\n"
      << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode. "
      << m_is_unittest << "])"
      << std::endl;
    exit(1);
  }
  return true;
}

void Dynamic_plane::create_planes(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(dice_transaction != 0);

  // Add a scene group where the shapes should be added
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Add a light and a material
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
    headlight->set_intensity(color_intensity);
    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());
    group_node->append(headlight_tag, dice_transaction);

    // Add a fully ambient material for the planes, so that lighting doesn't matter
    mi::base::Handle<nv::index::IPhong_gl> phong_1(
      scene_edit->create_attribute<nv::index::IPhong_gl>());
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(mi::math::Color(1.0f, 1.0f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.0f));
    phong_1->set_specular(mi::math::Color(0.0f));

   mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.get());
    check_success(phong_1_tag.is_valid());
    group_node->append(phong_1_tag, dice_transaction);
```

```
  check_success(m_plane_tag_vec.empty()); // planes have not yet been created
}

//
// Plane 1: RGBA mode, axis aligned, no filtering
//
{
  // Place plane centered in the origin, normal pointing along the z-axis
  mi::math::Vector<mi::Float32, 3> plane_point(-400.f, -250.f, 0.0f);
  mi::math::Vector<mi::Float32, 3> plane_normal(0.f, 0.f, 1.f);
  mi::math::Vector<mi::Float32, 3> plane_up(0.f, 1.f, 0.f);
  mi::math::Vector<mi::Float32, 2> plane_extent(800.0f, 500.0f);

  mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
  scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
  check_success(tex_filter.is_valid_interface());
 mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.ge
  check_success(tex_filter_tag.is_valid());
  group_node->append(tex_filter_tag, dice_transaction);

 // Access an application layer component that provides some sample techniques such as the mandelbro
 mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> p
  get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_proces
  check_success(processing.is_valid_interface());
 // Create a mandelbrot technique and add it to the scene. For more details on how to implement the ma
 // source that was shipped with the application layer component 'nv::index::app::data_analysis_and
  mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
    processing->get_sample_tool_set()->create_mandelbrot_2d_technique(
   plane_extent, nv::index::IDistributed_compute_destination_buffer_2d_texture::FORMAT_RGBA_FLO
  check_success(mapping.is_valid_interface());
 mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.get());
  check_success(mapping_tag.is_valid());
  group_node->append(mapping_tag, dice_transaction);

  mi::base::Handle<nv::index::IPlane> plane(
    scene_edit->create_shape<nv::index::IPlane>());
  check_success(plane.is_valid_interface());
  plane->set_point(plane_point);
  plane->set_normal(plane_normal);
  plane->set_up(plane_up);
  plane->set_extent(plane_extent);

 const mi::neuraylib::Tag plane_tag = dice_transaction->store_for_reference_counting(plane.get())
  check_success(plane_tag.is_valid());
  m_plane_tag_vec.push_back(plane_tag);

  group_node->append(plane_tag, dice_transaction);
}

//
// Plane 2: Color map mode, arbitrary orientation, use texture filtering
//
{
  mi::math::Vector<mi::Float32, 3> plane_point(-50.0f, 50.0f, 700.0f);
  mi::math::Vector<mi::Float32, 3> plane_normal(1.f, 1.f, 1.f);
  mi::math::Vector<mi::Float32, 3> plane_up(-1.f, 1.f, 0.f);
  mi::math::Vector<mi::Float32, 2> plane_extent(400.0f, 400.0f);
```

```
    const bool enable_texture_filtering = true;
   const mi::Sint32 colormap_entry_id = 40; // same as demo application's colormap file 40
    mi::neuraylib::Tag colormap_tag =
      create_colormap(colormap_entry_id, scene_edit, dice_transaction);
    check_success(colormap_tag.is_valid());
    group_node->append(colormap_tag, dice_transaction);

    mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter;

    if (enable_texture_filtering)
    {
    tex_filter = scene_edit->create_attribute<nv::index::ITexture_filter_mode_linear>();
    }
    else
    {
    tex_filter = scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>();
    }
    check_success(tex_filter.is_valid_interface());
   mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.ge
    check_success(tex_filter_tag.is_valid());
    group_node->append(tex_filter_tag, dice_transaction);

   // Access an application layer component that provides some sample techniques such as the mandelbro
   mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> p
   get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_processi
    check_success(processing.is_valid_interface());
   // Create a mandelbrot technique and add it to the scene. For more details on how to implement the ma
   // source that was shipped with the application layer component 'nv::index::app::data_analysis_and
    mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
      processing->get_sample_tool_set()->create_mandelbrot_2d_technique(
     plane_extent, nv::index::IDistributed_compute_destination_buffer_2d_texture::FORMAT_SCALAR_U
    check_success(mapping.is_valid_interface());
   mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.get());
    check_success(mapping_tag.is_valid());
    group_node->append(mapping_tag, dice_transaction);

    mi::base::Handle<nv::index::IPlane> plane(
      scene_edit->create_shape<nv::index::IPlane>());
    check_success(plane.is_valid_interface());
    plane->set_point(plane_point);
    plane->set_normal(plane_normal);
    plane->set_up(plane_up);
    plane->set_extent(plane_extent);

   mi::neuraylib::Tag plane_tag = dice_transaction->store_for_reference_counting(plane.get());
    check_success(plane_tag.is_valid());
    m_plane_tag_vec.push_back(plane_tag);

    group_node->append(plane_tag, dice_transaction);
  }

  // Finally append everything to the root of the hierachical scene description
 mi::neuraylib::Tag group_tag = dice_transaction->store_for_reference_counting(group_node.get());
  check_success(group_tag.is_valid());
  scene_edit->append(group_tag, dice_transaction);
}
```

```
void Dynamic_plane::move_planes(mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(!m_plane_tag_vec.empty());

  for(mi::Size i = 0; i < m_plane_tag_vec.size(); ++i)
  {
    mi::base::Handle<nv::index::IPlane> plane(
      dice_transaction->edit<nv::index::IPlane>(m_plane_tag_vec.at(i)));
    check_success(plane.is_valid_interface());

    // Retrieve plane point
    mi::math::Vector<mi::Float32, 3> point  = plane->get_point();
    mi::math::Vector<mi::Float32, 3> normal = plane->get_normal();

    // Move it a bit into the direction of its normal vector
    point += normal * -5.f;

    plane->set_point(point);
  }
}

void Dynamic_plane::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene.
  const mi::math::Vector<mi::Float32, 3> from(100.0f, 0.0f, -1000.0f);
  const mi::math::Vector<mi::Float32, 3> to  (0.0f, 250.0f, 0.0f);
  const mi::math::Vector<mi::Float32, 3> up  (0.0f, 1.0f, 0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(80.0f);
  cam->set_clip_max(5000.0f);
}

nv::index::IFrame_results* Dynamic_plane::render_frame(const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
```

```
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
 sdict.insert("max_iter", "20");                     // default max rendering loop iterations
  sdict.insert("outfname", "frame_dynamic_plane");           // output file base name
  sdict.insert("verify_image_fname", "");                     // for unit test
  sdict.insert("unittest", "0");                              // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                // default: not call from make check.

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io data_analysis_and_processing");

  // Initialize application
  Dynamic_plane dynamic_plane;
  dynamic_plane.initialize(argc, argv, sdict);
  check_success(dynamic_plane.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = dynamic_plane.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.20 dynamic_point_set.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/ipoint_set.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include <iostream>
#include <sstream>

#include "utility/frame_command.h"

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>

class Dynamic_point_set:
  public nv::index::app::Index_connect
{
public:
  Dynamic_point_set()
    :
    Index_connect(),
    m_is_unittest(false),
    m_max_iter(0),
    m_resolution_x(0),
    m_resolution_y(0),
    m_roi_x(0),
    m_roi_y(0),
    m_roi_z(0),
    m_is_save_all_frame(false),
    m_is_animation_on(false)
  {
    // INFO_LOG << "DEBUG: Dynamic_point_set() ctor";
  }

  virtual ~Dynamic_point_set()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Dynamic_point_set() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
```

```
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // attribute to color map example
  // \param[in] attrib attribute value
  // \return color defined by an attribute
  mi::math::Color_struct get_color_st_from_attribute(mi::Sint32 attrib) const;
  // attribute to radius map example
  // \param[in] attrib attribute value
  // \return a radius corresponding to the attribute value
  mi::Float32 get_radius_from_attribute(mi::Sint32 attrib) const;
  // create point set in the scene.
  // \param[in]  session_tag session tag
  // \param[in]  dice_transaction dice transaction
  void create_point_set(
    const mi::neuraylib::Tag&         session_tag,
    mi::neuraylib::IDice_transaction* dice_transaction);

  // translate point set
  //
  // \param[in] point_set_tag point set tag to translate the position
  // \param[in] trans_delta_vec translation delta vector
  // \param[in] dice_transaction dice transaction
  void translate_point_set(
    const mi::neuraylib::Tag&                 point_set_tag,
    const mi::math::Vector< mi::Float32, 3 >& trans_delta_vec,
    mi::neuraylib::IDice_transaction*         dice_transaction) const;
  // delete scene element
  // \param[in] scene_element_tag scene element tag to be deleted
  // \param[in] transform_tag     transform tag
  // \param[in] dice_transaction  dice transaction
  void delete_scene_element(
    const mi::neuraylib::Tag&         scene_element_tag,
    const mi::neuraylib::Tag&         transform_tag,
    mi::neuraylib::IDice_transaction* dice_transaction) const;

  // All top view
  //
  // Note: this function assumes that the screen shape is height <=
```

```
  // width resolution.
  //
  // \param[in]  vbbox  viewing bounding box
  // \param[in]  aspect aspect ratio of the screen
  // \param[in]  fovy_2 half of fov of y in radian
  // \param[out] from   camera from
  // \param[out] to     camera to
  // \param[out] up     camera up
  // \param[out] clip_min cliping plane min distance
  // \param[out] clip_max cliping plane max distance
  void get_top_view_param(const mi::math::Bbox<mi::Float32, 3>& vbbox,
              const mi::Float32                    aspect,
              const mi::Float32                    fovy_2,
              mi::math::Vector< mi::Float32, 3 >&   from,
              mi::math::Vector< mi::Float32, 3 >&   to,
              mi::math::Vector< mi::Float32, 3 >&   up,
              mi::Float32                       &   clip_min,
              mi::Float32                       &   clip_max) const;

  // get resolution from opt
  mi::math::Vector<mi::Uint32, 2> get_resolution() const;

  // setup camera to see this example scene
  // \param[in] cam        a camera
  // \param[in] roi_max    region of interest max (min is fixed to (0,0,0)
  // \param[in] dice_transaction dice transaction
  void setup_camera(
    nv::index::IPerspective_camera*        cam,
    const mi::math::Vector<mi::Uint32, 3>& roi_max,
    mi::neuraylib::IDice_transaction*      dice_transaction) const;

  // set up as the main host
  // \return true when success
  bool setup_main_host();

  // animate point
  // \param[in] session_tag session tag
  // \param[in] frame_idx  current frame index used for animate the point set
  // \param[in] dice_transaction dice transaction
  void animate_point(
    const mi::neuraylib::Tag&         session_tag,
    mi::Sint32                        frame_idx,
    mi::neuraylib::IDice_transaction* dice_transaction);

  // render a frame
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // update the scene and render a frame
  // \param[in] frame_idx current frame index
  // \return true when success
  bool update_scene_and_render_frame(mi::Sint32 frame_idx);

  // main host rendering loop
  // \return true when success
  bool mainhost_rendering_loop();
```

```
  // set up dynamic scene: points
  void setup_animation();

  // This session tag
 mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
 mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
 mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
 std::string                                                              m_outfname;
 bool                                                                   m_is_unittest;
 // std::string                                                       m_verify_image_fname;
 mi::Sint32                                                               m_max_iter;
 mi::Uint32                                                             m_resolution_x;
 mi::Uint32                                                             m_resolution_y;
 mi::Sint32                                                                    m_roi_x;
 mi::Sint32                                                                    m_roi_y;
 mi::Sint32                                                                    m_roi_z;
 bool                                                                 m_is_save_all_frame;
 bool                                                                   m_is_animation_on;
  // group node tag which contains point_set
 mi::neuraylib::Tag                                                 m_point_set_group_tag;
  // point_set tag
 mi::neuraylib::Tag                                                       m_point_set_tag;
  // dynamic scene commands. Use as a stack
 std::vector<Frame_command>                                           m_frame_command_vec;
};

mi::Sint32 Dynamic_point_set::launch()
{
  // required on host side too
  nv::index::app::util::time::sleep(0.1f); // wait for 0.1 second

  // setup main host
  check_success(setup_main_host());

  // set up dynamic scene
  setup_animation();

  // call main host rendering loop
  mainhost_rendering_loop();

  return 0;
}

bool Dynamic_point_set::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
```

```
    }
    sdict.insert("max_iter", "20");
    sdict.insert("outfname", ""); // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname          = sdict.get("outfname");
  m_max_iter          = nv::index::app::get_sint32(sdict.get("max_iter"));
  m_resolution_x      = nv::index::app::get_uint32(sdict.get("resolution_x"));
  m_resolution_y      = nv::index::app::get_uint32(sdict.get("resolution_y"));
  m_roi_x             = nv::index::app::get_sint32(sdict.get("roi_x"));
  m_roi_y             = nv::index::app::get_sint32(sdict.get("roi_y"));
  m_roi_z             = nv::index::app::get_sint32(sdict.get("roi_z"));
  m_is_save_all_frame = nv::index::app::get_bool(sdict.get("is_save_all_frame"));
  m_is_animation_on   = nv::index::app::get_bool(sdict.get("is_animation_on"));

  info_cout(std::string("running ") + com_name, sdict);
  info_cout(std::string("outfname = [") + m_outfname +
       "], dice::verbose = " + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name <<" [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-dice::network::multicast_address address]\n"
      << "            set multicast address. (default: "
      << sdict.get("dice::network::multicast_address") + ")\n"

      << "        [-max_iter number_of_iteration]\n"
      << "            set rendering loop iterations. default: " << m_max_iter
      << " frames\n"

      << "        [-resolution_x int] [-resolution_y int]\n"
      << "            screen resolution x and y. (default " << m_resolution_x
      << " " << m_resolution_y<< ")\n"

      << "        [-roi_x int] [-roi_y int] [-roi_z int]\n"
      << "            region of interest max position.\n"
      << "            (The min position is set to (0,0,0).)\n"
     << "          (default: " << m_roi_x << " " << m_roi_y << " " << m_roi_z << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When "", no output.\n"
      << "            The frame number and extension(.ppm) will be added.\n"
      << "            (default: " << m_outfname << ")\n"

      << "        [-is_save_all_frame bool]\n"
      << "            when 1, save only the last frame. (default: "
      << m_is_save_all_frame << ")\n"
```

```
      << "        [-is_animation_on bool]\n"
      << "            when 1, points are animated in every frame. (default: "
      << m_is_animation_on << ")\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode (create smaller volume). "
      << "(default: " << m_is_unittest << ")"
      << std::endl;
    exit(1);
  }
  return true;
}

mi::math::Color_struct Dynamic_point_set::get_color_st_from_attribute(mi::Sint32 attrib) const
{
  const mi::Sint32 intval = abs(attrib);
  mi::math::Color_struct col;
  const mi::Float32 coef = 1.0f / 15.0f;
  col.r = coef * static_cast< mi::Float32 >(intval & 15);
  col.g = coef * static_cast< mi::Float32 >((intval >> 4) & 15);
  col.b = coef * static_cast< mi::Float32 >((intval >> 8) & 15);
  col.a = 1.0;

  return col;
}

mi::Float32 Dynamic_point_set::get_radius_from_attribute(mi::Sint32 attrib) const
{
  const mi::Float32 rad = static_cast< mi::Float32 >(abs(attrib) % 12);
  return rad;
}

void Dynamic_point_set::create_point_set(
  const mi::neuraylib::Tag&         session_tag,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(session_tag.is_valid());
  check_success(dice_transaction != 0);

  mi::base::Handle<nv::index::ISession const> session(
    dice_transaction->access<nv::index::ISession const>(
      session_tag));
  check_success(session.is_valid_interface());

  mi::base::Handle<nv::index::IScene> scene_edit(
    dice_transaction->edit<nv::index::IScene>(
      session->get_scene()));
  check_success(scene_edit.is_valid_interface());

  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // add a light and a material
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
```

```
    scene_edit->create_attribute<nv::index::IDirectional_headlight>());
  check_success(headlight.is_valid_interface());
  const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
  headlight->set_intensity(color_intensity);
  headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
 const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
  check_success(headlight_tag.is_valid());
  group_node->append(headlight_tag, dice_transaction);

  // add material for 3D points shape (the material is only effective for 3D shape)
 mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
  check_success(phong_1.is_valid_interface());
  phong_1->set_ambient(mi::math::Color(0.3f, 0.3f, 0.3f, 1.0f));
  phong_1->set_diffuse(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
  phong_1->set_specular(mi::math::Color(0.4f));
  phong_1->set_shininess(100.f);
 const mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.ge
  check_success(phong_1_tag.is_valid());
  group_node->append(phong_1_tag, dice_transaction);
}

// Point coordinates and its attributes. According to the
// attributes, color and radius are defined.
// Here, we create two point sets.
std::vector< mi::math::Vector_struct< mi::Float32, 3> > point_pos_vec;
std::vector< mi::math::Color_struct >                   color_vec;
std::vector< mi::Float32 >                              radii_vec;

// square shaped positions
const mi::Sint32  column_count= 20;
const mi::Sint32  point_count = 200;
const mi::Float32 x_mag = 25.0;
const mi::Float32 y_mag = 25.0;
const mi::Float32 z     = 30.0;
const mi::Float32 y_offset = 0.0f;
for(mi::Sint32 i = 0; i < point_count; ++i){
  mi::math::Vector_struct< mi::Float32, 3> pos;
  pos.x = static_cast< mi::Float32 >(i % column_count) * x_mag;
  pos.y = static_cast< mi::Float32 >(i / column_count) * y_mag + y_offset;
  pos.z = z;
  point_pos_vec.push_back(pos);
  color_vec.push_back(get_color_st_from_attribute(i));
  radii_vec.push_back(get_radius_from_attribute(i));
}

// Create a point sets
mi::base::Handle< nv::index::IPoint_set > point_set(scene_edit->create_shape<nv::index::IPoint_se
check_success(point_set.is_valid_interface());

nv::index::IPoint_set::Point_style style =nv::index::IPoint_set::SHADED_CIRCLE;
point_set->set_point_style(style);
point_set->set_vertices(&point_pos_vec[0], point_pos_vec.size());
point_set->set_colors(  &color_vec[0],     color_vec.size());
point_set->set_radii(   &radii_vec[0],     radii_vec.size());

// Add the points to the database
m_point_set_tag = dice_transaction->store_for_reference_counting(point_set.get());
```

```
  // if you are not registered Attribute_point_set, the next check fails.
  check_success(m_point_set_tag.is_valid());

  // Add to the scene description
  group_node->append(m_point_set_tag, dice_transaction);
  INFO_LOG << "Added point_set (size: " << point_count << ") to the scene (tag id: "
      << m_point_set_tag.id << ").";

 m_point_set_group_tag = dice_transaction->store_for_reference_counting(group_node.get());
  check_success(m_point_set_group_tag.is_valid());
  scene_edit->append(m_point_set_group_tag, dice_transaction);
}

void Dynamic_point_set::translate_point_set(
  const mi::neuraylib::Tag&                point_set_tag,
  const mi::math::Vector< mi::Float32, 3 >& trans_delta_vec,
  mi::neuraylib::IDice_transaction*        dice_transaction) const
{
  check_success(point_set_tag.is_valid());
  check_success(dice_transaction != 0);

  mi::base::Handle< nv::index::IPoint_set > point_set(
    dice_transaction->edit< nv::index::IPoint_set >(point_set_tag));
  check_success(point_set.is_valid_interface());

  mi::Size const vertex_count = point_set->get_nb_vertices();
  if(vertex_count == 0)
  {
    // no vertices
    return;
  }

 mi::math::Vector_struct<mi::Float32, 3> const * const p_vtx = point_set->get_vertices();
  check_success(p_vtx != 0);

  std::vector< mi::math::Vector_struct<mi::Float32, 3> > vcopy;
  vcopy.reserve(vertex_count);

  for(mi::Uint32 i = 0; i < vertex_count; ++i)
  {
    mi::math::Vector_struct<mi::Float32, 3> vpos = p_vtx[i];
    vpos.x += trans_delta_vec.x;
    vpos.y += trans_delta_vec.y;
    vpos.z += trans_delta_vec.z;
    vcopy.push_back(vpos);
  }
  check_success(vcopy.size() == vertex_count);

  point_set->set_vertices(&(vcopy[0]), vcopy.size());
}

void Dynamic_point_set::delete_scene_element(
  const mi::neuraylib::Tag&         scene_element_tag,
  const mi::neuraylib::Tag&         transform_tag,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(scene_element_tag.is_valid());
```

```
  check_success(transform_tag.is_valid());
  check_success(dice_transaction != 0);

  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    dice_transaction->edit<nv::index::ITransformed_scene_group>(transform_tag));
  check_success(group_node.is_valid_interface());

  group_node->remove(scene_element_tag, dice_transaction);

  std::stringstream sstr;
  sstr << "scene element: " << scene_element_tag.id;
  DEBUG_LOG << sstr.str() << " was removed from the scene.";
}

void Dynamic_point_set::get_top_view_param(const mi::math::Bbox<mi::Float32, 3>& vbbox,
                      const mi::Float32                    aspect,
                      const mi::Float32                    fovy_2,
                      mi::math::Vector< mi::Float32, 3 >&  from,
                      mi::math::Vector< mi::Float32, 3 >&  to,
                      mi::math::Vector< mi::Float32, 3 >&  up,
                      mi::Float32                       &  clip_min,
                      mi::Float32                       &  clip_max) const
{
  check_success((0.0 < fovy_2) && (fovy_2 < M_PI_2));
 const mi::Float32 dist = static_cast<mi::Float32>((0.6 * mi::math::euclidean_distance(vbbox.max, vb
                          (sqrt(2.0) * tan(fovy_2)));
 from = -(dist * mi::math::Vector< mi::Float32, 3 >(0.0f, 0.0f, -1.0f)) + vbbox.center();
  to   = vbbox.center();
  up   = mi::math::Vector< mi::Float32, 3 >(0.0f, 1.0f, 0.0f);
  clip_min = 0.1f  * dist;
  clip_max = 10.0f * dist;
}

mi::math::Vector<mi::Uint32, 2> Dynamic_point_set::get_resolution() const
{
  mi::math::Vector<mi::Uint32, 2> res(m_resolution_x, m_resolution_y);
  if((res.x == 0) || (res.y == 0))
  {
   ERROR_LOG << "illegal_resolution setting: [" << res.x << "x" << res.y << "], use 1024x1024.";
    res.x = 1024;
    res.y = 1024;
  }
  return res;
}

void Dynamic_point_set::setup_camera(
  nv::index::IPerspective_camera*        cam,
  const mi::math::Vector<mi::Uint32, 3>& roi_max,
  mi::neuraylib::IDice_transaction*      dice_transaction) const
{
  check_success(cam != 0);

  cam->set_aperture(0.033f);
  cam->set_focal(0.03f);

  const mi::math::Vector<mi::Uint32, 2> res = get_resolution();
  const mi::Float32 pix_aspect_ratio_1x1 = 1.0f;
```

```
  // static_cast< mi::Float32 >(res.x) / static_cast< mi::Float32 >(res.y);

  cam->set_aspect(pix_aspect_ratio_1x1);

  // get top view for this scene
  mi::math::Bbox< mi::Float32, 3 > const
    vbbox(0.0f, 0.0f, 0.0f,
       static_cast< mi::Float32>(roi_max.x), static_cast< mi::Float32>(roi_max.y),
       static_cast< mi::Float32>(roi_max.z));
  mi::math::Vector< mi::Float32, 3 > from(0.0f, 0.0f, 0.0f);
  mi::math::Vector< mi::Float32, 3 > to  (0.0f, 0.0f, 0.0f);
  mi::math::Vector< mi::Float32, 3 > up  (0.0f, 0.0f, 0.0f);
  mi::Float32 clip_min =  0.1f;
  mi::Float32 clip_max = 10.0f;
  get_top_view_param(vbbox,
            static_cast<mi::Float32>(cam->get_aspect()),
            static_cast<mi::Float32>((cam->get_fov_y_rad() / 2.0)),
            from, to, up, clip_min, clip_max);

  INFO_LOG << "set camera resolution [" << res.x << " " << res.y
      << "], aspect_ratio: " << pix_aspect_ratio_1x1;

  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_clip_min(clip_min);
  cam->set_clip_max(clip_max);
}

bool Dynamic_point_set::setup_main_host()
{
  // roi max
  mi::math::Vector< mi::Uint32, 3 > roi_max(m_roi_x, m_roi_y, m_roi_z);
  for(mi::Sint32 i = 0; i < 3; ++i)
  {
    if(roi_max[i] <= 0)
    {
     ERROR_LOG << "illegal roi setting: [" << i <<"] = " << roi_max[i] << ", use 1024.";
      roi_max[i] = 1024;
    }
  }

  // Access the IndeX rendering query interface
  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  // DiCE database access
```

```
mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
  m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
check_success(dice_transaction.is_valid_interface());
{
  // Setup session information
  m_session_tag =
    m_index_session->create_session(dice_transaction.get());
  check_success(m_session_tag.is_valid());
  //------------------------------------------------------------------------
  // Scene setup

  // no scene setup in the start up in this example

  // Create and edit a camera. Data distribution is based on
  // the camera. (Because only visible massive data are
  // considered)
  mi::base::Handle< nv::index::ISession const > session(
    dice_transaction->access< nv::index::ISession const >(
      m_session_tag));
  check_success(session.is_valid_interface());

  mi::base::Handle< nv::index::IScene > scene_edit(
    dice_transaction->edit<nv::index::IScene>(session->get_scene()));
  check_success(scene_edit.is_valid_interface());

  mi::base::Handle< nv::index::IPerspective_camera > cam(
    scene_edit->create_camera<nv::index::IPerspective_camera>());
  check_success(cam.is_valid_interface());
  setup_camera(cam.get(), roi_max, dice_transaction.get());
  const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
  check_success(camera_tag.is_valid());

  const mi::math::Vector<mi::Uint32, 2> buffer_resolution = get_resolution();
  m_image_file_canvas->set_resolution(buffer_resolution);

  // Set up the scene
  mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
    { 0.0f, 0.0f, 0.0f, },
    { static_cast< mi::Float32 >(roi_max.x),
     static_cast< mi::Float32 >(roi_max.y),
     static_cast< mi::Float32 >(roi_max.z), },
  };

  // set the region of interest
  const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
  check_success(xyz_roi.is_volume());
  scene_edit->set_clipped_bounding_box(xyz_roi_st);

  // Set the scene global transformation matrix.
  // only change the coordinate system
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f,  0.0f,  0.0f,  0.0f,
    0.0f,  1.0f,  0.0f,  0.0f,
    0.0f,  0.0f, -1.0f,  0.0f,
    0.0f,  0.0f,  0.0f,  1.0f
    );
  scene_edit->set_transform_matrix(transform_mat);
```

```
    // Set the current camera to the scene.
    check_success(camera_tag.is_valid());
    scene_edit->set_camera(camera_tag);

   INFO_LOG << "ROI set to [0, 0, 0]-[" << roi_max.x << ", " << roi_max.y << ", " << roi_max.z << "]";
  }
  dice_transaction->commit();

  INFO_LOG << "Initialization complete.";

  return true;
}

void Dynamic_point_set::animate_point(
  const mi::neuraylib::Tag&         session_tag,
  mi::Sint32                        frame_idx,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  if(m_frame_command_vec.empty())
  {
    // no more processing commands
    return;
  }

  if(m_frame_command_vec.back().get_frame() != frame_idx)
  {
    return;
  }

  Frame_command fc = m_frame_command_vec.back(); // copy
  check_success(!m_frame_command_vec.empty());
  m_frame_command_vec.pop_back();

  INFO_LOG << "execute command: " << fc.to_string();
  const std::string & com = fc.get_command();
  if(com == "create")
  {
    if(m_point_set_tag.is_valid())
    {
      ERROR_LOG << "create: Point set has been already created. ignored.";
      return;
    }
    create_point_set(session_tag, dice_transaction);
  }
  else if(com == "delete")
  {
    if(!(m_point_set_tag.is_valid()))
    {
      ERROR_LOG << ("delete: Point set doesn't exist. ignored.");
      return;
    }
    delete_scene_element(m_point_set_tag,
              m_point_set_group_tag,
              dice_transaction);
    m_point_set_tag = mi::neuraylib::NULL_TAG;
  }
```

```
  else if(com == "translate_delta")
  {
    if(!(m_point_set_tag.is_valid()))
    {
      ERROR_LOG << ("translate_delta: Point set doesn't exist. ignored.");
      return;
    }
    std::vector< std::string > args = fc.get_argument();
    check_success(args.size() == 4);
   mi::math::Vector< mi::Float32, 3 > trans_delta_vec(nv::index::app::get_float32(args[1]),
                              nv::index::app::get_float32(args[2]),
                              nv::index::app::get_float32(args[3]));
    translate_point_set(m_point_set_tag,
              trans_delta_vec,
              dice_transaction);
  }
  else
  {
    ERROR_LOG << ("unknown command [" + com + "]. ignored.");
  }
}

nv::index::IFrame_results* Dynamic_point_set::render_frame(const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

bool Dynamic_point_set::update_scene_and_render_frame(mi::Sint32 frame_idx)
{
  bool success = true;

 mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
  check_success(dice_transaction.is_valid_interface());
  {
```

```
    // Change the scene element, camera, etc. here. This example
    // does not change any, but usually you can change camera
    // parameters, add/remove scene elements here.
    animate_point(m_session_tag, frame_idx, dice_transaction.get());
  }
  dice_transaction->commit();


  // Render a frame and save the rendered image to a file.
  // Only save the file at the end of the iteration.
  std::string fname = "";
  if (m_is_save_all_frame)
  {
    fname = get_output_file_name(m_outfname, frame_idx);
  }

  mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
 const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
  if (err_set->any_errors())
  {
    std::ostringstream os;

   const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    const mi::Uint32 nb_err = err_set->get_nb_errors();
    for (mi::Uint32 e = 0; e < nb_err; ++e)
    {
      if (e != 0) os << '\n';
      const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
      os << err->get_error_string();
    }

   ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
         << os.str();
    success = false;
  }

  return success;
}

bool Dynamic_point_set::mainhost_rendering_loop()
{
  check_success(m_max_iter > 0);
  const mi::Sint32 max_frame_count = m_max_iter - 1;
  INFO_LOG << "set max iteration: " << m_max_iter;
  INFO_LOG << "set is_save_all_frame: " << (m_is_save_all_frame ? "true" : "false");

  // rendering loop
  bool is_ok = true;
  for(mi::Sint32 i = 0; i < m_max_iter; ++i)
  {
    if((!m_is_save_all_frame) && (i != max_frame_count))
    {
      m_is_save_all_frame = true;
    }

    is_ok = update_scene_and_render_frame(i);
    if(!is_ok)
```

```
    {
      break;
    }
  }

  INFO_LOG << "Finished main host rendering loop.";

  return is_ok;
}

void Dynamic_point_set::setup_animation()
{
  // commands
  std::vector< std::string > com_create;
  com_create.push_back("create");
  std::vector< std::string > com_translate;
  com_translate.push_back("translate_delta");
  com_translate.push_back("20.0"); // dx
  com_translate.push_back("20.0"); // dy
  com_translate.push_back("20.0"); // dz

  std::vector< std::string > com_delete;
  com_delete.push_back("delete");

  m_frame_command_vec.push_back(Frame_command(5, com_create));

  for(mi::Sint32 i = 10; i < 20; ++i)
  {
    m_frame_command_vec.push_back(Frame_command(i, com_translate));
  }
  m_frame_command_vec.push_back(Frame_command(22, com_delete));

  if(!m_is_unittest)
  {
    // non unit test mode
    // create and delete again
    mi::Sint32 const base_frame = 30;
    mi::Sint32 const interval   = 5; // must be larger than offset
    mi::Sint32 const iter       = 3;
    for(mi::Sint32 i = 0; i < iter; ++i)
    {
      mi::Sint32 iter_base = base_frame + (i * interval);
      mi::Sint32 offset = 0;
      m_frame_command_vec.push_back(Frame_command(iter_base + offset, com_create));
      ++offset;
     m_frame_command_vec.push_back(Frame_command(iter_base + offset, com_translate));
      ++offset;
      m_frame_command_vec.push_back(Frame_command(iter_base + offset, com_delete));
      ++offset;
      check_success(offset < interval);
    }
  }

 std::sort(m_frame_command_vec.begin(), m_frame_command_vec.end(), Frame_command_sorter());

  // descendant order: debug print (need this int)
  assert(!(m_frame_command_vec.empty()));
```

```
  for(mi::Sint32 i = static_cast<mi::Sint32>(m_frame_command_vec.size() - 1); i >=0 ; --i)
   {
     // std::cout << "idx: " << i << std::endl;
     DEBUG_LOG << m_frame_command_vec.at(i).to_string();
   }
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                              // log level
 sdict.insert("dice::network::multicast_address", "224.1.3.2"); // default multicast address
 sdict.insert("max_iter", "40");                       // default max rendering loop iterations
  sdict.insert("resolution_x", "1024");                            // X resolution
  sdict.insert("resolution_y", "1024");                            // Y resolution
 sdict.insert("roi_x", "512");                              // region of interest max X
 sdict.insert("roi_y", "512");                              // region of interest max Y
 sdict.insert("roi_z", "512");                              // region of interest max Z
 sdict.insert("outfname", "frame_dynamic_point");              // output file base name
  sdict.insert("is_save_all_frame", "0");                          // save all frame
  sdict.insert("is_animation_on", "1");                            // points animation
 sdict.insert("point_rendering_mode", "raytracing");          // point rendering mode
  sdict.insert("point_radius_mode", "object_space");             // point radius mode
  sdict.insert("unittest", "0");                                   // unit test mode
 sdict.insert("is_dump_comparison_image_when_failed", "1");    // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");                 // default: not call from make check.

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");

  // Initialize application
  Dynamic_point_set dynamic_point_set;
  dynamic_point_set.initialize(argc, argv, sdict);
  check_success(dynamic_point_set.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = dynamic_point_set.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.21 embedded_heightfield_geometry.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/idata_sample.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iregular_heightfield.h>
#include <nv/index/iheightfield_pick_result.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/itexture_filter_mode.h>
#include <nv/index/isparse_volume_rendering_properties.h>

#include "utility/app_rendering_context.h"
#include "utility/canvas_utility.h"
#include "utility/example_shared.h"

#include <nv/index/app/idata_analysis_and_processing.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include <iostream>
#include <sstream>

class Embedded_heightfield_geometry:
  public nv::index::app::Index_connect
{
public:
  Embedded_heightfield_geometry()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Embedded_heightfield_geometry() ctor";
  }

  virtual ~Embedded_heightfield_geometry()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Embedded_heightfield_geometry() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
```

```
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  void setup_camera(nv::index::IPerspective_camera* cam) const;
  void pick_call(
    const mi::math::Vector_struct<mi::Uint32, 2>& pick_location,
    mi::neuraylib::IDice_transaction*             dice_transaction) const;
  void setup_scene(mi::neuraylib::IDice_transaction* dice_transaction);
  // render a frame
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  mi::base::Handle<nv::index::IIndex_scene_query>                       m_index_query;
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                        m_verify_image_fname;
  std::string                                                           m_heightfield;
  std::string                                                                  m_mask;
  std::string                                                              m_geometry;
  std::string                                                         m_voxel_format;
  std::string                                                               m_render;
  mi::Float32                                                                 m_size;
  std::string                                                              m_texture;
  bool                                                                 m_use_texture;
  bool                                                                        m_zoom;
  bool                                                                        m_pick;
  mi::Uint32                                                         m_supersampling;
};

mi::Sint32 Embedded_heightfield_geometry::launch()
{
  mi::Sint32 exit_code = 0;
  {
   m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configura
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
```

```
m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
 check_success(m_image_file_canvas.is_valid_interface());

m_index_query = get_index_interface()->get_api_component<nv::index::IIndex_scene_query>();
 check_success(m_index_query.is_valid_interface());

// Verify that the local host has joined, this may fail when there is a license problem
 check_success(is_local_host_joined(m_cluster_configuration.get()));

 {
   // Create our DiCE transaction
   mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
     m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
   check_success(dice_transaction.is_valid_interface());
   {
     // Create an IndeX session
     m_session_tag = m_index_session->create_session(dice_transaction.get());
     check_success(m_session_tag.is_valid());

     if(m_supersampling > 0)
     {
       mi::base::Handle<const nv::index::ISession> session(
         dice_transaction->access<const nv::index::ISession>(m_session_tag));
       check_success(session.is_valid_interface());

       mi::base::Handle<nv::index::IConfig_settings> config_settings(
     dice_transaction->edit<nv::index::IConfig_settings>(session->get_config()));
       check_success(config_settings.is_valid_interface());

       // Enable supersampling
       config_settings->set_rendering_samples(m_supersampling);
     }


     // Create the scene
     setup_scene(dice_transaction.get());
   }
   dice_transaction->commit();
 }

 // Render the scene
 {
   const mi::Sint32  frame_idx = 0;
   const std::string fname     = get_output_file_name(m_outfname, frame_idx);
   m_image_file_canvas->set_rgba_file_name(fname.c_str());

  mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
 const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
   if (err_set->any_errors())
   {
     std::ostringstream os;

  const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    const mi::Uint32                           nb_err = err_set->get_nb_errors();
     for (mi::Uint32 e = 0; e < nb_err; ++e)
     {
       if (e != 0) os << '\n';
```

```
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

      ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        exit_code = 1;
      }

      // Verify the generated image
      if (!verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), m_verify_image_fname, get_options()))
      {
        exit_code = 1;
      }
    }

    // Picking
    {
      // Create our DiCE transaction
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        if (m_pick)
        {
       pick_call(mi::math::Vector<mi::Uint32, 2>(386, 322), dice_transaction.get());
       pick_call(mi::math::Vector<mi::Uint32, 2>(98, 318),  dice_transaction.get());
       pick_call(mi::math::Vector<mi::Uint32, 2>(422, 555), dice_transaction.get());
      pick_call(mi::math::Vector<mi::Uint32, 2>(1024 - 962, 1024 - 453), dice_transaction.get());

          // Picks a line in zoom mode
       pick_call(mi::math::Vector<mi::Uint32, 2>(524, 496), dice_transaction.get());

          // Pick a point in zoom mode
       pick_call(mi::math::Vector<mi::Uint32, 2>(405, 578), dice_transaction.get());
        }
      }
      // Finish the picking transaction
      dice_transaction->commit();
    }
  }
  return exit_code;
}

bool Embedded_heightfield_geometry::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", ""); // turn off file output in the unit test mode
```

```
    sdict.insert("dice::verbose", "2");
  }

  m_outfname           = sdict.get("outfname");
  m_verify_image_fname = sdict.get("verify_image_fname");
  m_heightfield        = sdict.get("heightfield");
  m_mask               = sdict.get("mask");
  m_geometry           = sdict.get("geometry");
  m_voxel_format       = sdict.get("voxel_format");
  m_render             = sdict.get("render");
  m_size               = nv::index::app::get_float32(sdict.get("size"));
  m_texture            = sdict.get("texture");
  m_use_texture        = (m_texture != "none");
  m_zoom               = nv::index::app::get_bool(sdict.get("zoom", "false"));
  m_pick               = nv::index::app::get_bool(sdict.get("pick", "false"));
  m_supersampling      = nv::index::app::get_uint32(sdict.get("supersampling"));

  info_cout("running " + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_fname = [" + m_verify_image_fname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            print out this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info). (default: " + sdict.get("dice::verbose") << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_fname image_fname]\n"
      << "            when image_fname exist, verify the rendering image.\n"
      << "            (default: [" << m_verify_image_fname << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode.\n"
      << "            (default: [" << m_is_unittest << "])\n"

      << "        [-heightfield ppm_file]\n"
     << "          filename of a grayscale PPM image that represents the heightfield.\n"
      << "            (default: [" << m_heightfield << "])\n"

      << "        [-mask pgm_file]\n"
    << "          filename of a monochrome PGM image that will be used as a mask on the heightfield,\n"
     << "           creating hole and therefore isolated points and connecting lines.\n"
      << "            (default: [" << m_mask << "])\n"

      << "        [-geometry mode]\n"
      << "            NOTE: THIS MODE == volume IS DISABLED.\n"
      << "            color modes for embedded heightfield geometry.\n"
```

```
      << "            Available modes: none, fixed, material, volume, texture\n"
      << "            (default: [" << m_geometry << "])\n"

      << "        [-voxel_format format]\n"
      << "            voxel format for the volume geometry mode.\n"
      << "            Available formats: uint8, rgba8\n"
      << "            (default: [" << m_voxel_format << "])\n"

      << "        [-render mode]\n"
      << "            rendering modes for embedded heightfield geometry.\n"
      << "            Available modes: z-axis, screen, raster\n"
      << "            (default: [" << m_render << "])\n"

      << "        [-size float]\n"
      << "            geometry size for z-axis or screen, ignored for raster.\n"
      << "            (default: [" << m_size << "])\n"

      << "        [-texture mode]\n"
      << "            map a computed texture onto the heightfield.\n"
      << "            Available modes: none, mandelbrot\n"
      << "            (default: " << m_texture << ")\n"

      << "        [-zoom bool]\n"
      << "            zoom in to show details of the geometry.\n"
      << "            (default: [" << m_zoom << "])\n"

      << "        [-pick bool]\n"
      << "            apply picking operation at various screen positions.\n"
      << "            (default: [" << m_pick << "])\n"

      << "        [-supersampling int]\n"
    << "        apply supersampling using the given number of samples. If set to '0' supersampling is
      << "            (default: [" << m_supersampling << "])\n"

      << std::endl;

    std::exit(1);
  }
  return true;
}

void Embedded_heightfield_geometry::setup_camera(nv::index::IPerspective_camera* cam) const
{
  // Set the camera parameters to see the whole scene
  mi::math::Vector<mi::Float32, 3> from;
  mi::math::Vector<mi::Float32, 3> to;
  mi::math::Vector<mi::Float32, 3> up(0.f, 0.f, 1.0);

  if (m_zoom)
  {
    from = mi::math::Vector<mi::Float32, 3>(200.f, 60.f, 100.f);
    to   = mi::math::Vector<mi::Float32, 3>(170.f, 20.f, 70.f);
  }
  else
  {
    from = mi::math::Vector<mi::Float32, 3>(-0.f, -50.f, 200.f);
    to   = mi::math::Vector<mi::Float32, 3>(150.f, 150.f, 0.f);
```

```
  }

  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(1000.0f);
}

void Embedded_heightfield_geometry::pick_call(
  const mi::math::Vector_struct<mi::Uint32, 2>& pick_location,
  mi::neuraylib::IDice_transaction*             dice_transaction) const
{
  INFO_LOG << "Picking at screen position: " << pick_location;

  // Run the picking operation
  mi::base::Handle<nv::index::IScene_pick_results> scene_pick_results(
   m_index_query->pick(pick_location, m_image_file_canvas.get(), m_session_tag, dice_transaction));

  // Print the picking result
  const mi::Uint32 nb_results = scene_pick_results->get_nb_results();
  if (nb_results > 0)
  {
    INFO_LOG << "Number of pick results: " << nb_results;
    for (mi::Uint32 i=0; i < nb_results; i++)
    {
      // Generic information
    const mi::base::Handle<nv::index::IScene_pick_result> result(scene_pick_results->get_result(i)

      std::stringstream log_out;
      log_out << "Intersection no. " << i << "\n"
         << "\t\t Element (tag)          " << result->get_scene_element().id << "\n"
       << "\t\t Sub index:          " << result->get_scene_element_sub_index() << "\n"
          << "\t\t Distance:               " << result->get_distance() << "\n"
          << "\t\t Position (local space): " << result->get_intersection() << "\n"
          << "\t\t Color (evaluated):      " << result->get_color() << "\n";

    const mi::base::Handle<const nv::index::IData_sample> data_sample(result->get_data_sample());
      if (data_sample)
      {
     const mi::base::Handle<const nv::index::IData_sample_uint8>  ds_uint8(data_sample->get_interf
     const mi::base::Handle<const nv::index::IData_sample_uint16>  ds_uint16(data_sample->get_inte
     const mi::base::Handle<const nv::index::IData_sample_float32> ds_float32(data_sample->get_int
     const mi::base::Handle<const nv::index::IData_sample_rgba8>  ds_rgba8(data_sample->get_interf
        if (ds_uint8) {
       log_out << "\t Data sample:    " << mi::Uint32(ds_uint8->get_sample_value()) << "\n";
        }
        else if (ds_uint16) {
         log_out << "\t Data sample:     " << ds_uint16->get_sample_value() << "\n";
        }
        else if (ds_float32) {
         log_out << "\t Data sample:     " << ds_float32->get_sample_value() << "\n";
        }
```

```
        else if (ds_rgba8) {
          log_out << "\t Data sample:     " << ds_rgba8->get_sample_value() << "\n";
        }
      }

      INFO_LOG << log_out.str();

      // Shape-specific information
    if (result->get_intersection_info_class() == nv::index::IHeightfield_pick_result::IID())
      {
     mi::base::Handle<const nv::index::IHeightfield_pick_result> compute_pick_result(
          result->get_interface<const nv::index::IHeightfield_pick_result>());
        if (compute_pick_result && compute_pick_result->is_computing_enabled())
        {
          INFO_LOG << "Specific pick results for computed heightfield texture:";
       INFO_LOG << "\t Computed color value:   " << compute_pick_result->get_computed_color();
        }
      }
    }
  }
  else
  {
    INFO_LOG << "No result at pick position: " << pick_location;
  }
}

void Embedded_heightfield_geometry::setup_scene(
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  // Access the session
  mi::base::Handle<const nv::index::ISession> session(
    dice_transaction->access<nv::index::ISession>(m_session_tag));
  check_success(session.is_valid_interface());

  // Edit the scene
  mi::base::Handle<nv::index::IScene> scene_edit(
    dice_transaction->edit<nv::index::IScene>(session->get_scene()));
  check_success(scene_edit.is_valid_interface());

  // Create static group node where all the scene elements will be attached
  mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
    scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
  check_success(static_group_node.is_valid_interface());

  mi::neuraylib::Tag colormap_tag;
  if (m_geometry == "volume")
  {
    // Create a color map using an external utility function.
   const mi::Sint32 colormap_entry_id = 40; // same as demo application's colormap file 40
   colormap_tag               = create_colormap(colormap_entry_id, scene_edit.get(), dice_transact
    check_success(colormap_tag.is_valid());
  }

  // Create the heightfield
  mi::neuraylib::Tag                      heightfield_tag;
  const mi::math::Vector<mi::Uint32, 2>  heightfield_size(300, 300);
  const mi::math::Vector<mi::Float32, 2> elevation_range(0.f, 300.f);
```

```
{
  // Creating a ppm heightfield  importer.
  nv::index::app::String_dict heightfield_opt;
  heightfield_opt.insert("args::type",       "heightfield");
 heightfield_opt.insert("args::importer",  "nv::index::plugin::legacy_importer.PPM_heightfield_i
  heightfield_opt.insert("args::input_file", m_heightfield);
  {
    std::stringstream sstr;
    sstr << heightfield_size.x << " " << heightfield_size.y;
    heightfield_opt.insert("args::size",  sstr.str());
  }
  heightfield_opt.insert("args::importer_scale",  "0.3");
  heightfield_opt.insert("args::importer_offset", "0");
  heightfield_opt.insert("args::importer_binary_mask", m_mask);
  {
    std::stringstream sstr;
    sstr << elevation_range.x << " " << elevation_range.y;
    heightfield_opt.insert("args::range", sstr.str());
  }
  nv::index::IDistributed_data_import_callback* importer_callback =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::legacy_importer.PPM_heightfield_importer",
      heightfield_opt);

  // Heightfield scene element
  const mi::Float32                      rotate_k = 0.f;
  const mi::math::Vector<mi::Float32, 3> translate(0.f, 0.f, 1.f);
  const mi::math::Vector<mi::Float32, 3> scale(1.f, 1.f, 1.f);
  mi::base::Handle<nv::index::IRegular_heightfield> heightfield_scene_element(
    scene_edit->create_regular_heightfield(
      scale, rotate_k, translate,
      heightfield_size,
      elevation_range,
      importer_callback,
      dice_transaction));
  check_success(heightfield_scene_element.is_valid_interface());

  if (m_geometry == "volume")
  {
    // Enable volume mapping for the heightfield
    heightfield_scene_element->set_colormap_mapping(true);
    heightfield_scene_element->assign_colormap(colormap_tag);
  }

 heightfield_tag = dice_transaction->store_for_reference_counting(heightfield_scene_element.get(
  check_success(heightfield_tag.is_valid());
}

// Add a light, a material, a rendering property, and a colormap
{
  // Light
  mi::base::Handle<nv::index::IDirectional_headlight> headlight(
    scene_edit->create_attribute<nv::index::IDirectional_headlight>());
  check_success(headlight.is_valid_interface());
  const mi::math::Color    color_intensity(1.f, 1.f, 1.f, 1.f);
  headlight->set_intensity(color_intensity);
```

```
  headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.f, -1.f, -1.f));
 const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
  check_success(headlight_tag.is_valid());
  static_group_node->append(headlight_tag, dice_transaction);

  // Material for the heightfield
 mi::base::Handle<nv::index::IPhong_gl> material(scene_edit->create_attribute<nv::index::IPhong_
  check_success(material.is_valid_interface());

  if (m_use_texture)
  {
    // Use just ambient white with texture
    material->set_ambient(mi::math::Color(1.f));
    material->set_diffuse(mi::math::Color(0.f));
    material->set_specular(mi::math::Color(0.f));
  }
  else
  {
    // Define a more interesting greenish material
    material->set_ambient(mi::math::Color(0.f, 0.1f, 0.0f));
    material->set_diffuse(mi::math::Color(0.f, 0.8f, 0.2f));
    material->set_specular(mi::math::Color(0.3f));
    material->set_shininess(100.f);
  }

  const mi::neuraylib::Tag material_tag
    = dice_transaction->store_for_reference_counting(material.get());
  check_success(material_tag.is_valid());
  static_group_node->append(material_tag, dice_transaction);

  // Add a sparse_volume_render_properties to the scene.
 mi::base::Handle<nv::index::ISparse_volume_rendering_properties> sparse_render_prop(
  scene_edit->create_attribute<nv::index::ISparse_volume_rendering_properties>());
  sparse_render_prop->set_filter_mode(nv::index::SPARSE_VOLUME_FILTER_NEAREST);
  sparse_render_prop->set_sampling_distance(             1.0);
  sparse_render_prop->set_reference_sampling_distance(   1.0);
 sparse_render_prop->set_voxel_offsets(           mi::math::Vector<mi::Float32, 3>(0.0f, 0.0f, 0
  sparse_render_prop->set_preintegrated_volume_rendering(false);
  sparse_render_prop->set_lod_rendering_enabled(         false);
  sparse_render_prop->set_lod_pixel_threshold(           2.0);
  sparse_render_prop->set_debug_visualization_option(    0);
  const mi::neuraylib::Tag sparse_render_prop_tag
    = dice_transaction->store_for_reference_counting(sparse_render_prop.get());
  check_success(sparse_render_prop_tag.is_valid());
  static_group_node->append(sparse_render_prop_tag, dice_transaction);

  // Add a colormap to the scene.
 const mi::Sint32 colormap_entry_id   = 1; // same as demo application's colormap file 1
  const mi::neuraylib::Tag colormap_tag =
    create_colormap(colormap_entry_id, scene_edit.get(), dice_transaction);
  check_success(colormap_tag.is_valid());
  static_group_node->append(colormap_tag, dice_transaction);
}

// Optionally map a computed texture onto the heightfield
if (m_use_texture)
{
```

```
    // Mandelbrot texture
    if (m_texture == "mandelbrot")
    {
      mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
     scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
      check_success(tex_filter.is_valid_interface());
    mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.g
      check_success(tex_filter_tag.is_valid());
      static_group_node->append(tex_filter_tag, dice_transaction);

    // Access an application layer component that provides some sample techniques such as the mandelbr
    mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing>
    get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_proces
    // Create a mandelbrot technique and add it to the scene. For more details on how to implement the m
    // source that was shipped with the application layer component 'nv::index::app::data_analysis_ar
      // Create the computed texture, with a size fitting the heightfield:
    mi::base::Handle<nv::index::app::data_analysis_and_processing::ISample_tool_set> tool_set(proc
      mi::base::Handle<nv::index::IDistributed_compute_technique> mapping(
        tool_set->create_mandelbrot_2d_technique(
       mi::math::Vector<mi::Float32, 2>(heightfield_size), nv::index::IDistributed_compute_destinat

      // Add the mapping to the scene before the heightfield
    mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.get())
      check_success(mapping_tag.is_valid());
      static_group_node->append(mapping_tag, dice_transaction);
    }
    else
    {
      ERROR_LOG << "Unknown texture mode '" << m_texture << "'";
      check_success(false);
    }
  }

  // Configure rendering of the embedded geometry
  if (m_geometry == "fixed")
  {
    // Rendering mode
    nv::index::IHeightfield_geometry_settings::Render_mode render_mode =
      nv::index::IHeightfield_geometry_settings::RENDER_DEFAULT;
    if (m_render == "z-axis")
    {
     render_mode = nv::index::IHeightfield_geometry_settings::RENDER_Z_AXIS_ALIGNED;
    }
    else if (m_render == "screen")
    {
     render_mode = nv::index::IHeightfield_geometry_settings::RENDER_SCREEN_ALIGNED;
    }
    else if (m_render == "raster")
    {
      render_mode = nv::index::IHeightfield_geometry_settings::RENDER_RASTERIZED;

      // Use fixed geometry size in rasterized mode (only use for picking)
      m_size = 2.0f;
    }
    else
    {
      ERROR_LOG << "Unknown render mode '" << m_render << "'";
```

```
      check_success(false);
    }

    //
    // Use separate settings for points and lines
    //

    // Settings for isolated points
    mi::base::Handle<nv::index::IHeightfield_geometry_settings> point_settings(
      scene_edit->create_attribute<nv::index::IHeightfield_geometry_settings>());
    check_success(point_settings.is_valid_interface());
    // Apply this attribute only to isolated points
  point_settings->set_type_mask(nv::index::IHeightfield_geometry_settings::TYPE_ISOLATED_POINTS);
    // Use a fixed color with no lighting
  point_settings->set_color_mode(nv::index::IHeightfield_geometry_settings::MODE_FIXED);
    // Blue points please
    point_settings->set_color(mi::math::Color(0.f, 0.f, 1.f));
    // Enable rendering of seed points
    point_settings->set_visible(true);
    // Set requested render mode
    point_settings->set_render_mode(render_mode);
    // Line width / point radius
    point_settings->set_geometry_size(m_size);

    const mi::neuraylib::Tag point_settings_tag
      = dice_transaction->store_for_reference_counting(point_settings.get());
    check_success(point_settings_tag.is_valid());
    static_group_node->append(point_settings_tag, dice_transaction);

    // Settings for seed lines
    mi::base::Handle<nv::index::IHeightfield_geometry_settings> line_settings(
      scene_edit->create_attribute<nv::index::IHeightfield_geometry_settings>());
    check_success(line_settings.is_valid_interface());
    // Apply this attribute only to connecting lines
  line_settings->set_type_mask(nv::index::IHeightfield_geometry_settings::TYPE_CONNECTING_LINES);
    // Use a fixed color with no lighting
  line_settings->set_color_mode(nv::index::IHeightfield_geometry_settings::MODE_FIXED);
    // Red lines please
    line_settings->set_color(mi::math::Color(1.f, 0.f, 0.f));
    // Enable rendering of seed lines
    line_settings->set_visible(true);
    // Set requested render mode
    line_settings->set_render_mode(render_mode);
    // Line width / point radius
    line_settings->set_geometry_size(m_size);

    const mi::neuraylib::Tag line_settings_tag
      = dice_transaction->store_for_reference_counting(line_settings.get());
    check_success(line_settings_tag.is_valid());
    static_group_node->append(line_settings_tag, dice_transaction);
  }
  else if (m_geometry != "none")
  {
    //
    // Use the same settings for points and lines
    //
    mi::base::Handle<nv::index::IHeightfield_geometry_settings> geometry_settings(
```

```
    scene_edit->create_attribute<nv::index::IHeightfield_geometry_settings>());
  check_success(geometry_settings.is_valid_interface());

  // Apply this attribute to both isolated points and connecting lines
  geometry_settings->set_type_mask(
    nv::index::IHeightfield_geometry_settings::TYPE_ISOLATED_POINTS |
    nv::index::IHeightfield_geometry_settings::TYPE_CONNECTING_LINES);

  if (m_geometry == "material")
  {
    // Apply the heightfield material on the embedded geometry
  geometry_settings->set_color_mode(nv::index::IHeightfield_geometry_settings::MODE_MATERIAL);
  }
  else if (m_geometry == "volume")
  {
    // Map the volume texture on the embedded geometry
  geometry_settings->set_color_mode(nv::index::IHeightfield_geometry_settings::MODE_VOLUME_TEXTU
  }
  else if (m_geometry == "texture")
  {
    // Map the computed heightfield texture on the embedded geometry
  geometry_settings->set_color_mode(nv::index::IHeightfield_geometry_settings::MODE_COMPUTED_TEX
  }
  else
  {
    ERROR_LOG << "Unknown geometry mode '" << m_geometry << "'";
    check_success(false);
  }

 // Set color to white (the default), it will modulate the material or texture color
  geometry_settings->set_color(mi::math::Color(1.f, 1.f, 1.f));
  // Enable rendering (default)
  geometry_settings->set_visible(true);

  const mi::neuraylib::Tag geometry_settings_tag
    = dice_transaction->store_for_reference_counting(geometry_settings.get());
  check_success(geometry_settings_tag.is_valid());
  static_group_node->append(geometry_settings_tag, dice_transaction);
}

// Create an artifical volume if requested
if (m_geometry == "volume")
{
  // Set up parameter for the synthetic volume generation technique
  const mi::math::Vector<mi::Uint32, 3> volume_size(
  heightfield_size.x, heightfield_size.y, static_cast<mi::Uint32>(elevation_range.y));
  const mi::math::Bbox<mi::Uint32, 3> bbox(
    mi::math::Vector<mi::Uint32, 3>(0), volume_size);

  // Creating a sparse volume synthetic volume generator.
  nv::index::app::String_dict        sparse_volume_opt;
  sparse_volume_opt.insert("args::type",     "sparse_volume");
  sparse_volume_opt.insert("args::importer", "synthetic");
  std::stringstream                         sstr;
 sstr << "0 0 0 " << volume_size.x << " " << volume_size.y << " " << volume_size.z;
  sparse_volume_opt.insert("args::bbox",           sstr.str());
  sparse_volume_opt.insert("args::voxel_format",   "uint8");
```

```
    sparse_volume_opt.insert("args::synthetic_type", "sphere_0");
    nv::index::IDistributed_data_import_callback* importer_callback =
      get_importer_from_application_layer(
        get_application_layer_interface(),
        "nv::index::plugin::base_importer.Sparse_volume_generator_synthetic",
        sparse_volume_opt);
    const mi::math::Bbox<mi::Float32, 3> ijk_bbox(0.0f, 0.0f, 0.0f,
                          static_cast<mi::Float32>(volume_size.x),
                          static_cast<mi::Float32>(volume_size.y),
                          static_cast<mi::Float32>(volume_size.z));
   const mi::math::Matrix<mi::Float32, 4, 4> transform_mat(1.0f); // Identity matrix
    mi::base::Handle<nv::index::ISparse_volume_scene_element> volume_scene_element(
    scene_edit->create_sparse_volume(ijk_bbox, transform_mat, importer_callback, dice_transaction)
    check_success(volume_scene_element.is_valid_interface());

    // Do not render the volume directly, just map it onto the heightfield
    volume_scene_element->set_enabled(false);

   const mi::neuraylib::Tag volume_tag = dice_transaction->store_for_reference_counting(volume_scen
    check_success(volume_tag.is_valid());

    static_group_node->append(volume_tag, dice_transaction);
  }

  // Append the heightfield to the scene group after all other scene elements
  static_group_node->append(heightfield_tag, dice_transaction);
  mi::neuraylib::Tag static_group_node_tag =
    dice_transaction->store_for_reference_counting(static_group_node.get());
  check_success(static_group_node_tag.is_valid());

  // Append the static scene group to the scene
  scene_edit->append(static_group_node_tag, dice_transaction);

  // Create a camera and adjust its parameters
  mi::base::Handle< nv::index::IPerspective_camera > cam(
    scene_edit->create_camera<nv::index::IPerspective_camera>());
  check_success(cam.is_valid_interface());
  setup_camera(cam.get());

  const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
  check_success(camera_tag.is_valid());

  scene_edit->set_camera(camera_tag);

  const mi::math::Vector<mi::Uint32, 2> buffer_resolution(1024, 1024);
  m_image_file_canvas->set_resolution(buffer_resolution);

  // Define the region of interest
  const mi::math::Bbox<mi::Float32, 3> region_of_interest(
    0.f, 0.f, 0.f,
    static_cast<mi::Float32>(heightfield_size.x),
    static_cast<mi::Float32>(heightfield_size.y),
    static_cast<mi::Float32>(elevation_range.y));
  scene_edit->set_clipped_bounding_box(region_of_interest);
}

nv::index::IFrame_results* Embedded_heightfield_geometry::render_frame(const std::string& output_fr
```

```
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3"); // log level
 sdict.insert("outfname", "frame_embedded_heightfield_geometry"); // output file base name
  sdict.insert("verify_image_fname", ""); // for unit test
  sdict.insert("unittest", "0"); // unit test mode

  sdict.insert("heightfield", "../embedded_heightfield_geometry/spiral.ppm");
  sdict.insert("mask", "../embedded_heightfield_geometry/spiral-mask.pgm");
  sdict.insert("texture", "none");
  sdict.insert("geometry", "fixed");
  sdict.insert("voxel_format", "uint8");
  sdict.insert("render", "z-axis");
  sdict.insert("size", "0.5");
  sdict.insert("zoom", "0");
  sdict.insert("pick", "1");
  sdict.insert("supersampling", "0");
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
  sdict.insert("is_call_from_test", "0"); // default: not call from make check.

 // Disabled until height_field can reference a sparse volume in the test code 2020-02-24
  if (sdict.get("geometry") == "volume")
  {
    ERROR_LOG << "geometry == volume is currently not supported.";
    return 1;
  }

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");
```

```
  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io data_analysis_and_processing");
  sdict.insert("index::app::plugins::base_importer::enabled",   "true");
  sdict.insert("index::app::plugins::legacy_importer::enabled", "true");

  // Initialize application
  Embedded_heightfield_geometry embedded_heightfield_geometry;
  embedded_heightfield_geometry.initialize(argc, argv, sdict);
  check_success(embedded_heightfield_geometry.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = embedded_heightfield_geometry.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.22 example_shared.h

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#ifndef NVIDIA_INDEX_EXAMPLES_UTILITY_EXAMPLE_SHARED_H
#define NVIDIA_INDEX_EXAMPLES_UTILITY_EXAMPLE_SHARED_H

#include <string>
#include <vector>
#include <map>
#include <set>

#include <mi/dice.h>
#include <nv/index/iindex.h>
#include <nv/index/isession.h>
#include <nv/index/iscene.h>

#include <nv/index/app/application_layer_utility.h>
#include <nv/index/app/iapplication_layer_component.h>
#include <nv/index/app/iapplication_layer_plugin.h>
#include <nv/index/app/istring_dict.h>
#include <nv/index/app/string_dict.h>

#include <nv/index/app/iindex_canvas.h>
#include <nv/index/app/icanvas_factory.h>

void exit_with_fail_message(const char* msg, const char* filename, int line);

#define check_success(expr) \
  { if (!(expr)) { exit_with_fail_message(#expr, __FILE__, __LINE__); } }

#define see_file_line_macro "(see file '" << __FILE__ << "', line: " << __LINE__ << ")"

template <class T>
inline void inline_template_argument_ignore_fn(T&)
{
  // empty
}

#ifdef  USE_NVINDEX_ACCESS
class Example_data
{
public:
  /// destructor
  virtual ~Example_data()
  {
    // empty
  }

  /// Get the example data instance
  static Example_data * instance()
  {
    if(G_p_example_data == 0) {
      G_p_example_data = new Example_data();
    }
```

```
    return G_p_example_data;
  }

  /// delete the singleton instance
  static void delete_instance()
  {
    if(G_p_example_data != 0)
    {
      delete G_p_example_data;
      G_p_example_data = 0;
    }
  }

public:
  /// Pointer to the DSO handle. Cached here for unload().
  void* m_p_dso_iindex_handle;
  /// The DSO filename. Cached here for unload().
  std::string m_nvindexlib_fname;

private:
  // Default constructor
  Example_data()
    :
    m_p_dso_iindex_handle(0),
    m_nvindexlib_fname()
  {
    // empty
  }

  // Copy constructor. Unused for now.
  Example_data(Example_data const &);
  // operator=. Unused for now.
  Example_data const & operator=(Example_data const &);

  // Singleton instance
  static Example_data * G_p_example_data;
};

inline nv::index::IIndex* load_and_get_iindex(const std::string& nvindexlib_fname)
{
#ifdef MI_PLATFORM_WINDOWS
  void* handle = LoadLibrary(TEXT(nvindexlib_fname.c_str()));
  if (!handle)
  {
   printf("error: Could not retrieve a handle to the %s library\n", nvindexlib_fname.c_str());
    return NULL;
  }
  void* symbol = GetProcAddress((HMODULE)handle, "nv_factory");
  if (!symbol)
  {
   printf("error: Could not retrieve the entry point into the %s library\n", nvindexlib_fname.c_str()
    return NULL;
  }
#else // MI_PLATFORM_WINDOWS
  void* handle = dlopen(nvindexlib_fname.c_str(), RTLD_LAZY);
  if (!handle)
  {
```

```
    printf("error: iindex: handle: %s\n", dlerror());
    return 0;
  }

  void* symbol = dlsym(handle, "nv_factory");
  if (!symbol)
  {
    printf("error: iindex: symbol: %s\n", dlerror());
    return NULL;
  }
#endif // MI_PLATFORM_WINDOWS

  Example_data::instance()->m_nvindexlib_fname    = nvindexlib_fname;
  Example_data::instance()->m_p_dso_iindex_handle = handle;

  return nv::index::nv_factory<nv::index::IIndex>(symbol);
}

inline bool unload_iindex()
{
#ifdef MI_PLATFORM_WINDOWS
  void* handle = Example_data::instance()->m_p_dso_iindex_handle;
  if (TRUE != FreeLibrary((HMODULE)handle))
  {
    fprintf(stderr, "error: unload_iindex.\n");
    return false;
  }
#else // MI_PLATFORM_WINDOWS

  int result = dlclose(Example_data::instance()->m_p_dso_iindex_handle);
  if (result != 0)
  {
    fprintf(stderr, "error: unload_iindex: %s\n", dlerror());
    return false;
  }
#endif // MI_PLATFORM_WINDOWS

  Example_data::delete_instance(); // clean up
  return true;
}

class Nvindex_access
{
public:
  /// Constructor
  Nvindex_access()
  {
    m_application_layer_loader = 0;
  }

  /// Destructor
  virtual ~Nvindex_access()
  {
    // empty
  }

  /// Load nvindex library
```

```
  /// \return true if loading was successful
  bool load_library()
  {
    // load shared libraries.
    std::string lib_name = "libnvindex";

#if _WIN32
#ifdef DEBUG
    lib_name += ".dll";
#else //  DEBUG
    lib_name += ".dll";
#endif // DEBUG

#else // _WIN32
    lib_name += ".so";
#endif // _WIN32

    m_nvindex_interface = load_and_get_iindex(lib_name);

    // Ref count check
    // {
    //     m_nvindex_interface->retain();
    //     const mi::Uint32 ref = m_nvindex_interface->release();
   //   fprintf(stderr, "Nvindex_access::load_library: m_nvindex_interface ref: %u\n", ref);
    // }

    check_success(m_nvindex_interface != 0);

    return true;
  }

  /// Load application layer library and application layer components
  ///
  /// \return true if loading application_layer library was successful
  bool load_application_layer_library()
  {
    check_success(m_nvindex_interface.is_valid_interface());

    m_application_layer_loader = new nv::index::app::Application_layer_loader();
    check_success (m_application_layer_loader->load() == 0);

   mi::base::Handle<nv::index::app::IApplication_layer> application_layer(m_application_layer_load
    check_success(application_layer.is_valid_interface());
    check_success(application_layer->initialize(m_nvindex_interface.get()) == 0);
    VERBOSE_LOG << "Nvindex_access: application_layer loaded/initalized.";

    // keep the interface to the member
    m_application_layer_interface.swap(application_layer);

    return true;
  }


  /// Load an applicaton layer components new
  ///
  /// \param[in] component_dso_base_name component dso file basename
  /// \return true when success
```

```
bool load_application_layer_component(
  const char* component_dso_basename)
{
  check_success(m_application_layer_interface.is_valid_interface());
  check_success(component_dso_basename != 0);

  const bool is_loaded =
  m_application_layer_interface->load_application_layer_component(component_dso_basename);
  if (is_loaded)
  {
    std::string component_name = "unknown name";
    nv::index::app::IApplication_layer_component* component
   = m_application_layer_interface->get_application_layer_component(component_dso_basename);
    if (component != 0)
    {
      const char* tmp = component->get_component_name();
      if (tmp != 0)
      {
        component_name = tmp;
      }
    }

  INFO_LOG << "Loaded NVIDIA IndeX application layer component: " << component_dso_basename << " ("
    return true;
  }

 ERROR_LOG << "Failed to load NVIDIA IndeX application layer component: " << component_dso_basename;
  return false;
}


/// Load an applicaton layer plugin
///
/// \param[in] plugin_dso_base_name plugin dso file basename
/// \return true when success
bool load_application_layer_plugin(
  const char* plugin_dso_basename)
{
  check_success(m_application_layer_interface.is_valid_interface());
  check_success(plugin_dso_basename != 0);

  const bool is_loaded =
  m_application_layer_interface->load_application_layer_plugin(plugin_dso_basename);
  if (is_loaded)
  {
    std::string plugin_name = "unknown name";
    nv::index::app::IApplication_layer_plugin* plugin
   = m_application_layer_interface->get_application_layer_plugin(plugin_dso_basename);
    if (plugin != 0)
    {
      const char* tmp = plugin->get_plugin_name();
      if (tmp != 0)
      {
        plugin_name = tmp;
      }
    }
```

```
    INFO_LOG << "Loaded NVIDIA IndeX application layer plugin: " << plugin_dso_basename << " (" << plug
      return true;
    }

   ERROR_LOG << "Failed to load NVIDIA IndeX application layer plugin: " << plugin_dso_basename;
    return false;
  }

  /// Is the nvindex library initialized?
  /// \return true when it has been initialized
  bool is_initialized() const
  {
    return m_nvindex_interface.is_valid_interface();
  }


 /// FIXME (not to return the reference of the interface, since the ref count is a bit tricky)
  /// Get to the nvindex interface
  /// \return reference to the nvindex interface handle
  mi::base::Handle<nv::index::IIndex>& get_interface()
  {
    return m_nvindex_interface;
  }

  /// Get the application layer interface
  /// \return application layer interface raw pointer
  nv::index::app::IApplication_layer* get_application_layer_interface()
  {
    return m_application_layer_interface.get(); // no ref count management
  }

  /// Start IndeX service
  ///  * start DiCE, IndeX, Application_layer (if available).
  ///
  /// \return success when 0
  mi::Sint32 start_service()
  {
    check_success(m_nvindex_interface.is_valid_interface());

    if (m_application_layer_interface.is_valid_interface())
    {
      // set_configuration call for the image component (openimageio plugin load)
      {
        mi::base::Handle<nv::index::app::IString_dict> prop(
          m_application_layer_interface->create<nv::index::app::IString_dict>());
        m_application_layer_interface->set_configuration(prop.get());
      }
    }

    m_nvindex_interface->start();
    VERBOSE_LOG << "IndeX service has been started.";

    // If you need the application_layer set_configuration(). It
    // should be here, it should be after IndeX start(), before
    // IApplication_layer start().  In that case, IndeX service start
    // and application_layer start should be separated.
```

```
    if (m_application_layer_interface.is_valid_interface())
    {
      m_application_layer_interface->start();
      VERBOSE_LOG << "Application_layer service has been started.";
    }

    // There are examples which doesn't need the application layer.
    // INFO_LOG << "Application_layer is not available.";
    return 0;
  }


  /// Shut down the nvindex and dice library
  /// \return true when shutdown was successful
  bool shutdown()
  {
    if (m_application_layer_interface.is_valid_interface())
    {
      INFO_LOG << "Nvindex_access::shutdown";
      m_application_layer_interface->shutdown();
      m_application_layer_interface->deinitialize();
      m_application_layer_interface.reset();

    delete m_application_layer_loader; // does the clear and unload... when doing unload before delet
      m_application_layer_loader = 0;
    }
    else
    {
      VERBOSE_LOG << "Nvindex_access::no application_layer shutdown";
    }
    check_success(m_nvindex_interface.is_valid_interface());
    {
      // removing logger
     mi::base::Handle< mi::neuraylib::ILogging_configuration > logging_configuration(
      this->get_interface()->get_api_component< mi::neuraylib::ILogging_configuration >());
      check_success(logging_configuration.is_valid_interface());
      logging_configuration->set_receiving_logger(0);
    }

    mi::Sint32 index_shutdown = m_nvindex_interface->shutdown();
    if (index_shutdown != 0)
    {
    fprintf(stderr, "error: failed NVIDIA IndeX shutdown (code: %d)\n", index_shutdown);
    }
    m_nvindex_interface.reset();

    return index_shutdown == 0;
  }


  /// create image file canvas with default values. Need to use a Handle.
  // DELETEME DEPRECATED
 nv::index::app::canvas_infrastructure::IIndex_image_file_canvas *create_image_file_canvas()
  {
    // check whether forget to load application_layer
    check_success(get_application_layer_interface() != 0);
   mi::base::Handle<nv::index::app::canvas_infrastructure::ICanvas_factory> canvas_factory(
```

```
      get_application_layer_interface()->
      get_api_component<nv::index::app::canvas_infrastructure::ICanvas_factory>());

    // check whether forget load canvas_infrastructure
    check_success(canvas_factory.is_valid_interface());
   mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> image_canvas(
    canvas_factory->create_canvas<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas>
    check_success(image_canvas.is_valid_interface());

    image_canvas->retain();     // ref-count up

    return image_canvas.get();
  }


private:
  // IIndex interface
  mi::base::Handle<nv::index::IIndex>                  m_nvindex_interface;
  // Application_layer loader
  nv::index::app::Application_layer_loader*            m_application_layer_loader;
  // Application layer interface
 mi::base::Handle<nv::index::app::IApplication_layer> m_application_layer_interface;
};
#endif  // USE_NVINDEX_ACCESS

nv::index::IIndex* load_and_get_iindex(
  std::string const & nvindexlib_fname);

bool unload_iindex();

inline mi::math::Vector_struct< mi::Float32, 3 > convert_to_vector_float32_3_st(
  const mi::math::Vector< mi::Float32, 3 > & float32_3)
{
  mi::math::Vector_struct< mi::Float32, 3 > float32_3_st = {
    float32_3.x, float32_3.y, float32_3.z,
  };
  return float32_3_st;
}

inline mi::math::Color_struct convert_to_color_st(
  const mi::math::Color & col)
{
  mi::math::Color_struct col_st = {
    col.r, col.g, col.b, col.a,
  };
  return col_st;
}

#ifdef  USE_NVINDEX_ACCESS
void info_cout(
  const std::string & mes,
  const std::map< std::string, std::string > & opt_map);
#endif  // USE_NVINDEX_ACCESS
void info_cout(
  const std::string & mes,
  const nv::index::app::String_dict& sdict);
```

```
mi::Sint64 get_volume_index(
  mi::math::Vector< mi::Sint64, 3 > const & ijk,
  mi::math::Bbox<   mi::Sint64, 3 > const & volume_raw_bbox);

mi::Sint64 get_heightfield_index(
  const mi::math::Vector< mi::Sint64, 2 >& ij,
  const mi::math::Bbox<   mi::Sint64, 2 >& heightfield_raw_bbox);

mi::neuraylib::Tag create_colormap(
  mi::Sint32                       colormap_idx,
  const nv::index::IScene*         scene,
  mi::neuraylib::IDice_transaction* dice_transaction);

mi::neuraylib::Tag create_colormap(
  const mi::math::Color&             color_1,
  const mi::math::Color&             color_2,
  mi::Uint32                         nb_values,
  const nv::index::IScene*           scene,
  mi::neuraylib::IDice_transaction*  dice_transaction);


#ifdef  USE_NVINDEX_ACCESS
void initialize_log_module(Nvindex_access & nvindex_accessor,
              const std::map< std::string, std::string > & opt_map);
void process_command_line(int argc, char *argv[],
             std::map< std::string, std::string > & opt_map);

#endif  // USE_NVINDEX_ACCESS
void process_command_line(int argc, char *argv[],
             nv::index::app::String_dict& sdict);

std::string get_output_file_name(const std::string & fname_base,
                mi::Sint32 frame_idx);

#ifdef  USE_NVINDEX_ACCESS
void enable_admin_port(Nvindex_access & nvindex_accessor);
void export_session(
  mi::neuraylib::Tag                                    session_tag,
  Nvindex_access&                                       nvindex_accessor,
  mi::neuraylib::IDice_transaction*                     dice_transaction,
  const std::string&                                    output_filename = "");

nv::index::IDistributed_data_import_callback* get_importer_from_application_layer(
  nv::index::app::IApplication_layer*       app_layer,
  const std::string&                        ns_importer_name,
  const std::map<std::string, std::string>& opt_map);


mi::Sint32 get_sint32_es(std::string const & instr);

std::string to_string_es(mi::Sint32 val);

inline bool has_key(const std::map< std::string, std::string >& opt_map, const std::string& key) {
  return opt_map.find(key) != opt_map.end();
}

#endif  // USE_NVINDEX_ACCESS
```

```
nv::index::IDistributed_data_import_callback* get_importer_from_application_layer(
  nv::index::app::IApplication_layer*       app_layer,
  const std::string&                        ns_importer_name,
  const nv::index::app::String_dict&        sdict);

std::vector<bool> get_bool_vec_from_string(const std::string& source_str);

inline nv::index::app::canvas_infrastructure::IIndex_image_file_canvas *
create_image_file_canvas(nv::index::app::IApplication_layer* app_layer)
{
  check_success(app_layer != 0);
 mi::base::Handle<nv::index::app::canvas_infrastructure::ICanvas_factory> canvas_factory(
   app_layer->get_api_component<nv::index::app::canvas_infrastructure::ICanvas_factory>());

  // check whether forget load canvas_infrastructure
  check_success(canvas_factory.is_valid_interface());
 mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> image_canvas(
   canvas_factory->create_canvas<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas>(
  check_success(image_canvas.is_valid_interface());

  image_canvas->retain();     // ref-count up

  return image_canvas.get();
}

#endif // NVIDIA_INDEX_EXAMPLES_UTILITY_EXAMPLE_SHARED_H
```

9.23 multi_view_heightfield.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/idistributed_data_access.h>
#include <nv/index/idistributed_compute_technique.h>
#include <nv/index/iheightfield_pick_result.h>
#include <nv/index/iindex.h>
#include <nv/index/iindex_debug_configuration.h>
#include <nv/index/ilight.h>
#include <nv/index/iline_set.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/itexture_filter_mode.h>
#include <nv/index/iviewport.h>

#include <nv/index/app/idata_analysis_and_processing.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/app_rendering_context.h"
#include "utility/canvas_utility.h"

#include <sstream>
#include <iostream>

class Multi_view_heightfield:
    public nv::index::app::Index_connect
{
public:
  Multi_view_heightfield()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Multi_view_heightfield() ctor";
  }

  virtual ~Multi_view_heightfield()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Multi_view_heightfield() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
```

```
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // visualize heightfield normal with lines
  bool create_normal_line_set(
    const mi::Float32*                              height_data,
    const mi::math::Vector_struct<mi::Float32, 3>* normal_data,
    const mi::math::Bbox<mi::Sint64, 2>&           hf_bbox,
    const mi::neuraylib::Tag&                      session_tag,
    mi::neuraylib::IDice_transaction*              dice_transaction);
  // visualize heightfield normal with lines
  void visualize_heightfield_normal(
    mi::neuraylib::Tag               heightfield_tag,
    mi::neuraylib::Tag               session_tag,
    mi::neuraylib::IDice_transaction* dice_transaction);
  // Setup camera to see this example scene
  //
  // \param[in] camera_tag camera tag
  // \param[in] view_idx   view index
  // \param[in] dice_transaction dice transaction
  void setup_camera(
    const mi::neuraylib::Tag&        camera_tag,
    mi::Size                         view_idx,
    mi::neuraylib::IDice_transaction* dice_transaction) const;
  // localize scene elements
  void localize_scene_element() const;

  //-------------------------------------------------------------------------
  // Filter enable viewport index list
  // \return newly created filtered viewport list
  nv::index::IViewport_list* filter_enabled_viewport_index_list() const;
  // Render one frame
  // \param[in] frame_idx        current frame index
  // \return true when success
  bool render_one_frame(mi::Sint32 frame_idx);
  // create multiple viewports
  //
  // The viewport structure of this example
  //                                          (511,511) (pixel coordinates)
  //        +-----------------+-----------------+
```

```
  //        | 0. global scope | 1. view 1       |
  // (0,256)|                 | camera change   |(511,256)
  //        +-----------------+-----------------+
  // (0,255)| 2. view 2       | 3. view 3       |(511,255)
  //        | add normal vis. | Mandelbrot tex. |
  //        +-----------------+-----------------+
  //       (0,0)       (255,0) (256,0)           (511,0)
  //
  //
  // \param[in,out] arc  application rendering context. multiple views will be updated.
  void create_views() const;
  //------------------------------------------------------------------------
  // set up the scene
  // Create a scene that contains a synthetically generated heightfield.
  // \param[in] dice_transaction db transaction
  void setup_scene(mi::neuraylib::IDice_transaction* dice_transaction);

  //------------------------------------------------------------------------
  // set up as the main host
  // \return true when success
  bool setup_main_host();
  // print pick results
  //
  // \param[in] scene_pick_results scene pick result for one view
  void print_pick_results(
    nv::index::IScene_pick_results* scene_pick_results) const;
  // pick test call.
  //
  // \param[in] pick_location_on_canvas picking location
  // \param[in] iindex_query            index query object
  // \param[in] enable_view_vec         enabled/disabled viewport state list
  // \param[in] expected_is_hit         hit is expected?
  // \return true when expectation is satisfied
  bool pick_test_call(
    const mi::math::Vector<mi::Sint32, 2>& pick_location_on_canvas,
    nv::index::IIndex_scene_query*         iindex_query,
    bool                                   expected_is_hit) const;

  // This session tag
 mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
 mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
 mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
 std::string                                                              m_outfname;
 bool                                                                  m_is_unittest;
 std::string                                                 m_verify_image_path_base;
 std::vector<bool>                                             m_enable_view_idx_vec;
  // camera tag
 mi::neuraylib::Tag                                                     m_camera_tag;
  // heightfield scene element tag
 mi::neuraylib::Tag                                                 m_heightfield_tag;
  // normal line tag
 mi::neuraylib::Tag                                                 m_normal_line_tag;
  // phong tag
 mi::neuraylib::Tag                                                     m_phong_1_tag;
```

```
  // mandelbrot mapping tag
 mi::neuraylib::Tag                                          m_mandelbrot_map_tag;
  // height value mapping tag
 mi::neuraylib::Tag                                         m_height_value_map_tag;
  // static group node tag for heightfield
 mi::neuraylib::Tag                                         m_hf_static_group_node_tag;
 mi::base::Handle<nv::index::IViewport_list>                        m_viewport_list;
};

mi::Sint32 Multi_view_heightfield::launch()
{
  // setup
  {
    bool is_all_viewport_enabeld = true;
   for (std::vector<bool>::const_iterator bi = m_enable_view_idx_vec.begin(); bi != m_enable_view_id
    {
      is_all_viewport_enabeld = is_all_viewport_enabeld && (*bi);
    }

    check_success(setup_main_host());

    // Create multiple views in the arc.
    create_views();

    mi::Sint32 frame_idx = 0;
    // Need to render a frame before the data access.
    {
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }

    // visualize normal as line set at the global scope
    {
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        visualize_heightfield_normal(m_heightfield_tag,
                      m_session_tag, dice_transaction.get());
      }
      dice_transaction->commit();
    }

    // localize scene elements
    localize_scene_element();

    // Render multiple views by a single render call
    {
      // Render a frame and save the rendered image to a file.
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }

    // Pick test
    {
```

```
      const mi::Sint32 nb_pick_point = 3;
     const mi::math::Vector<mi::Sint32, 2> pick_location_on_canvas[nb_pick_point] = {
        // viewport 0, heightfield hit
        mi::math::Vector<mi::Sint32, 2>(120, 400),
        // viewport 0, none
        mi::math::Vector<mi::Sint32, 2>( 45, 470),
        // viewport 1, compute plane hit
        mi::math::Vector<mi::Sint32, 2>(360, 390),
      };
      const bool is_hit[nb_pick_point] = {
        true,
        false,
        true,
      };

      mi::base::Handle<nv::index::IIndex_scene_query> iindex_query(
        get_index_interface()->get_api_component<nv::index::IIndex_scene_query>());
      check_success(iindex_query.is_valid_interface());
      for (mi::Sint32 i = 0; i < nb_pick_point; ++i)
      {
        const bool is_success =
          pick_test_call(pick_location_on_canvas[i],
                  iindex_query.get(),
                  is_hit[i]);

        if (!is_all_viewport_enabeld)
        {
          WARN_LOG << "Not all viewport is enabled, the pick test may fail.";
        }
        check_success(is_success);
      }
    }
  }

  return 0;                   // success (exit, otherwise)
}

bool Multi_view_heightfield::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    sdict.insert("dice::verbose", "2");
  }

  m_outfname               = sdict.get("outfname");
  m_verify_image_path_base = sdict.get("verify_image_path_base");
  m_enable_view_idx_vec    = get_bool_vec_from_string(sdict.get("enable_vidx_vec"));

  info_cout("running " + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_path_base = [" + m_verify_image_path_base +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
```

```
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            print out this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_path_base image path basename]\n"
     << "          when image_fname path basename exist, verify the rendering images.\n"
      << "            (default: [" << m_verify_image_path_base << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode. "
      << "(default: [" << m_is_unittest << "])\n"

      << "        [-enable_vidx_vec enabled_vector]\n"
      << "            Specify which viewport is enabled by a bool vector.\n"
      << "            (default: [" << sdict.get("enable_vidx_vec") << "])"

      << std::endl;
    exit(1);
  }

  return true;
}

bool Multi_view_heightfield::create_normal_line_set(
  const mi::Float32*                              height_data,
  const mi::math::Vector_struct<mi::Float32, 3>* normal_data,
  const mi::math::Bbox<mi::Sint64, 2>&           hf_bbox,
  const mi::neuraylib::Tag&                      session_tag,
  mi::neuraylib::IDice_transaction*              dice_transaction)
{
  check_success(dice_transaction != 0);
  check_success(session_tag.is_valid());
  check_success(height_data != 0);
  check_success(normal_data != 0);
  check_success(hf_bbox.volume() > 0);

  mi::base::Handle< nv::index::ISession const > session(
    dice_transaction->access<nv::index::ISession const>(session_tag));
  check_success(session.is_valid_interface());

  mi::base::Handle< nv::index::IScene > scene_edit(
    dice_transaction->edit<nv::index::IScene>(session->get_scene()));
  check_success(scene_edit.is_valid_interface());

  // hierarchical scene description node for the point set
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
```

```
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Line coordinates and its attributes. According to the
  // attributes, color and width are defined.
  // Here, we create two line sets.
  std::vector<mi::math::Vector_struct<mi::Float32, 3> > line_seg_vec;
  std::vector<mi::math::Color_struct> color_vec;
  mi::math::Color_struct red_color = { 1.0f, 0.0f, 0.0f, 1.0f, };
  std::vector<mi::Float32> width_vec;
  mi::Float32 vis_line_len = 60.0f;
  mi::Sint64 x_step = 16;
  mi::Sint64 y_step = 8;

  // data access
  mi::math::Vector<mi::Sint64, 2> ij(0, 0);
  for (ij[0] = hf_bbox.min[0]; ij[0] < hf_bbox.max[0]; ij[0] += x_step)
  {
    for (ij[1] = hf_bbox.min[1]; ij[1] < hf_bbox.max[1]; ij[1] += y_step)
    {
      {
        mi::Sint64 idx = get_heightfield_index(ij, hf_bbox);
        mi::math::Vector<mi::Float32, 3> org_pos(
          static_cast<mi::Float32>(ij[0]),
          static_cast<mi::Float32>(ij[1]),
          static_cast<mi::Float32>(height_data[idx]));

        mi::math::Vector<mi::Float32, 3> dst_pos = org_pos +
          vis_line_len * mi::math::Vector<mi::Float32, 3>(normal_data[idx]);

        line_seg_vec.push_back(org_pos);
        line_seg_vec.push_back(dst_pos);
        color_vec.push_back(red_color);
        width_vec.push_back(1.0f);
      }
      {
        mi::math::Vector<mi::Sint64, 2> ij_x_next = ij;
        ++(ij_x_next[0]);
        if (ij_x_next[0] < hf_bbox.max[0])
        {
          mi::Sint64 idx = get_heightfield_index(ij_x_next, hf_bbox);

          mi::math::Vector<mi::Float32, 3> org_pos(
            static_cast<mi::Float32>(ij_x_next[0]),
            static_cast<mi::Float32>(ij_x_next[1]),
            static_cast<mi::Float32>(height_data[idx]));

          mi::math::Vector<mi::Float32, 3> dst_pos = org_pos +
            vis_line_len * mi::math::Vector<mi::Float32, 3>(normal_data[idx]);

          line_seg_vec.push_back(org_pos);
          line_seg_vec.push_back(dst_pos);
          color_vec.push_back(red_color);
          width_vec.push_back(1.0f);
        }
      }
    }
```

```
  }

  {
    // Create line_set scene element and add it to the scene
   mi::base::Handle<nv::index::ILine_set> line_set(scene_edit->create_shape<nv::index::ILine_set>(
    check_success(line_set.is_valid_interface());
    line_set->set_line_type(nv::index::ILine_set::LINE_TYPE_SEGMENTS);

   line_set->set_lines(&(line_seg_vec[0]), line_seg_vec.size()); // may cut the last point
    line_set->set_colors(&(color_vec[0]), color_vec.size());
    line_set->set_widths(&(width_vec[0]), width_vec.size());

    // Add the line segments to the database
   const mi::neuraylib::Tag line_set_tag = dice_transaction->store_for_reference_counting(line_set.
    check_success(line_set_tag.is_valid());
    check_success(!(m_normal_line_tag.is_valid()));
    m_normal_line_tag = line_set_tag;

    // Add to the scene description
    group_node->append(line_set_tag, dice_transaction);
   INFO_LOG << "Added line_set (size: " << line_seg_vec.size() << ") to the scene (tag id: "
        << line_set_tag.id << ").";
  }

 mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_node.get
  check_success(group_node_tag.is_valid());
  scene_edit->append(group_node_tag, dice_transaction);

  return true;
}

void Multi_view_heightfield::visualize_heightfield_normal(
  mi::neuraylib::Tag                heightfield_tag,
  mi::neuraylib::Tag                session_tag,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(heightfield_tag.is_valid());
  check_success(session_tag.is_valid());
  check_success(dice_transaction != 0);

  mi::base::Handle<const nv::index::ISession> session(
    dice_transaction->access<nv::index::ISession>(session_tag));
  check_success(session.is_valid_interface());

  mi::base::Handle<const nv::index::IRegular_heightfield> heightfield(
    dice_transaction->access<nv::index::IRegular_heightfield>(heightfield_tag));
  check_success(heightfield.is_valid_interface());

 const mi::math::Bbox<mi::Float32, 3> heightfield_bbox = heightfield->get_IJK_bounding_box();

  const mi::math::Bbox<mi::Uint32, 2> query_bbox(
   static_cast<mi::Uint32>(heightfield_bbox.min.x), static_cast<mi::Uint32>(heightfield_bbox.min.y
   static_cast<mi::Uint32>(heightfield_bbox.max.x), static_cast<mi::Uint32>(heightfield_bbox.max.y

  // Access the distribution scheme
  const mi::neuraylib::Tag dist_layout_tag = session->get_distribution_layout();
  check_success(dist_layout_tag.is_valid());
```

```
  mi::base::Handle<const nv::index::IData_distribution> distribution_layout(
    dice_transaction->access<nv::index::IData_distribution>(dist_layout_tag));
  check_success(distribution_layout.is_valid_interface());

  // Find out on which hosts the data is located and print this information
  mi::base::Handle<nv::index::Regular_heightfield_locality_query_mode> mode(
   new nv::index::Regular_heightfield_locality_query_mode(heightfield_tag, query_bbox, false));
  mi::base::Handle<nv::index::IDistributed_data_locality> data_locality(
   distribution_layout->get_data_locality<nv::index::IRegular_heightfield>(mode.get(), dice_transa
  check_success(data_locality.is_valid_interface());

  std::ostringstream hosts;
  for (mi::Uint32 i = 0; i < data_locality->get_nb_cluster_nodes(); ++i)
  {
    check_success(data_locality->get_cluster_node(i) != 0);
    hosts << data_locality->get_cluster_node(i) << " ";
  }
 INFO_LOG << "Height field data for the current query bbox is located on these hosts: " << hosts.str();

  // Create the access factory
  const mi::neuraylib::Tag data_access_tag = session->get_data_access_factory();
  check_success(data_access_tag.is_valid());

 mi::base::Handle<const nv::index::IDistributed_data_access_factory> access_factory(
  dice_transaction->access<nv::index::IDistributed_data_access_factory>(data_access_tag));
  check_success(access_factory.is_valid_interface());

 mi::base::Handle<nv::index::IRegular_heightfield_data_access> heightfield_data_access(
    access_factory->create_regular_heightfield_data_access(heightfield_tag));
  check_success(heightfield_data_access.is_valid_interface());

  // Now retrieve the data
  heightfield_data_access->access(query_bbox, dice_transaction);

 const mi::math::Bbox<mi::Uint32, 2> effective_bbox = heightfield_data_access->get_patch_bounding_b
  check_success(effective_bbox == query_bbox);

  // Average the contents of the trace
  const mi::Float32* height_data = heightfield_data_access->get_elevation_values();
 const mi::math::Vector_struct<mi::Float32, 3>* normal_data = heightfield_data_access->get_normal_v
  const mi::math::Bbox<mi::Sint64, 2> hf_bbox(effective_bbox);

  create_normal_line_set(height_data,
              normal_data,
              hf_bbox,
              session_tag,
              dice_transaction);
}

void Multi_view_heightfield::setup_camera(
  const mi::neuraylib::Tag&         camera_tag,
  mi::Size                          view_idx,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(camera_tag.is_valid());
```

```
  mi::base::Handle<nv::index::IPerspective_camera> cam(
    dice_transaction->edit<nv::index::IPerspective_camera>(camera_tag));
  check_success(cam.is_valid_interface());

  // Set the camera parameters to see the whole scene
  const mi::math::Vector<mi::Float32, 3> from(700.0f,   700.0f,   200.0f);
  mi::math::Vector<mi::Float32, 3> dir  ( -1.125f,  -1.125f,  -1.0f);
  const mi::math::Vector<mi::Float32, 3> up  ( 0.0f, 0.0f, 1.0f);
  dir.normalize();

  cam->set(from, dir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(1000.0f);
}

void Multi_view_heightfield::localize_scene_element() const
{
  check_success(m_viewport_list.is_valid_interface());

  // Process local views first.

  // Set up view 1: change the camera
  {
    const mi::Size view_idx = 1;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "1");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      mi::Sint32 ret_localize = 1;
      ret_localize = dice_transaction->localize(m_camera_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      mi::base::Handle<nv::index::ICamera> cam(
        dice_transaction->edit<nv::index::ICamera>(m_camera_tag));
      check_success(cam.is_valid_interface());

      const mi::math::Vector<mi::Float32, 3> from(0.0f,   700.0f,   200.0f);
      mi::math::Vector<mi::Float32, 3> dir  (1.0f,  -1.125f,  -1.0f);
      const mi::math::Vector<mi::Float32, 3> up  ( 0.0f, 0.0f, 1.0f);
      dir.normalize();

      cam->set(from, dir, up);
```

```
    }
    dice_transaction->commit();
  }

  // Set up view 2: height color + normal lines
  {
    const mi::Size view_idx = 2;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);


    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "2");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      mi::Sint32 ret_localize = 1;
      ret_localize = dice_transaction->localize(m_normal_line_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_mandelbrot_map_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_height_value_map_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      mi::base::Handle<nv::index::ILine_set> line_set(
        dice_transaction->edit<nv::index::ILine_set>(m_normal_line_tag));
      check_success(line_set.is_valid_interface());
      line_set->set_enabled(true);

      mi::base::Handle<nv::index::IDistributed_compute_technique> man_map(
     dice_transaction->edit<nv::index::IDistributed_compute_technique>(m_mandelbrot_map_tag));
      man_map->set_enabled(false);

      mi::base::Handle<nv::index::IDistributed_compute_technique> hv_map(
     dice_transaction->edit<nv::index::IDistributed_compute_technique>(m_height_value_map_tag));
      hv_map->set_enabled(true);

      mi::base::Handle<nv::index::IPhong_gl> phong_1(
        dice_transaction->edit<nv::index::IPhong_gl>(m_phong_1_tag));
      check_success(phong_1.is_valid_interface());

      // Use just ambient white with texture
      phong_1->set_ambient(mi::math::Color(1.0f));
      phong_1->set_diffuse(mi::math::Color(0.0f));
      phong_1->set_specular(mi::math::Color(0.0f));
    }
    dice_transaction->commit();
```

```
  }

  // Set up view 3: Mandelbrot texture
  {
    const mi::Size view_idx = 3;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "3");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      mi::Sint32 ret_localize = 1;
      ret_localize = dice_transaction->localize(m_phong_1_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_mandelbrot_map_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_height_value_map_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      mi::base::Handle<nv::index::IDistributed_compute_technique> man_map(
     dice_transaction->edit<nv::index::IDistributed_compute_technique>(m_mandelbrot_map_tag));
      man_map->set_enabled(true);

      mi::base::Handle<nv::index::IDistributed_compute_technique> hv_map(
     dice_transaction->edit<nv::index::IDistributed_compute_technique>(m_height_value_map_tag));
      hv_map->set_enabled(false);

      mi::base::Handle<nv::index::IPhong_gl> phong_1(
        dice_transaction->edit<nv::index::IPhong_gl>(m_phong_1_tag));
      check_success(phong_1.is_valid_interface());

      // Use just ambient white with texture
      phong_1->set_ambient(mi::math::Color(1.0f));
      phong_1->set_diffuse(mi::math::Color(0.0f));
      phong_1->set_specular(mi::math::Color(0.0f));
    }
    dice_transaction->commit();
  }

  // Set up view 0, global: disable normal lines
  {
    const mi::Size view_idx = 0;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
```

```
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "0");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // edit scene elements
    {
      mi::base::Handle<nv::index::ILine_set> line_set(
        dice_transaction->edit<nv::index::ILine_set>(m_normal_line_tag));
      check_success(line_set.is_valid_interface());
      line_set->set_enabled(false);
    }
    dice_transaction->commit();
  }
}

nv::index::IViewport_list* Multi_view_heightfield::filter_enabled_viewport_index_list() const
{
  mi::base::Handle<nv::index::IViewport_list> new_viewport_list;
  {
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    assert(dice_transaction.is_valid_interface());
    {
      assert(m_session_tag.is_valid());
      mi::base::Handle<const nv::index::ISession> session(
        dice_transaction->access<nv::index::ISession>(m_session_tag));
      assert(session.is_valid_interface());

      // create viewport from a session
      new_viewport_list = session->create_viewport_list();
    }
    dice_transaction->commit();
  }

  check_success(new_viewport_list.is_valid_interface());
  const mi::Size nb_views = m_viewport_list->size();

  std::stringstream sstr;
  mi::Size nb_enabled = 0;
  for (mi::Size i = 0; i < nb_views; ++i)
  {
    if (m_enable_view_idx_vec[i])
    {
      sstr << i << " ";
      mi::base::Handle<nv::index::IViewport> vp (m_viewport_list->get(i));
      new_viewport_list->append(vp.get());
      ++nb_enabled;
    }
  }
  // copy advisory state
  new_viewport_list->set_advisory_enabled(m_viewport_list->get_advisory_enabled());
```

```
  if (nb_enabled == nb_views)
  {
    INFO_LOG << "All views are enabled.";
  }
  else
  {
    INFO_LOG << "Filtered views. Enabled: " << sstr.str();
  }

  new_viewport_list->retain();
  return new_viewport_list.get();
}

bool Multi_view_heightfield::render_one_frame(mi::Sint32 frame_idx)
{
  bool success = true;

  // Render a frame and save the rendered image to a file.
  // Only save the file at the end of the iteration.
  std::string fname = "";
  check_success(!(m_outfname.empty()));
  {
    fname = get_output_file_name(m_outfname, frame_idx);
  }
  check_success(m_index_rendering.is_valid_interface());

  // set up canvas. output_fname.empty() is valid (no output file)
  m_image_file_canvas->set_rgba_file_name(fname.c_str());

  // set advisory to see the multiview rendering details
  m_viewport_list->set_advisory_enabled(true);

  mi::base::Handle<nv::index::IViewport_list> cur_viewport_list(
    filter_enabled_viewport_index_list());
  check_success(cur_viewport_list.is_valid_interface());

  mi::base::Handle<nv::index::IFrame_results_list> frame_results_list(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      cur_viewport_list.get()));
  check_success(frame_results_list.is_valid_interface());

  if (frame_results_list->size() == 0)
  {
    ERROR_LOG << "IIndex_rendering rendering call has no results.";
    success = false;
  }
  else
  {
    for (mi::Size i = 0; i < frame_results_list->size(); ++i)
    {
      mi::base::Handle<nv::index::IFrame_results>
        frame_results(frame_results_list->get(i));
      check_success(frame_results.is_valid_interface());
```

```
    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      check_success(err_set.is_valid_interface());
      if (err_set->any_errors())
      {
        std::ostringstream os;

     const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
            << os.str();
        success = false;
      }
    }
  }

  // verify the generated frame
  if (!(m_verify_image_path_base.empty()))
  {
   const std::string ref_img_fpath = get_output_file_name(m_verify_image_path_base, frame_idx);
    if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), ref_img_fpath, get_options())))
    {
      success = false;
    }
  }

  return success;
}

void Multi_view_heightfield::create_views() const
{
  // Check session and multiple views in the arc
  check_success(m_session_tag.is_valid());
  check_success(m_viewport_list.is_valid_interface());
  check_success(m_viewport_list->size() == 0);

  // Multiple view itself lives in the global scope.
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<nv::index::ISession>(m_session_tag));
    check_success(session.is_valid_interface());

    // 0. create a single view in the global scope.
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());
```

```
    const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0, 256);
    const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

    viewport->set_position(viewport_pos);
    viewport->set_size(viewport_size);
    viewport->set_scope(m_global_scope.get()); // ref count up

    m_viewport_list->append(viewport.get());

    // set up the camera
    const mi::Size view_idx = 0;
    setup_camera(m_camera_tag, view_idx, dice_transaction.get());
  }

  // 1. viewport
  {
    mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
    check_success(viewport.is_valid_interface());

    mi::neuraylib::IScope* parent = 0; // this means global scope in this context
    mi::Uint8 privacy_level = 1;
    bool      is_temp       = false;
  mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
    check_success(local_scope.is_valid_interface());

    const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256, 256);
    const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

    viewport->set_position(viewport_pos);
    viewport->set_size(viewport_size);
    viewport->set_scope(local_scope.get()); // ref count up

    m_viewport_list->append(viewport.get());
  }

  // 2. viewport
  {
    mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
    check_success(viewport.is_valid_interface());

    mi::neuraylib::IScope* parent = 0; // this means global scope in this context
    mi::Uint8 privacy_level = 1;
    bool      is_temp       = false;
  mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
    check_success(local_scope.is_valid_interface());

    const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0,   0);
    const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

    viewport->set_position(viewport_pos);
    viewport->set_size(viewport_size);
    viewport->set_scope(local_scope.get()); // ref count up

    m_viewport_list->append(viewport.get());
  }

  // 3. viewport
```

```
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256,   0);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }
  }
  dice_transaction->commit();
}


void Multi_view_heightfield::setup_scene(
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(dice_transaction != 0);

  // Access the session instance from the database.
  mi::base::Handle<const nv::index::ISession> session(
    dice_transaction->access<const nv::index::ISession>(m_session_tag));
  check_success(session.is_valid_interface());

  // Access (edit mode) the scene instance from the database.
  mi::base::Handle<nv::index::IScene> scene_edit(
    dice_transaction->edit<nv::index::IScene>(session->get_scene()));
  check_success(scene_edit.is_valid_interface());

  // Create static group node for large data
  mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
    scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
  check_success(static_group_node.is_valid_interface());

  // Details for creating a synthetic heightfield.
  const mi::math::Vector<mi::Uint32, 2> heightfield_size(500, 500);

  nv::index::app::String_dict heightfield_opt;
  heightfield_opt.insert("args::type",           "heightfield");
 heightfield_opt.insert("args::importer",     "nv::index::plugin::legacy_importer.Synthetic_heigh
  heightfield_opt.insert("args::synthetic_type", "i");
  heightfield_opt.insert("args::size",           "500 500");
  heightfield_opt.insert("args::range",          "0.1 1000");
  nv::index::IDistributed_data_import_callback* importer_callback =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::legacy_importer.Synthetic_heightfield_generator",
```

```
      heightfield_opt);


  // Create heightfield scene element and add it to the scene
  const mi::Float32 rotate_k = 0.0f;
  const mi::math::Vector<mi::Float32, 3> translate(0.0f, 0.0f, 0.0f);
  const mi::math::Vector<mi::Float32, 3> scale(1.0f, 1.0f, 1.0f);
  // const mi::math::Vector<mi::Float32, 2> elevation_range(50.0f, 256.0f);
  const mi::math::Vector<mi::Float32, 2> elevation_range(0.0f, 256.0f);

  mi::base::Handle<nv::index::IRegular_heightfield> heightfield_scene_element(
    scene_edit->create_regular_heightfield(
      scale, rotate_k, translate,
      heightfield_size,
      elevation_range,
      importer_callback,
      dice_transaction));
  check_success(heightfield_scene_element.is_valid_interface());

 const mi::math::Bbox<mi::Float32, 3> heightfield_bbox = heightfield_scene_element->get_IJK_boundin

  // Set the name of the heightfield scene element
  const std::string heightfield_name = "synthetic heightfield";
  heightfield_scene_element->set_name(heightfield_name.c_str());

  // Store the heightfield scene element in the DB
  const mi::neuraylib::Tag heightfield_tag =
    dice_transaction->store_for_reference_counting(heightfield_scene_element.get());
  check_success(heightfield_tag.is_valid());
  check_success(!m_heightfield_tag.is_valid());
  m_heightfield_tag = heightfield_tag;

  // Add a light and a material to the static group node
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color color_intensity(1.0f, 1.0f, 1.0f, 1.0f);
    headlight->set_intensity(color_intensity);
    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());
    static_group_node->append(headlight_tag, dice_transaction);

    // Material for the heightfield
   mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
    check_success(phong_1.is_valid_interface());

    // Define a more interesting greenish material
    phong_1->set_ambient(mi::math::Color(0.f, 0.3f, 0.0f, 0.3f));
    phong_1->set_diffuse(mi::math::Color(0.f, 0.8f, 0.2f, 0.3f));
    phong_1->set_specular(mi::math::Color(0.6f));
    phong_1->set_shininess(100.f);

    const mi::neuraylib::Tag phong_1_tag
      = dice_transaction->store_for_reference_counting(phong_1.get());
```

```
    check_success(phong_1_tag.is_valid());
    check_success(!(m_phong_1_tag.is_valid()));
    m_phong_1_tag = phong_1_tag;

    static_group_node->append(phong_1_tag, dice_transaction);
  }

  // Put a computed texture onto the heightfield. Both disabled in default
  // Mandelbrot texture
  {
    // Create the computed texture, with a size fitting the heightfield
    mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
    scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
    check_success(tex_filter.is_valid_interface());
  mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.ge
    check_success(tex_filter_tag.is_valid());
    static_group_node->append(tex_filter_tag, dice_transaction);

  mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> p
    get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_proces
    check_success(processing.is_valid_interface());
  // Create a mandelbrot technique and add it to the scene. For more details on how to implement the ch
  // source that was shipped with the application layer component 'nv::index::app::data_analysis_and
    mi::base::Handle<nv::index::IDistributed_compute_technique> man_map(
    processing->get_sample_tool_set()->create_mandelbrot_2d_technique(mi::math::Vector<mi::Float32
    check_success(man_map.is_valid_interface());
    man_map->set_enabled(false);

    // Add the mapping to the scene before the heightfield
  mi::neuraylib::Tag man_map_tag = dice_transaction->store_for_reference_counting(man_map.get());
    check_success(man_map_tag.is_valid());
    check_success(!m_mandelbrot_map_tag.is_valid());
    m_mandelbrot_map_tag = man_map_tag;

    static_group_node->append(man_map_tag, dice_transaction);
  }

  // Map height values in the heightfield to colors
  {
    // Create the computed texture, passing the height range of the heightfield
    const mi::math::Vector<mi::Float32, 2> height_range(
      heightfield_bbox.min.z, heightfield_bbox.max.z);

    mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
    scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
    check_success(tex_filter.is_valid_interface());
  mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.ge
    check_success(tex_filter_tag.is_valid());
    static_group_node->append(tex_filter_tag, dice_transaction);

    // Create the computed texture, with a size fitting the heightfield
  mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> p
    get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_proces
    check_success(processing.is_valid_interface());
  // Create a mandelbrot technique and add it to the scene. For more details on how to implement the ch
  // source that was shipped with the application layer component 'nv::index::app::data_analysis_and
    const mi::math::Color color0(0.f, 1.f, 0.f, 1.f); // green
```

```
  const mi::math::Color color1(1.f, 0.f, 0.f, 1.f); // red
  mi::base::Handle<nv::index::IDistributed_compute_technique> hv_map(
  processing->get_sample_tool_set()->create_color_encoded_elevation_technique(height_range, colo
  check_success(hv_map.is_valid_interface());
  hv_map->set_enabled(true);

  // Add the mapping to the scene before the heightfield
 mi::neuraylib::Tag hv_map_tag = dice_transaction->store_for_reference_counting(hv_map.get());
  check_success(hv_map_tag.is_valid());
  check_success(!m_height_value_map_tag.is_valid());
  m_height_value_map_tag = hv_map_tag;

  static_group_node->append(hv_map_tag, dice_transaction);
}

// Append the heightfield to the scene group
static_group_node->append(heightfield_tag, dice_transaction);
mi::neuraylib::Tag static_group_node_tag =
  dice_transaction->store_for_reference_counting(static_group_node.get());
check_success(static_group_node_tag.is_valid());
check_success(!m_hf_static_group_node_tag.is_valid());
m_hf_static_group_node_tag = static_group_node_tag;

// Append the static scene group to the scene.
scene_edit->append(static_group_node_tag, dice_transaction);

std::stringstream sstr;
sstr << "Created an synthetic heightfield: size = "
  << heightfield_size << ", tag = " << heightfield_tag.id;
INFO_LOG << sstr.str();

// Create a camera and adjust the camera parameter.
mi::base::Handle< nv::index::IPerspective_camera > cam(
  scene_edit->create_camera<nv::index::IPerspective_camera>());
check_success(cam.is_valid_interface());
m_camera_tag = dice_transaction->store(cam.get());

check_success(m_camera_tag.is_valid());
const mi::math::Vector<mi::Uint32,2> buffer_resolution(512, 512);
m_image_file_canvas->set_resolution(buffer_resolution);

// Define a region of interest for the entire scene in the
// scene's global coordinate system.
const mi::math::Bbox<mi::Float32, 3> region_of_interest(
  0.f, 0.f, 0.f,
  static_cast< mi::Float32 >(heightfield_size.x),
  static_cast< mi::Float32 >(heightfield_size.y),
 static_cast< mi::Float32 >(heightfield_size.x)  // Note: using the x dimension to define the z size
  );
scene_edit->set_clipped_bounding_box(region_of_interest);

// Finally, optionally adjust the scene's coordinate system
mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
  1.0f,  0.0f,  0.0f,  0.0f,
  0.0f,  1.0f,  0.0f,  0.0f,
  0.0f,  0.0f, -1.0f,  0.0f, // adjust for coordinate system
  0.0f,  0.0f,  0.0f,  1.0f
```

```
    );

  scene_edit->set_transform_matrix(transform_mat);

  // ... and add the camera to the scene.
  scene_edit->set_camera(m_camera_tag);
}

bool Multi_view_heightfield::setup_main_host()
{
 // Access the IndeX rendering query interface for querying performance values and pick results
  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  // DiCE database access
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    //------------------------------------------------------------------------
    // Setup session information
    m_session_tag =
      m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<nv::index::ISession>(
        m_session_tag));
    check_success(session.is_valid_interface());

    // Setup the main multiple views, but it is empty.
    m_viewport_list = session->create_viewport_list();
    check_success(m_viewport_list.is_valid_interface());

    //------------------------------------------------------------------------
    // Scene setup in the global scope
    setup_scene(dice_transaction.get());
  }
  dice_transaction->commit();

  // set canvas size
  m_image_file_canvas->set_resolution(mi::math::Vector<mi::Uint32, 2>(512, 512));

  INFO_LOG << "Initialization complete.";

  return true;
}

void Multi_view_heightfield::print_pick_results(
```

```
  nv::index::IScene_pick_results* scene_pick_results) const
{
  assert(scene_pick_results != 0);
  const mi::Uint32 nb_results = scene_pick_results->get_nb_results();
  if (nb_results > 0)
  {
    INFO_LOG << "Number of pick results: " << nb_results << " on viewport: "
        << scene_pick_results->get_viewport_index();
    for (mi::Uint32 i = 0; i < nb_results; i++)
    {
      // Generic information
    const mi::base::Handle<nv::index::IScene_pick_result> result(scene_pick_results->get_result(i)

      std::stringstream log_out;
      log_out << "Intersection no. " << i << "\n"
         << "\t\t Element (tag)          " << result->get_scene_element().id << "\n"
       << "\t\t Sub index:          " << result->get_scene_element_sub_index() << "\n"
          << "\t\t Distance:               " << result->get_distance() << "\n"
          << "\t\t Position (local space): " << result->get_intersection() << "\n"
          << "\t\t Color (evaluated):      " << result->get_color() << "\n";

    const mi::base::Handle<const nv::index::IData_sample> data_sample(result->get_data_sample());
      if (data_sample)
      {
     const mi::base::Handle<const nv::index::IData_sample_uint8>  ds_uint8(data_sample->get_interf
     const mi::base::Handle<const nv::index::IData_sample_uint16> ds_uint16(data_sample->get_inte
     const mi::base::Handle<const nv::index::IData_sample_float32> ds_float32(data_sample->get_int
     const mi::base::Handle<const nv::index::IData_sample_rgba8>  ds_rgba8(data_sample->get_interf
        if (ds_uint8)
        {
       log_out << "\t Data sample:    " << mi::Uint32(ds_uint8->get_sample_value()) << "\n";
        }
        else if (ds_uint16)
        {
          log_out << "\t Data sample:     " << ds_uint16->get_sample_value() << "\n";
        }
        else if (ds_float32)
        {
         log_out << "\t Data sample:     " << ds_float32->get_sample_value() << "\n";
        }
        else if (ds_rgba8)
        {
          log_out << "\t Data sample:     " << ds_rgba8->get_sample_value() << "\n";
        }
      }

      INFO_LOG << log_out.str();

      // Shape-specific information
    if (result->get_intersection_info_class() == nv::index::IHeightfield_pick_result::IID())
      {
     mi::base::Handle<const nv::index::IHeightfield_pick_result> compute_pick_result(
          result->get_interface<const nv::index::IHeightfield_pick_result>());
        if (compute_pick_result && compute_pick_result->is_computing_enabled())
        {
          INFO_LOG << "Specific pick results for computed heightfield texture:";
       INFO_LOG << "\t Computed color value:   " << compute_pick_result->get_computed_color();
```

```
        }
      }
    }
  }
  else
  {
    INFO_LOG << "No pick result.";
  }
}

bool Multi_view_heightfield::pick_test_call(
  const mi::math::Vector<mi::Sint32, 2>& pick_location_on_canvas,
  nv::index::IIndex_scene_query*         iindex_query,
  bool                                   expected_is_hit) const
{
  assert(iindex_query != 0);
  INFO_LOG << "Picking at canvas position: " << pick_location_on_canvas;

  mi::base::Handle<nv::index::IViewport_list> cur_viewport_list(
    filter_enabled_viewport_index_list());
  check_success(cur_viewport_list.is_valid_interface());

  // Run the picking operation
  mi::base::Handle<nv::index::IScene_pick_results_list> scene_pick_results_list(
    iindex_query->pick(pick_location_on_canvas,
              m_image_file_canvas.get(),
              cur_viewport_list.get(),
              m_session_tag));
  assert(scene_pick_results_list.is_valid_interface());

  const mi::Size nb_list = scene_pick_results_list->size();
  mi::Size sum_results = 0;

  for (mi::Size i = 0; i < nb_list; ++i)
  {
    // Print the picking result
    mi::base::Handle<nv::index::IScene_pick_results> scene_pick_results(
      scene_pick_results_list->get(i));
    print_pick_results(scene_pick_results.get());
    sum_results += scene_pick_results->get_nb_results();
  }

  if ((sum_results > 0) != expected_is_hit)
  {
   INFO_LOG << "expected_is_hit: " << expected_is_hit << ", but sum_results: " << sum_results;
    return false;
  }

  return true;
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "4");                          // log level
 sdict.insert("outfname", "frame_multi_view_heightfield");   // output file base name
  sdict.insert("verify_image_path_base", "");                  // for unit test
```

```
  sdict.insert("unittest", "0");                                  // unit test mode
 sdict.insert("enable_vidx_vec", "1 1 1 1");             // enabled view indices vector

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io data_analysis_and_processing");
  sdict.insert("index::app::plugins::base_importer::enabled",   "true");
  sdict.insert("index::app::plugins::legacy_importer::enabled", "true");

  // Initialize application
  Multi_view_heightfield multi_view_heightfield;
  multi_view_heightfield.initialize(argc, argv, sdict);
  check_success(multi_view_heightfield.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = multi_view_heightfield.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.24 multi_view_shape.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/ifont.h>
#include <nv/index/ilabel.h>

#include <nv/index/icamera.h>
#include <nv/index/icircle.h>
#include <nv/index/icone.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/icylinder.h>
#include <nv/index/idepth_offset.h>
#include <nv/index/idistributed_data_access.h>
#include <nv/index/iellipse.h>
#include <nv/index/iellipsoid.h>
#include <nv/index/iicon.h>
#include <nv/index/iindex.h>
#include <nv/index/iindex_debug_configuration.h>
#include <nv/index/ilabel.h>
#include <nv/index/ilight.h>
#include <nv/index/iline_set.h>
#include <nv/index/imaterial.h>
#include <nv/index/ipath.h>
#include <nv/index/iplane.h>
#include <nv/index/ipoint_set.h>
#include <nv/index/ipolygon.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/isphere.h>
#include <nv/index/itexture.h>
#include <nv/index/itexture_filter_mode.h>
#include <nv/index/iviewport.h>

#include <nv/index/app/idata_analysis_and_processing.h>
#include <nv/index/app/iimage_importer.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/app_rendering_context.h"
#include "utility/canvas_utility.h"

#include <sstream>
#include <iostream>

class Multi_view_shape:
    public nv::index::app::Index_connect
{
public:
```

```
  Multi_view_shape()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Multi_view_shape() ctor";
  }

  virtual ~Multi_view_shape()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Multi_view_shape() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Setup camera to see this example scene
  //
  // \param[in] camera_tag camera tag
  // \param[in] dice_transaction dice transaction
  void setup_camera(
    const mi::neuraylib::Tag&         camera_tag,
    mi::neuraylib::IDice_transaction* dice_transaction) const;
  // localize scene elements
  void localize_scene_element() const;
  // Filter enable viewport index list
  // \return newly created filtered viewport list
  nv::index::IViewport_list* filter_enabled_viewport_index_list();
  // Render one frame
  // \param[in] frame_idx        current frame index
  // \return true when success
  bool render_one_frame(mi::Sint32 frame_idx);
  // create multiple views
  //
  // The view structure of this example
```

```
  //
  //        +-----------------+-----------------+(511,511)
  // (0,256)| 0. global scope | 1. view 1       |
  //        +-----------------+-----------------+(511,255)
  // (0,255)| 2. view 2       | 3. view 3       |
  //        +-----------------+-----------------+
  //       (0,0)       (255,0)                 (511,0)
  //
  //
 // \param[in,out] arc  application rendering context. multiple views will be updated.
  void create_views() const;

  // create lights and materials and put them into the tag map
  //
  // \param[in]  scene_edit scene root for edit
  // \param[in]  dice_transaction dice db transaction
  // \param[out] light_mat_tag_map {M: light and material name -> tag}
  void create_light_material_instance(
    nv::index::IScene*                          scene_edit,
    mi::neuraylib::IDice_transaction*           dice_transaction,
    std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map) const;
  mi::neuraylib::Tag get_valid_light_mat_tag_by_name(
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    const std::string& light_mat_name) const;
  void add_scene_defaut_light(
    nv::index::IScene*                                scene_edit,
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    mi::neuraylib::IDice_transaction*                 dice_transaction);
  void add_scene_group_simple_geometry(
    nv::index::IScene*                                scene_edit,
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    mi::neuraylib::IDice_transaction*                 dice_transaction);
  void add_scene_group_planes(
    nv::index::IScene*                                scene_edit,
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    mi::neuraylib::IDice_transaction*                 dice_transaction);
  void add_scene_group_raster(
    nv::index::IScene*                                scene_edit,
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    mi::neuraylib::IDice_transaction*                 dice_transaction);
  void add_scene_group_paths(
    nv::index::IScene*                                scene_edit,
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    mi::neuraylib::IDice_transaction*                 dice_transaction);
  void add_scene_group_labels(
    nv::index::IScene*                                scene_edit,
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    mi::neuraylib::IDice_transaction*                 dice_transaction);
  void add_scene_group_icons(
    nv::index::IScene*                                scene_edit,
    const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
    mi::neuraylib::IDice_transaction*                 dice_transaction);
  // set up the scene
  // Create a scene that contains a triangle mesh.
  // \param[in] dice_transaction  db transaction
  void setup_scene(mi::neuraylib::IDice_transaction* dice_transaction);
  // set up as the main host
```

```
  // \return true when success
  bool setup_main_host();

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                 m_verify_image_path_base;
  std::string                                                           m_font_fpath;
  std::string                                                             m_iconfile;
  std::vector<bool>                                                m_enable_view_idx_vec;
  // camera tag
  mi::neuraylib::Tag                                                    m_camera_tag;
  // simple geometry group node tag
  mi::neuraylib::Tag                                               m_simple_geo_group_node_tag;
  // plane tag
  mi::neuraylib::Tag                                                     m_plane_tag;
  // raster object circle tag
  mi::neuraylib::Tag                                              m_raster_circle1_tag;
  // raster object ellipses tag
  mi::neuraylib::Tag                                              m_raster_ellipses1_tag;
  // IPath3D path tag
  mi::neuraylib::Tag                                                    m_path3d_1_tag;
  // ILabel2D tag
  mi::neuraylib::Tag                                                    m_label2d_tag;
  // Polygon group tag
  mi::neuraylib::Tag                                               m_polygon_group_tag;
  mi::base::Handle<nv::index::IViewport_list>                             m_viewport_list;
};

mi::Sint32 Multi_view_shape::launch()
{
  // setup
  {
    check_success(setup_main_host());

    // Create multiple views in the arc.
    create_views();

    mi::Sint32 frame_idx = 0;
    // Render multiple views for a single render call. Before localize.
    {
      // Render a frame and save the rendered image to a file.
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }

    // localize scene elements
    localize_scene_element();

    // Render multiple views for a single render call.
```

```
    {
      // Render a frame and save the rendered image to a file.
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }
  }
  // index_connect shutdown
  return 0;
}

bool Multi_view_shape::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));
  if (m_is_unittest)
  {
    sdict.insert("dice::verbose", "2");
  }

  m_outfname               = sdict.get("outfname");
  m_verify_image_path_base = sdict.get("verify_image_path_base");
  m_enable_view_idx_vec    = get_bool_vec_from_string(sdict.get("enable_vidx_vec"));
  m_font_fpath             = sdict.get("font_fpath");
  m_iconfile               = sdict.get("iconfile");

  info_cout("running " + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_path_base = [" + m_verify_image_path_base +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            print out this message\n"

      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-font_fpath FONT_FILE_PATH]\n"
      << "           font file path. (default: " << sdict.get("font_fpath") << ")\n"

      << "        [-icon_file ICON_FILE_PATH]\n"
    << "         Specify an icon file. (default: [" << sdict.get("iconfile") << "])\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << sdict.get("outfname") << "])\n"

      << "        [-verify_image_path_base image path basename]\n"
    << "         when image_fname path basename exist, verify the rendering images.\n"
      << "            (default: [" << sdict.get("verify_image_path_base") << "])\n"
```

```
      << "        [-unittest bool]\n"
      << "            when true, unit test mode. "
      << "(default: [" << sdict.get("unittest") << "])\n"

      << "        [-enable_vidx_vec enabled_vector]\n"
      << "            Specify which viewport is enabled by a bool vector.\n"
      << "            (default: [" << sdict.get("enable_vidx_vec") << "])"

      << std::endl;
    exit(1);
  }

  return true;
}

void Multi_view_shape::setup_camera(
  const mi::neuraylib::Tag&         camera_tag,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(camera_tag.is_valid());

  mi::base::Handle<nv::index::IPerspective_camera> cam(
    dice_transaction->edit<nv::index::IPerspective_camera>(camera_tag));
  check_success(cam.is_valid_interface());

  const mi::math::Vector<mi::Float32, 3> from(500.0f, 1500.0f, 250.0f);
  mi::math::Vector<mi::Float32, 3>       dir (  0.0f,   -0.9f,   0.2f);
  const mi::math::Vector<mi::Float32, 3> up  (  0.0f,   -0.2f,  -1.0f);
  dir.normalize();

  cam->set(from, dir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(1000.0f);
}

void Multi_view_shape::localize_scene_element() const
{
  // Process local views first.

  // Set up view 1:
  {
    const mi::Size view_idx = 1;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "1");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
```

```
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements for viewport 1
    {
      mi::Sint32 ret_localize = 1;
      ret_localize = dice_transaction->localize(m_simple_geo_group_node_tag,
                           mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_plane_tag,
                           mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_raster_circle1_tag,
                           mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_label2d_tag,
                           mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      // change the transformation in viewport 1
    mi::base::Handle<nv::index::ITransformed_scene_group> simple_geometry_group_node(
     dice_transaction->edit<nv::index::ITransformed_scene_group>(m_simple_geo_group_node_tag));
      check_success(simple_geometry_group_node.is_valid_interface());
      mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
        1.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f, 0.0f,
        200.0f - 100.0f, 200.0f, 600.0f, 1.0f // move x -100
        );
      simple_geometry_group_node->set_transform(transform_mat);

      mi::base::Handle<nv::index::IPlane> plane(
        dice_transaction->edit<nv::index::IPlane>(m_plane_tag));
    plane->set_extent(mi::math::Vector<mi::Float32,2>(600.0f - 100.0f, 800.0f - 200.0f)); // make it

      mi::base::Handle<nv::index::ICircle> circle(
        dice_transaction->edit<nv::index::ICircle>(m_raster_circle1_tag));
      check_success(circle.is_valid_interface());
      circle->set_enabled(false);

      mi::base::Handle<nv::index::ILabel_2D> label_2d(
        dice_transaction->edit<nv::index::ILabel_2D>(m_label2d_tag));
      check_success(label_2d.is_valid_interface());
      label_2d->set_text("View 1: transform + disable some");

    }
    dice_transaction->commit();
  }

  // Set up viewport 2:
  {
    const mi::Size view_idx = 2;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
```

```
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "2");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      mi::Sint32 ret_localize = 1;
      ret_localize = dice_transaction->localize(m_raster_ellipses1_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_path3d_1_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_label2d_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_polygon_group_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      mi::base::Handle<nv::index::IEllipse> ellipse(
        dice_transaction->edit<nv::index::IEllipse>(m_raster_ellipses1_tag));
      check_success(ellipse.is_valid_interface());
      ellipse->set_enabled(false);

      mi::base::Handle<nv::index::IPath_3D> path1(
        dice_transaction->edit<nv::index::IPath_3D>(m_path3d_1_tag));
      path1->set_radius(20.0f); // from 10.0

      mi::base::Handle<nv::index::ILabel_2D> label_2d(
        dice_transaction->edit<nv::index::ILabel_2D>(m_label2d_tag));
      check_success(label_2d.is_valid_interface());
      label_2d->set_text("View 2: changed ellipse/path/polygon");

      mi::base::Handle<nv::index::ITransformed_scene_group> polygon_group_node(
     dice_transaction->edit<nv::index::ITransformed_scene_group>(m_polygon_group_tag));
      check_success(polygon_group_node.is_valid_interface());
      polygon_group_node->set_enabled(false);
    }
    dice_transaction->commit();
  }

  // Set up viewport 3:
  {
    const mi::Size view_idx = 3;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "3");
```

```
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      // no changes in this viewport
    }
    dice_transaction->commit();
  }

  // Set up viewport 0 with the global scope:
  {
    const mi::Size view_idx = 0;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "0");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      // no changes for this viewport
    }
    dice_transaction->commit();
  }
}

nv::index::IViewport_list* Multi_view_shape::filter_enabled_viewport_index_list()
{
  mi::base::Handle<nv::index::IViewport_list> new_viewport_list;
  {
   mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
    assert(dice_transaction.is_valid_interface());
    {
      assert(m_session_tag.is_valid());
      mi::base::Handle<const nv::index::ISession> session(
        dice_transaction->access<nv::index::ISession>(m_session_tag));
      assert(session.is_valid_interface());

      // create viewport from a session
      new_viewport_list = session->create_viewport_list();
    }
    dice_transaction->commit();
  }

  check_success(new_viewport_list.is_valid_interface());
  const mi::Size nb_views = m_viewport_list->size();

  std::stringstream sstr;
```

```
  mi::Size nb_enabled = 0;
  for (mi::Size i = 0; i < nb_views; ++i)
  {
    if (m_enable_view_idx_vec[i])
    {
      sstr << i << " ";
      mi::base::Handle<nv::index::IViewport> vp (m_viewport_list->get(i));
      new_viewport_list->append(vp.get());
      ++nb_enabled;
    }
  }
  // copy advisory state
  new_viewport_list->set_advisory_enabled(m_viewport_list->get_advisory_enabled());

  if (nb_enabled == nb_views)
  {
    INFO_LOG << "All views are enabled.";
  }
  else
  {
    INFO_LOG << "Filtered views. Enabled: " << sstr.str();
  }

  new_viewport_list->retain();
  return new_viewport_list.get();
}

bool Multi_view_shape::render_one_frame(mi::Sint32 frame_idx)
{
  bool success = true;

  // Render a frame and save the rendered image to a file.
  // Only save the file at the end of the iteration.
  std::string fname = "";
  if (!(m_outfname.empty()))
  {
    fname = get_output_file_name(m_outfname, frame_idx);
  }
  check_success(m_index_rendering.is_valid_interface());

  // set up canvas. output_fname.empty() is valid (no output file)
  m_image_file_canvas->set_rgba_file_name(fname.c_str());

  // set advisory to see the multiview rendering details
  m_viewport_list->set_advisory_enabled(true);

  mi::base::Handle<nv::index::IViewport_list> cur_viewport_list(
    filter_enabled_viewport_index_list());
  check_success(cur_viewport_list.is_valid_interface());

  mi::base::Handle<nv::index::IFrame_results_list> frame_results_list(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      cur_viewport_list.get()));
  check_success(frame_results_list.is_valid_interface());
```

```
  if (frame_results_list->size() == 0)
  {
    ERROR_LOG << "IIndex_rendering rendering call has no results.";
    success = false;
  }
  else
  {
    for (mi::Size i = 0; i < frame_results_list->size(); ++i)
    {
      mi::base::Handle<nv::index::IFrame_results>
        frame_results(frame_results_list->get(i));
      check_success(frame_results.is_valid_interface());

    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      check_success(err_set.is_valid_interface());
      if (err_set->any_errors())
      {
        std::ostringstream os;

     const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

      ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        success = false;
      }
    }
  }

  // verify the generated frame
  if (!(m_verify_image_path_base.empty()))
  {
   const std::string ref_img_fpath = get_output_file_name(m_verify_image_path_base, frame_idx);
    if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), ref_img_fpath, get_options())))
    {
      success = false;
    }
  }

  return success;
}

void Multi_view_shape::create_views() const
{
  // Check session and multiple views in the arc
  check_success(m_session_tag.is_valid());
  check_success(m_viewport_list.is_valid_interface());
  check_success(m_viewport_list->size() == 0);

  // Multiple view itself lives in the global scope.
```

```
mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
  m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
check_success(dice_transaction.is_valid_interface());
{
  mi::base::Handle<const nv::index::ISession> session(
    dice_transaction->access<nv::index::ISession>(m_session_tag));
  check_success(session.is_valid_interface());

  // 0. create a single viewport in the global scope.
  {
    mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
    check_success(viewport.is_valid_interface());

    const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0, 256);
    const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

    viewport->set_position(viewport_pos);
    viewport->set_size(viewport_size);
    viewport->set_scope(m_global_scope.get()); // ref count up

    m_viewport_list->append(viewport.get());
  }

  // 1. viewport
  {
    mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
    check_success(viewport.is_valid_interface());

    mi::neuraylib::IScope* parent = 0; // this means global scope in this context
    mi::Uint8 privacy_level = 1;
    bool      is_temp       = false;
  mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
    check_success(local_scope.is_valid_interface());

    const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256, 256);
    const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

    viewport->set_position(viewport_pos);
    viewport->set_size(viewport_size);
    viewport->set_scope(local_scope.get()); // ref count up

    m_viewport_list->append(viewport.get());
  }

  // 2. viewport
  {
    mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
    check_success(viewport.is_valid_interface());

    mi::neuraylib::IScope* parent = 0; // this means global scope in this context
    mi::Uint8 privacy_level = 1;
    bool      is_temp       = false;
  mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
    check_success(local_scope.is_valid_interface());

    const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0,   0);
    const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);
```

```
      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }

    // 3. viewport
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256,   0);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }
  }
  dice_transaction->commit();
}

void Multi_view_shape::create_light_material_instance(
  nv::index::IScene*                          scene_edit,
  mi::neuraylib::IDice_transaction*           dice_transaction,
  std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map) const
{
  assert(scene_edit != 0);
  assert(dice_transaction != 0);
  assert(light_mat_tag_map.empty()); // should be empty

  // create lights
  {
    mi::base::Handle<nv::index::IDirectional_headlight> default_light(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(default_light.is_valid_interface());

   default_light->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
    const mi::math::Color color_intensity(1.0f, 1.0f, 1.0f, 1.0f);
    default_light->set_intensity(color_intensity);

   const mi::neuraylib::Tag default_light_tag = dice_transaction->store_for_reference_counting(defa
    check_success(default_light_tag.is_valid());

    light_mat_tag_map["default_light"] = default_light_tag;
  }
```

```
  {
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());

    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
    const mi::math::Color color_intensity(1.0f, 1.0f, 1.0f, 1.0f);
    headlight->set_intensity(color_intensity);

   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());

    light_mat_tag_map["headlight"] = headlight_tag;
  }

  {
    mi::base::Handle<nv::index::IDirectional_light> dirlight(
      scene_edit->create_attribute<nv::index::IDirectional_light>());
    check_success(dirlight.is_valid_interface());

    dirlight->set_direction(mi::math::Vector<mi::Float32, 3>(-1.0f, 0.0f, 0.0f));
    const mi::math::Color color_intensity(1.0f, 1.0f, 1.0f, 1.0f);
    dirlight->set_intensity(color_intensity);

   const mi::neuraylib::Tag dirlight_tag = dice_transaction->store_for_reference_counting(dirlight.
    check_success(dirlight_tag.is_valid());

    light_mat_tag_map["dirlight"] = dirlight_tag;
  }

  // create materials
  {
    std::vector<std::string>     matname_vec;
    matname_vec.push_back("mat_red");
    matname_vec.push_back("mat_green");
    matname_vec.push_back("mat_blue");
    matname_vec.push_back("mat_yellow");
    matname_vec.push_back("mat_white");
    matname_vec.push_back("mat_pink");
    matname_vec.push_back("mat_exotic");

    std::vector<mi::math::Color> ambient_vec;
    ambient_vec.push_back(mi::math::Color(0.3f, 0.0f, 0.0f, 1.0f));
    ambient_vec.push_back(mi::math::Color(0.0f, 0.3f, 0.0f, 1.0f));
    ambient_vec.push_back(mi::math::Color(0.0f, 0.0f, 0.3f, 1.0f));
    ambient_vec.push_back(mi::math::Color(0.6f, 0.6f, 0.0f, 1.0f));
    ambient_vec.push_back(mi::math::Color(0.3f, 0.3f, 0.3f, 1.0f));
    ambient_vec.push_back(mi::math::Color(1.0f, 0.0f, 1.0f, 1.0f));
    ambient_vec.push_back(mi::math::Color(0.5f, 0.0f, 0.0f, 1.0f));

    std::vector<mi::math::Color> diffuse_vec;
    diffuse_vec.push_back(mi::math::Color(0.4f, 0.0f, 0.0f, 1.0f));
    diffuse_vec.push_back(mi::math::Color(0.0f, 0.4f, 0.0f, 1.0f));
    diffuse_vec.push_back(mi::math::Color(0.0f, 0.0f, 0.4f, 1.0f));
    diffuse_vec.push_back(mi::math::Color(0.3f, 0.3f, 0.0f, 1.0f));
    diffuse_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    diffuse_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.0f, 1.0f));
```

```
    diffuse_vec.push_back(mi::math::Color(0.0f, 0.5f, 0.0f, 1.0f));

    std::vector<mi::math::Color> specular_vec;
    specular_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    specular_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    specular_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    specular_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    specular_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    specular_vec.push_back(mi::math::Color(0.4f, 0.4f, 0.4f, 1.0f));
    specular_vec.push_back(mi::math::Color(0.0f, 0.0f, 1.0f, 1.0f));

    std::vector<mi::Float32>     shiness_vec;
    shiness_vec.push_back(100.0f);
    shiness_vec.push_back(100.0f);
    shiness_vec.push_back(100.0f);
    shiness_vec.push_back(100.0f);
    shiness_vec.push_back(100.0f);
    shiness_vec.push_back(100.0f);
    shiness_vec.push_back(100.0f);

    assert(matname_vec.size() == ambient_vec.size());
    assert(matname_vec.size() == diffuse_vec.size());
    assert(matname_vec.size() == specular_vec.size());
    assert(matname_vec.size() == shiness_vec.size());

    // Materials
    for (mi::Size i = 0; i < matname_vec.size(); ++i)
    {
    mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_
      check_success(phong_1.is_valid_interface());

      phong_1->set_ambient(ambient_vec[i]);
      phong_1->set_diffuse(diffuse_vec[i]);
      phong_1->set_specular(specular_vec[i]);
      phong_1->set_shininess(shiness_vec[i]);
      const mi::neuraylib::Tag phong_1_tag
        = dice_transaction->store_for_reference_counting(phong_1.get());
      check_success(phong_1_tag.is_valid());

      if (light_mat_tag_map.find(matname_vec[i]) != light_mat_tag_map.end())
      {
        ERROR_LOG << "Duplicated mat name: " << matname_vec[i];
        assert(false);
      }
      light_mat_tag_map[matname_vec[i]] = phong_1_tag;
    }
  }
}

mi::neuraylib::Tag Multi_view_shape::get_valid_light_mat_tag_by_name(
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  const std::string& light_mat_name) const
{
  if (light_mat_tag_map.find(light_mat_name) == light_mat_tag_map.end())
  {
    ERROR_LOG << "Cannot find light/mat name: " << light_mat_name;
    assert(false);
```

```
  }
  const mi::neuraylib::Tag tag = light_mat_tag_map.find(light_mat_name)->second;
  assert(tag.is_valid());

  return tag;
}

void Multi_view_shape::add_scene_defaut_light(
  nv::index::IScene*                                scene_edit,
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  mi::neuraylib::IDice_transaction*                 dice_transaction)
{
 const mi::neuraylib::Tag default_light_tag = get_valid_light_mat_tag_by_name(light_mat_tag_map, "d
  scene_edit->append(default_light_tag, dice_transaction);
}

void Multi_view_shape::add_scene_group_simple_geometry(
  nv::index::IScene*                                scene_edit,
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  mi::neuraylib::IDice_transaction*                 dice_transaction)
{
  // Add a scene group where the shapes should be added
  mi::base::Handle<nv::index::ITransformed_scene_group> simple_geometry_group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(simple_geometry_group_node.is_valid_interface());
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    200.0f, 200.0f, 600.0f, 1.0f
    );
  simple_geometry_group_node->set_transform(transform_mat);

  // headlight
 simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "headlight

  // mat_red
 simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_red")

  // group_spheres
  std::vector<mi::neuraylib::Tag> sphere_tag_vec;
  {
    // app::scene::group_spheres::type = transformed_scene_group
    mi::base::Handle<nv::index::ITransformed_scene_group> group_spheres(
      scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
    check_success(group_spheres.is_valid_interface());

    // Create spheres (3D)
    //
    std::vector<mi::math::Vector<mi::Float32, 3> > center_vec;
    center_vec.push_back(mi::math::Vector<mi::Float32, 3>(   0.0f,    0.0f, 0.0f));
    center_vec.push_back(mi::math::Vector<mi::Float32, 3>(-100.0f, -100.0f, 0.0f));
    center_vec.push_back(mi::math::Vector<mi::Float32, 3>( -60.0f, -100.0f, 0.0f));
    center_vec.push_back(mi::math::Vector<mi::Float32, 3>( -20.0f, -100.0f, 0.0f));
    center_vec.push_back(mi::math::Vector<mi::Float32, 3>(  20.0f, -100.0f, 0.0f));
    center_vec.push_back(mi::math::Vector<mi::Float32, 3>(  60.0f, -100.0f, 0.0f));
    center_vec.push_back(mi::math::Vector<mi::Float32, 3>( 100.0f,    0.0f, 0.0f));
```

```
    std::vector<mi::Float32> radius_vec;
    radius_vec.push_back(50.0f);
    radius_vec.push_back(10.0f);
    radius_vec.push_back(12.0f);
    radius_vec.push_back(14.0f);
    radius_vec.push_back(16.0f);
    radius_vec.push_back(18.0f);
    radius_vec.push_back(25.0f);
    assert(center_vec.size() == radius_vec.size());
    assert(center_vec.size() == 7);

    for (mi::Size i = 0; i < center_vec.size(); ++i)
    {
      mi::base::Handle<nv::index::ISphere> sphere(
        scene_edit->create_shape<nv::index::ISphere>());
      sphere->set_center(center_vec[i]);
      sphere->set_radius(radius_vec[i]);

      const mi::neuraylib::Tag sphere_tag
        = dice_transaction->store_for_reference_counting(sphere.get());
      check_success(sphere_tag.is_valid());
      sphere_tag_vec.push_back(sphere_tag);
    }

   // app::scene::group_spheres::children = mat_green sphere1 mat_red sphere2 sphere3 sphere4 sphere5
   simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_gree
    simple_geometry_group_node->append(sphere_tag_vec[0], dice_transaction);
   simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_red"
    simple_geometry_group_node->append(sphere_tag_vec[1], dice_transaction);
    simple_geometry_group_node->append(sphere_tag_vec[2], dice_transaction);
    simple_geometry_group_node->append(sphere_tag_vec[3], dice_transaction);
    simple_geometry_group_node->append(sphere_tag_vec[4], dice_transaction);
    simple_geometry_group_node->append(sphere_tag_vec[5], dice_transaction);
  }

  // mat_exotic
  simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_exotic
  // dirlight
  simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "dirlight"
  // sphere_exotic
  simple_geometry_group_node->append(sphere_tag_vec[6], dice_transaction);
  // mat_blue
  simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_blue"
  // headlight
  simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "headlight
  // mat_yellow
  simple_geometry_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_yellow
  // ellipsoid
  {
    mi::base::Handle<nv::index::IEllipsoid> ellipsoid(
      scene_edit->create_shape<nv::index::IEllipsoid>());
    ellipsoid->set_center(mi::math::Vector<mi::Float32, 3>(200.0f, 0.0f, 0.0f));

    mi::math::Vector<mi::Float32, 3> a(55.0f,  0.0f, 0.0f);
    mi::math::Vector<mi::Float32, 3> b( 0.0f, 35.0f, 0.0f);
    mi::Float32 c = 35.0f;
```

```
    ellipsoid->set_semi_axes(a, b, c);

    const mi::neuraylib::Tag ellipsoid_tag
      = dice_transaction->store_for_reference_counting(ellipsoid.get());
    check_success(ellipsoid_tag.is_valid());
    simple_geometry_group_node->append(ellipsoid_tag, dice_transaction);
  }

  // group_cylinders
  {
    // Cylinders and cones (3D)
    mi::base::Handle<nv::index::ITransformed_scene_group> group_cylinders(
      scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
    check_success(group_cylinders.is_valid_interface());
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f, 0.0f, 0.0f, 0.0f,
      0.0f, 1.0f, 0.0f, 0.0f,
      0.0f, 0.0f, 1.0f, 0.0f,
      100.0f, 200.0f, 150.0f, 1.0f
      );
    group_cylinders->set_transform(transform_mat);

    // mat_green
   group_cylinders->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_green"), dice_t
    mi::base::Handle<nv::index::ICylinder> capped_cylinder(
      scene_edit->create_shape<nv::index::ICylinder>());
   capped_cylinder->set_bottom(mi::math::Vector<mi::Float32, 3>(50.0f, 0.0f, 90.0f));
    capped_cylinder->set_top(mi::math::Vector<mi::Float32, 3>(220.0f, 0.0f, 40.0f));
    capped_cylinder->set_radius(15.0f);
    capped_cylinder->set_capped(true);
    const mi::neuraylib::Tag capped_cylinder_tag
      = dice_transaction->store_for_reference_counting(capped_cylinder.get());
    check_success(capped_cylinder_tag.is_valid());
    group_cylinders->append(capped_cylinder_tag, dice_transaction);

    // mat_blue
   group_cylinders->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_blue"), dice_tra
    // uncapped_cylinder
    mi::base::Handle<nv::index::ICylinder> uncapped_cylinder(
      scene_edit->create_shape<nv::index::ICylinder>());
   uncapped_cylinder->set_bottom(mi::math::Vector<mi::Float32, 3>(0.0f, 75.0f, 0.0f));
   uncapped_cylinder->set_top(mi::math::Vector<mi::Float32, 3>(250.0f, 75.0f, 0.0f));
    uncapped_cylinder->set_radius(30.0f);
    uncapped_cylinder->set_capped(false);
    const mi::neuraylib::Tag uncapped_cylinder_tag
      = dice_transaction->store_for_reference_counting(uncapped_cylinder.get());
    check_success(uncapped_cylinder_tag.is_valid());
    group_cylinders->append(uncapped_cylinder_tag, dice_transaction);

    // mat_red
   group_cylinders->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_red"), dice_tra
    // cone
    mi::base::Handle<nv::index::ICone> cone(
      scene_edit->create_shape<nv::index::ICone>());
    cone->set_center(mi::math::Vector<mi::Float32, 3>(250.0f, 75.0f, 0.0f));
    cone->set_tip(mi::math::Vector<mi::Float32, 3>(300.0f, 75.0f, 0.0f));
    cone->set_radius(30.0f);
```

```
    cone->set_capped(true);

    const mi::neuraylib::Tag cone_tag
      = dice_transaction->store_for_reference_counting(cone.get());
    check_success(cone_tag.is_valid());
    group_cylinders->append(cone_tag, dice_transaction);

    const mi::neuraylib::Tag group_cylinders_tag
      = dice_transaction->store_for_reference_counting(group_cylinders.get());
    check_success(group_cylinders_tag.is_valid());
    simple_geometry_group_node->append(group_cylinders_tag, dice_transaction);
  }

  m_simple_geo_group_node_tag =
   dice_transaction->store_for_reference_counting(simple_geometry_group_node.get());
  check_success(m_simple_geo_group_node_tag.is_valid());

  scene_edit->append(m_simple_geo_group_node_tag, dice_transaction);
}

void Multi_view_shape::add_scene_group_planes(
  nv::index::IScene*                                scene_edit,
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  mi::neuraylib::IDice_transaction*                 dice_transaction)
{
  // Textured planes
  mi::base::Handle<nv::index::ITransformed_scene_group> group_plane(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_plane.is_valid_interface());

 group_plane->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "headlight"), dice_transac
 group_plane->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_red"), dice_transacti
  // tex_checkerboard
  mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
   scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
  check_success(tex_filter.is_valid_interface());
 mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.get
  check_success(tex_filter_tag.is_valid());
  group_plane->append(tex_filter_tag, dice_transaction);

 // Access an application layer component that provides some sample techniques such as the checkerboar
 mi::base::Handle<nv::index::app::data_analysis_and_processing::IData_analysis_and_processing> pr
 get_application_layer_interface()->get_api_component<nv::index::app::data_analysis_and_processin
  check_success(processing.is_valid_interface());
 // Create a checkerboard technique and add it to the scene. For more details on how to implement the ch
 // source that was shipped with the application layer component 'nv::index::app::data_analysis_and_p
  mi::base::Handle<nv::index::IDistributed_compute_technique> tex(
    processing->get_sample_tool_set()->create_checker_board_2d_technique(
      mi::math::Vector<mi::Float32, 2>(320.0f, 320.0f),
    nv::index::IDistributed_compute_destination_buffer_2d_texture::FORMAT_RGBA_FLOAT32,
      mi::math::Color(0.9f, 0.2f, 0.15f, 0.7f),
      mi::math::Color(0.2f, 0.2f, 0.8f,  1.0f)));
  check_success(tex.is_valid_interface());

 const mi::neuraylib::Tag tex_tag = dice_transaction->store_for_reference_counting(tex.get());
  check_success(tex_tag.is_valid());
  group_plane->append(tex_tag, dice_transaction);
```

```
  // plane_diagonal
  mi::base::Handle<nv::index::IPlane> plane(
    scene_edit->create_shape<nv::index::IPlane>());
  plane->set_point( mi::math::Vector<mi::Float32,3>(200.0f, -100.0f, 300.0f));
  plane->set_normal(mi::math::Vector<mi::Float32,3>(1.0f, 1.0f, 1.0f));
  plane->set_up(    mi::math::Vector<mi::Float32,3>(-1.0f, 1.0f, 0.0f));
  plane->set_extent(mi::math::Vector<mi::Float32,2>(600.0f, 800.0f));
  check_success(!m_plane_tag.is_valid());
  m_plane_tag =
    dice_transaction->store_for_reference_counting(plane.get());
  check_success(m_plane_tag.is_valid());
  group_plane->append(m_plane_tag, dice_transaction);

  const mi::neuraylib::Tag group_plane_tag
    = dice_transaction->store_for_reference_counting(group_plane.get());
  check_success(group_plane_tag.is_valid());
  scene_edit->append(group_plane_tag, dice_transaction);
}

void Multi_view_shape::add_scene_group_raster(
  nv::index::IScene*                                scene_edit,
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  mi::neuraylib::IDice_transaction*                 dice_transaction)
{
  // Raster geometry (2D)
  mi::base::Handle<nv::index::ITransformed_scene_group> raster_group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(raster_group_node.is_valid_interface());
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    100.0f, 100.0f, 100.0f, 1.0f
    );
  raster_group_node->set_transform(transform_mat);

  // headlight
 raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "headlight"), dice_t

  // group_lines_points
  {
    // Lines and points (2D)
    mi::base::Handle<nv::index::ITransformed_scene_group> group_lines_points_node(
      scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
    check_success(group_lines_points_node.is_valid_interface());
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f, 0.0f, 0.0f, 0.0f,
      0.0f, 1.0f, 0.0f, 0.0f,
      0.0f, 0.0f, 1.0f, 0.0f,
      50.0f, 0.0f, 100.0f, 1.0f
      );
    group_lines_points_node->set_transform(transform_mat);

    // mat_white
   raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_white"), dice_
    // points
```

```
  {
    // Some shaded points with a headlight, with per-vertex color
    std::vector<mi::math::Vector<mi::Float32,3> > pos_vec;
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(30.0f, 50.0f, 10.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(34.0f, 51.0f, 40.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(39.0f, 58.0f, 70.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(50.0f, 62.0f, 100.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(53.0f, 70.0f, 140.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(55.0f, 50.0f, 190.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(70.0f, 51.0f, 240.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(90.0f, 58.0f, 370.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(91.0f, 62.0f, 500.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(93.0f, 70.0f, 640.0f));

    std::vector<mi::math::Color> col_vec;
    col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(0.0f, 1.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(0.0f, 0.0f, 1.0f, 1.0f));
    col_vec.push_back(mi::math::Color(1.0f, 1.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(0.0f, 1.0f, 1.0f, 1.0f));
    col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));

    std::vector<mi::Float32> rad_vec;
    for (mi::Size i = 0; i < pos_vec.size(); ++i)
    {
      rad_vec.push_back(2.0f);
    }

    assert(pos_vec.size() == col_vec.size());
    assert(pos_vec.size() == rad_vec.size());

    mi::base::Handle<nv::index::IPoint_set> point_set(
      scene_edit->create_shape<nv::index::IPoint_set>());
    check_success(point_set.is_valid_interface());
    point_set->set_point_style(nv::index::IPoint_set::SHADED_CIRCLE);
    point_set->set_vertices(&pos_vec[0], pos_vec.size());
    point_set->set_colors(  &col_vec[0], col_vec.size());
    point_set->set_radii(   &rad_vec[0], rad_vec.size());

    const mi::neuraylib::Tag point_set_tag
      = dice_transaction->store_for_reference_counting(point_set.get());
    check_success(point_set_tag.is_valid());
    raster_group_node->append(point_set_tag, dice_transaction);
  }

  // mat_blue
raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_blue"), dice_
  // lines
  {
    std::vector<mi::math::Vector<mi::Float32,3> > pos_vec;
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(30.0f, 50.0f, 10.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(34.0f, 51.0f, 40.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(34.0f, 51.0f, 40.0f));
```

```
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(39.0f, 58.0f, 70.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(39.0f, 58.0f, 70.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(50.0f, 62.0f, 100.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(50.0f, 62.0f, 100.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(53.0f, 70.0f, 140.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(53.0f, 70.0f, 140.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(55.0f, 50.0f, 190.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(55.0f, 50.0f, 190.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(70.0f, 51.0f, 240.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(70.0f, 51.0f, 240.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(90.0f, 58.0f, 370.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(90.0f, 58.0f, 370.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(91.0f, 62.0f, 500.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(91.0f, 62.0f, 500.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(93.0f, 70.0f, 640.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(40.0f, 60.0f, 15.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(44.0f, 61.0f, 45.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(44.0f, 61.0f, 45.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(59.0f, 68.0f, 75.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(59.0f, 68.0f, 75.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(60.0f, 70.0f, 109.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(60.0f, 70.0f, 109.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(60.0f, 75.0f, 141.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(60.0f, 75.0f, 141.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(60.0f, 80.0f, 196.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(60.0f, 80.0f, 196.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(80.0f, 81.0f, 242.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(80.0f, 81.0f, 242.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(100.0f, 88.0f, 393.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(100.0f, 88.0f, 393.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(101.0f, 82.0f, 530.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(101.0f, 82.0f, 530.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(108.0f, 90.0f, 655.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(108.0f, 90.0f, 655.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(113.0f, 109.0f, 685.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(113.0f, 109.0f, 685.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(129.0f, 115.0f, 688.0f));

    // Create line_set scene element and add it to the scene
  mi::base::Handle<nv::index::ILine_set> line_set(scene_edit->create_shape<nv::index::ILine_set>
    check_success(line_set.is_valid_interface());

    line_set->set_line_type(nv::index::ILine_set::LINE_TYPE_SEGMENTS);
    line_set->set_line_style(nv::index::ILine_set::LINE_STYLE_DASHED);
    line_set->set_lines(&pos_vec[0], pos_vec.size());

    std::vector<mi::math::Color> col_vec;
    std::vector<mi::Float32>     width_vec;
    for (mi::Size i = 0; i < pos_vec.size(); ++i)
    {
      col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
      width_vec.push_back(2.0f);
    }
    line_set->set_colors(&col_vec[0],   col_vec.size()/2);
    line_set->set_widths(&width_vec[0], width_vec.size());

    const mi::neuraylib::Tag line_set_tag
```

```
      = dice_transaction->store_for_reference_counting(line_set.get());

    INFO_LOG << "line_set_tag: " << line_set_tag;
    check_success(line_set_tag.is_valid());
    raster_group_node->append(line_set_tag, dice_transaction);
  }

  // dirlight
raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "dirlight"), dice_
  // mat_blue
raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_blue"), dice_
  // point_large
  {
    mi::math::Vector<mi::Float32,3> pos(120.0f, 120.0f, 120.0f);
    mi::Float32 rad = 20.0f;

    mi::base::Handle<nv::index::IPoint_set> point_large(
      scene_edit->create_shape<nv::index::IPoint_set>());
    check_success(point_large.is_valid_interface());
    point_large->set_point_style(nv::index::IPoint_set::SHADED_CIRCLE);
    point_large->set_vertices(&pos, 1);
    point_large->set_radii(   &rad, 1);

    const mi::neuraylib::Tag point_large_tag
      = dice_transaction->store_for_reference_counting(point_large.get());
    check_success(point_large_tag.is_valid());
    raster_group_node->append(point_large_tag, dice_transaction);
  }

  // mat_white
raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_white"), dice
  // triangles
  {
    std::vector<mi::math::Vector<mi::Float32,3> > pos_vec;
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(10.0f, 60.0f, 10.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(30.0f, 50.0f, 10.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(80.0f, 40.0f, 10.0f));

    std::vector<mi::math::Color> col_vec;
    col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(0.0f, 1.0f, 0.0f, 1.0f));
    col_vec.push_back(mi::math::Color(0.0f, 0.0f, 1.0f, 1.0f));

    std::vector<mi::Float32> rad_vec;
    for (mi::Size i = 0; i < pos_vec.size(); ++i)
    {
      rad_vec.push_back(10.0f);
    }

    mi::base::Handle<nv::index::IPoint_set> point_set(
      scene_edit->create_shape<nv::index::IPoint_set>());
    check_success(point_set.is_valid_interface());
    point_set->set_point_style(nv::index::IPoint_set::FLAT_TRIANGLE);
    point_set->set_vertices(&pos_vec[0], pos_vec.size());
    point_set->set_colors(  &col_vec[0], col_vec.size());
    point_set->set_radii(   &rad_vec[0], rad_vec.size());
```

```
    const mi::neuraylib::Tag point_set_tag
      = dice_transaction->store_for_reference_counting(point_set.get());
    check_success(point_set_tag.is_valid());
    raster_group_node->append(point_set_tag, dice_transaction);
  }
}

// group_circles_ellipses
{
  // Circles and ellipses (2D)
 mi::base::Handle<nv::index::ITransformed_scene_group> circle_ellipses_group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(circle_ellipses_group_node.is_valid_interface());
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    300.0f, 500.0f, 100.0f, 1.0f
    );
  circle_ellipses_group_node->set_transform(transform_mat);

  // headlight
 raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "headlight"), dice
  // mat_white
 raster_group_node->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_white"), dice
  {
    // circle and ellipses
    std::vector<mi::neuraylib::Tag> circle_and_ellipses_tag_vec;
    {
      {
        std::vector<mi::math::Vector<mi::Float32,3> > pos_vec;
        pos_vec.push_back(mi::math::Vector<mi::Float32,3>(300.0f, 50.0f, 90.0f));
       pos_vec.push_back(mi::math::Vector<mi::Float32,3>(100.0f, 450.0f, 350.0f));
        pos_vec.push_back(mi::math::Vector<mi::Float32,3>(440.0f, 450.0f, 56.0f));
        pos_vec.push_back(mi::math::Vector<mi::Float32,3>(440.0f, 450.0f, 56.0f));

        std::vector<mi::Float32> rad_vec;
        rad_vec.push_back(20.0f);
        rad_vec.push_back( 8.0f);
        rad_vec.push_back(10.0f);
        rad_vec.push_back( 5.0f);

        std::vector<mi::math::Color> col_vec;
        col_vec.push_back(mi::math::Color(1.0f, 1.0f, 0.0f, 1.0f));
        col_vec.push_back(mi::math::Color(1.0f, 0.0f, 1.0f, 1.0f));
        col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.5f, 1.0f));
        col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));

        std::vector<mi::Float32> width_vec;
        width_vec.push_back(0.75f);
        width_vec.push_back(2.75f);
        width_vec.push_back(5.75f);
        width_vec.push_back(1.0f);

        std::vector<mi::math::Color> fill_col_vec;
        fill_col_vec.push_back(mi::math::Color(0.0f, 1.0f, 1.0f, 0.66f));
        fill_col_vec.push_back(mi::math::Color(0.0f, 1.0f, 0.0f, 0.75f));
```

```
    fill_col_vec.push_back(mi::math::Color(1.0f, 0.5f, 0.0f, 1.0f));
    fill_col_vec.push_back(mi::math::Color(1.0f, 1.0f, 0.0f, 1.0f));

    assert(pos_vec.size() == rad_vec.size());
    assert(pos_vec.size() == col_vec.size());
    assert(pos_vec.size() == width_vec.size());
    assert(pos_vec.size() == fill_col_vec.size());

    for (mi::Size i = 0; i < pos_vec.size(); ++i)
    {
      mi::base::Handle<nv::index::ICircle> circle(
        scene_edit->create_shape<nv::index::ICircle>());
      check_success(circle.is_valid_interface());

      circle->set_geometry(pos_vec[i], rad_vec[i]);
      circle->set_outline_style(col_vec[i], width_vec[i]);
      circle->set_fill_style(fill_col_vec[i], nv::index::ICircle::FILL_SOLID);

      const mi::neuraylib::Tag circle_tag
        = dice_transaction->store_for_reference_counting(circle.get());
      check_success(circle_tag.is_valid());

      circle_and_ellipses_tag_vec.push_back(circle_tag);
    }
  }

  {
    std::vector<mi::math::Vector<mi::Float32,3> > pos_vec;
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(30.0f, 50.0f, -50.0f));
    pos_vec.push_back(mi::math::Vector<mi::Float32,3>(50.0f, 100.0f, 50.0f));
   pos_vec.push_back(mi::math::Vector<mi::Float32,3>(150.0f, 100.0f, 200.0f));

    std::vector<mi::Float32> rad_x_vec;
    rad_x_vec.push_back(15.0f);
    rad_x_vec.push_back(40.0f);
    rad_x_vec.push_back(40.0f);

    std::vector<mi::Float32> rad_y_vec;
    rad_y_vec.push_back(20.0f);
    rad_y_vec.push_back(20.0f);
    rad_y_vec.push_back(10.0f);

    std::vector<mi::Float32> rot_vec;
    rot_vec.push_back(0.2f);
    rot_vec.push_back(0.15f);
    rot_vec.push_back(0.785f);

    std::vector<mi::math::Color> line_col_vec;
    line_col_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
    line_col_vec.push_back(mi::math::Color(0.0f, 0.0f, 1.0f, 1.0f));
    line_col_vec.push_back(mi::math::Color(0.0f, 1.0f, 1.0f, 1.0f));

    std::vector<mi::Float32> width_vec;
    width_vec.push_back(2.0);
    width_vec.push_back(3.0);
    width_vec.push_back(15.0);
```

```
    std::vector<mi::math::Color> fill_col_vec;
    fill_col_vec.push_back(mi::math::Color(1.0f, 1.0f, 0.0f, 1.0f));
    fill_col_vec.push_back(mi::math::Color(1.0f, 1.0f, 1.0f, 0.5f));
    fill_col_vec.push_back(mi::math::Color(1.0f, 1.0f, 1.0f, 0.0f));

    assert(pos_vec.size() == rad_x_vec.size());
    assert(pos_vec.size() == rad_y_vec.size());
    assert(pos_vec.size() == rot_vec.size());
    assert(pos_vec.size() == line_col_vec.size());
    assert(pos_vec.size() == width_vec.size());
    assert(pos_vec.size() == fill_col_vec.size());

    for (mi::Size i = 0; i < pos_vec.size(); ++i)
    {
      mi::base::Handle<nv::index::IEllipse> ellipse(
        scene_edit->create_shape<nv::index::IEllipse>());
      check_success(ellipse.is_valid_interface());

    ellipse->set_geometry(pos_vec[i], rad_x_vec[i], rad_y_vec[i], rot_vec[i]);
      ellipse->set_outline_style(line_col_vec[i], width_vec[i]);
    ellipse->set_fill_style(fill_col_vec[i], nv::index::IEllipse::FILL_SOLID);

      const mi::neuraylib::Tag ellipse_tag
        = dice_transaction->store_for_reference_counting(ellipse.get());
      check_success(ellipse_tag.is_valid());

      circle_and_ellipses_tag_vec.push_back(ellipse_tag);
    }
  }

}

// Depth offset is needed so that the circles 3a and 3b actually overlap
mi::base::Handle<nv::index::IDepth_offset> depth_offset_attr(
  scene_edit->create_attribute<nv::index::IDepth_offset>());
check_success(depth_offset_attr.is_valid_interface());
depth_offset_attr->set_depth_offset(10.0f);
const mi::neuraylib::Tag depth_offset_attr_tag
  = dice_transaction->store_for_reference_counting(depth_offset_attr.get());
check_success(depth_offset_attr_tag.is_valid());

// circle1
raster_group_node->append(circle_and_ellipses_tag_vec[0], dice_transaction);
// circle2
raster_group_node->append(circle_and_ellipses_tag_vec[1], dice_transaction);
// circle3a
raster_group_node->append(circle_and_ellipses_tag_vec[2], dice_transaction);
// depth_offset_circle
raster_group_node->append(depth_offset_attr_tag, dice_transaction);
// circle3b
raster_group_node->append(circle_and_ellipses_tag_vec[3], dice_transaction);
// ellipse1
raster_group_node->append(circle_and_ellipses_tag_vec[4], dice_transaction);
// ellipse2
raster_group_node->append(circle_and_ellipses_tag_vec[5], dice_transaction);
// ellipse3
raster_group_node->append(circle_and_ellipses_tag_vec[6], dice_transaction);
```

```
    m_raster_circle1_tag   = circle_and_ellipses_tag_vec[0];
    m_raster_ellipses1_tag = circle_and_ellipses_tag_vec[4];
  }
}

// group_polygons
{
  // Polygons (2D)
  mi::base::Handle<nv::index::ITransformed_scene_group> group_polygons(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_polygons.is_valid_interface());
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f, 0.0f, 0.0f, 0.0f, // Half size
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f
    );
  group_polygons->set_transform(transform_mat);

  // headlight
 group_polygons->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "headlight"), dice_tra
  // mat_white
 group_polygons->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_white"), dice_tra

  const mi::Size NB_POLY = 4;
  std::vector<mi::math::Vector<mi::Float32, 3> > center_vec;
  center_vec.push_back(mi::math::Vector<mi::Float32, 3>(90.0f, 50.0f, 150.0f));
  center_vec.push_back(mi::math::Vector<mi::Float32, 3>(129.0f, 300.0f, 50.0f));
  center_vec.push_back(mi::math::Vector<mi::Float32, 3>(200.0f, 150.0f, 200.0f));
  center_vec.push_back(mi::math::Vector<mi::Float32, 3>(530.0f, 50.0f, 340.0f));

  std::vector<mi::math::Vector<mi::Float32, 2> > vtx_vec[NB_POLY];
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, 20.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, 60.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, 60.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, 20.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(-60.0f, 20.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(-60.0f, -20.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, -20.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, -60.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, -60.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, -20.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(60.0f, -20.0f));
  vtx_vec[0].push_back(mi::math::Vector<mi::Float32, 2>(60.0f, 20.0f));

  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(0.0f, 60.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, 20.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(-60.0f, 20.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(-25.0f, 0.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(-40.0f, -40.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(0.0f, -20.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(40.0f, -40.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(25.0f, 0.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(60.0f, 20.0f));
  vtx_vec[1].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, 20.0f));
```

```
vtx_vec[2].push_back(mi::math::Vector<mi::Float32, 2>(0.0f, 40.0f));
vtx_vec[2].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, 0.0f));
vtx_vec[2].push_back(mi::math::Vector<mi::Float32, 2>(0.0f, -40.0f));
vtx_vec[2].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, 0.0f));

vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(0.0f, 20.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, 60.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(-60.0f, 60.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(-30.0f, 0.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(-60.0f, -60.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(-20.0f, -60.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(0.0f, -20.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, -60.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(60.0f, -60.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(30.0f, 0.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(60.0f, 60.0f));
vtx_vec[3].push_back(mi::math::Vector<mi::Float32, 2>(20.0f, 60.0f));

// Adjust for the canvas (Originally for 1024x1024, but this is 256x256)
for (mi::Size i = 0; i < NB_POLY; ++i)
{
  for (mi::Size j = 0; j < vtx_vec[i].size(); ++j)
  {
    vtx_vec[i][j] = 0.25f * vtx_vec[i][j];
  }
}

std::vector<mi::math::Color> line_color_vec;
line_color_vec.push_back(mi::math::Color(1.0f, 1.0f, 0.5f, 1.0f));
line_color_vec.push_back(mi::math::Color(1.0f, 0.5f, 0.5f, 1.0f));
line_color_vec.push_back(mi::math::Color(1.0f, 1.0f, 1.0f, 1.0f));
line_color_vec.push_back(mi::math::Color(0.7f, 0.7f, 1.0f, 1.0f));

std::vector<mi::Float32> line_width_vec;
line_width_vec.push_back(3.0f);
line_width_vec.push_back(2.5f);
line_width_vec.push_back(3.5f);
line_width_vec.push_back(1.7f);

std::vector<mi::math::Color> fill_color_vec;
fill_color_vec.push_back(mi::math::Color(0.0f, 1.0f, 0.0f, 1.0f));
fill_color_vec.push_back(mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f));
fill_color_vec.push_back(mi::math::Color(0.0f, 0.0f, 1.0f, 1.0f));
fill_color_vec.push_back(mi::math::Color(1.0f, 1.0f, 0.5f, 1.0f));

check_success(center_vec.size() == line_color_vec.size());
check_success(center_vec.size() == line_width_vec.size());
check_success(center_vec.size() == fill_color_vec.size());

for (mi::Size i = 0; i < center_vec.size(); ++i)
{
  mi::base::Handle<nv::index::IPolygon> polygon(
    scene_edit->create_shape<nv::index::IPolygon>());
check_success((polygon->set_geometry(&(vtx_vec[i][0]), vtx_vec[i].size(), center_vec[i])));
  polygon->set_fill_style(fill_color_vec[i], nv::index::IPolygon::FILL_SOLID);

const mi::neuraylib::Tag polygon_tag = dice_transaction->store_for_reference_counting(polygon.g
```

```
      check_success(polygon_tag.is_valid());
      group_polygons->append(polygon_tag, dice_transaction);
    }

    check_success(!m_polygon_group_tag.is_valid());
   m_polygon_group_tag = dice_transaction->store_for_reference_counting(group_polygons.get());
    check_success(m_polygon_group_tag.is_valid());
    raster_group_node->append(m_polygon_group_tag, dice_transaction);
  }

  const mi::neuraylib::Tag raster_group_node_tag
    = dice_transaction->store_for_reference_counting(raster_group_node.get());
  check_success(raster_group_node_tag.is_valid());
  scene_edit->append(raster_group_node_tag, dice_transaction);
}

void Multi_view_shape::add_scene_group_paths(
  nv::index::IScene*                                scene_edit,
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  mi::neuraylib::IDice_transaction*                 dice_transaction)
{
  // Paths (2D/3D)
  mi::base::Handle<nv::index::ITransformed_scene_group> group_paths(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_paths.is_valid_interface());
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    160.0f, 150.0f, 300.0f, 1.0f
    );
  group_paths->set_transform(transform_mat);

  // headlight
 group_paths->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "headlight"), dice_transac
  // mat_blue
 group_paths->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_blue"), dice_transact
  // mat_green
 group_paths->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_green"), dice_transac
  // mat_red
 group_paths->append(get_valid_light_mat_tag_by_name(light_mat_tag_map, "mat_red"), dice_transacti

  mi::base::Handle<nv::index::IPath_style> path_style_linear(
    scene_edit->create_attribute<nv::index::IPath_style>());
  check_success(path_style_linear.is_valid_interface());
 path_style_linear->set_interpolation(nv::index::IPath_style::INTERPOLATION_LINEAR);

 const mi::neuraylib::Tag path_style_linear_tag = dice_transaction->store_for_reference_counting(pa
  check_success(path_style_linear_tag.is_valid());
  group_paths->append(path_style_linear_tag, dice_transaction);

  // path1
  std::vector<mi::math::Vector<mi::Float32,3> > point_vec;
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(130.0f, 250.0f, 10.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(134.0f, 251.0f, 40.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(139.0f, 258.0f, 70.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(150.0f, 262.0f, 100.0f));
```

```
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(153.0f, 270.0f, 140.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(155.0f, 250.0f, 190.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(170.0f, 251.0f, 240.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(190.0f, 258.0f, 370.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(191.0f, 262.0f, 500.0f));
  point_vec.push_back(mi::math::Vector<mi::Float32,3>(193.0f, 270.0f, 640.0f));

  std::vector<mi::Uint32> mat_vec;
  mat_vec.push_back(2);
  mat_vec.push_back(1);
  mat_vec.push_back(0);
  mat_vec.push_back(2);
  mat_vec.push_back(2);
  mat_vec.push_back(1);
  mat_vec.push_back(1);
  mat_vec.push_back(0);
  mat_vec.push_back(2);
  mat_vec.push_back(1);

  mi::base::Handle<nv::index::IPath_3D> path1(
    scene_edit->create_shape<nv::index::IPath_3D>());
  check_success(path1.is_valid_interface());
  path1->set_radius(10.0f);
  path1->set_points(&(point_vec[0]), point_vec.size());
  path1->set_color_map_indexes(&(mat_vec[0]), mat_vec.size());

  check_success(!m_path3d_1_tag.is_valid());
  m_path3d_1_tag = dice_transaction->store_for_reference_counting(path1.get());
  check_success(m_path3d_1_tag.is_valid());
  group_paths->append(m_path3d_1_tag, dice_transaction);

 const mi::neuraylib::Tag group_paths_tag = dice_transaction->store_for_reference_counting(group_pa
  check_success(group_paths_tag.is_valid());
  scene_edit->append(group_paths_tag, dice_transaction);
}

void Multi_view_shape::add_scene_group_labels(
  nv::index::IScene*                                scene_edit,
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  mi::neuraylib::IDice_transaction*                 dice_transaction)
{
  // Labels (2D/3D)
  mi::base::Handle<nv::index::ITransformed_scene_group> group_labels(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_labels.is_valid_interface());
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f
    );
  group_labels->set_transform(transform_mat);

  // default_font
 mi::base::Handle<nv::index::IFont> font(scene_edit->create_attribute<nv::index::IFont>());
  check_success(font.is_valid_interface());
```

```
 if (font->set_file_name(m_font_fpath.c_str()))
 {
   INFO_LOG << "set the font path [" << m_font_fpath << "]";
 }
 else
 {
   ERROR_LOG << "Can not find the font path [" << m_font_fpath << "], "
        << "the rendering result may not correct.";
 }
 font->set_font_resolution(64.0f);
const mi::neuraylib::Tag font_tag = dice_transaction->store_for_reference_counting(font.get());
 check_success(font_tag.is_valid());
 group_labels->append(font_tag, dice_transaction);

 // label_layout1
 mi::base::Handle<nv::index::ILabel_layout> label_layout(
   scene_edit->create_attribute<nv::index::ILabel_layout>());
 check_success(label_layout.is_valid_interface());
 label_layout->set_padding(10.0f);
label_layout->set_color(mi::math::Color(1.0f, 1.0f, 1.0f, 1.0f), mi::math::Color(1.0f, 1.0f, 1.0f,
 const mi::neuraylib::Tag label_layout_tag =
   dice_transaction->store_for_reference_counting(label_layout.get());
 check_success(label_layout_tag.is_valid());
 group_labels->append(label_layout_tag, dice_transaction);

 // label_2d
mi::base::Handle<nv::index::ILabel_2D> label_2d(scene_edit->create_shape<nv::index::ILabel_2D>()
 check_success(label_2d.is_valid_interface());
 label_2d->set_text("Global scope view");
 const mi::math::Vector<mi::Float32, 3> label_2d_position(-40.0f, 000.0f, 570.0f);
 const mi::math::Vector<mi::Float32, 2> right_2d(1.0f, 0.0f);
 const mi::math::Vector<mi::Float32, 2> up_2d   (0.0f, 1.0f);
 label_2d->set_geometry(label_2d_position, right_2d, up_2d, 30.0f, -1.0f);
 m_label2d_tag =
   dice_transaction->store_for_reference_counting(label_2d.get());
 check_success(m_label2d_tag.is_valid());

 group_labels->append(m_label2d_tag, dice_transaction);

 // label_layout2
 mi::base::Handle<nv::index::ILabel_layout> label_layout2(
   scene_edit->create_attribute<nv::index::ILabel_layout>());
 check_success(label_layout2.is_valid_interface());
 label_layout2->set_padding(0.0f);
label_layout2->set_color(mi::math::Color(0.46f, 0.73f, 0.0f, 1.0f), mi::math::Color(0.0f, 0.0f, 0.
 const mi::neuraylib::Tag label_layout2_tag =
   dice_transaction->store_for_reference_counting(label_layout2.get());
 check_success(label_layout2_tag.is_valid());
 group_labels->append(label_layout2_tag, dice_transaction);

 // label_3d
mi::base::Handle<nv::index::ILabel_3D> label_3d(scene_edit->create_shape<nv::index::ILabel_3D>()
 check_success(label_3d.is_valid_interface());
 label_3d->set_text("NVIDIA IndeX");
 const mi::math::Vector<mi::Float32, 3> label_3d_position(550.0f, 200.0f, 150.0f);
 const mi::math::Vector<mi::Float32, 3> right_3d(1.0f, 0.0f, 0.0f);
 const mi::math::Vector<mi::Float32, 3> up_3d   (0.0f, 0.0f, -1.0f);
```

```
  label_3d->set_geometry(label_3d_position, right_3d, up_3d, 50.0f, 400.0f);
 const mi::neuraylib::Tag label_3d_tag = dice_transaction->store_for_reference_counting(label_3d.g
  check_success(label_3d_tag.is_valid());
  group_labels->append(label_3d_tag, dice_transaction);

  const mi::neuraylib::Tag group_labels_tag =
    dice_transaction->store_for_reference_counting(group_labels.get());
  check_success(group_labels_tag.is_valid());
  scene_edit->append(group_labels_tag, dice_transaction);
}

void Multi_view_shape::add_scene_group_icons(
  nv::index::IScene*                                scene_edit,
  const std::map<std::string, mi::neuraylib::Tag>& light_mat_tag_map,
  mi::neuraylib::IDice_transaction*                 dice_transaction)
{
  // Icons (2D/3D)
  mi::base::Handle<nv::index::ITransformed_scene_group> group_icons(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_icons.is_valid_interface());
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f
    );
  group_icons->set_transform(transform_mat);

  // texture_nvidia
  mi::base::Handle<nv::index::ITexture> tex(
    scene_edit->create_attribute<nv::index::ITexture>());

  // Load the texture from a file
  mi::base::Handle<nv::index::app::image::IImage_importer> image_importer(
   get_application_layer_interface()->get_api_component<nv::index::app::image::IImage_importer>())
  check_success(image_importer.is_valid_interface());
  mi::base::Handle<mi::neuraylib::ICanvas> canvas(
    image_importer->create_canvas_from_file(m_iconfile.c_str()));
  check_success(canvas.is_valid_interface());

  // Copy the pixel data
 check_success(tex->set_pixel_data(canvas.get(), nv::index::ITexture::RGBA_FLOAT32));

  const mi::neuraylib::Tag tex_tag =
    dice_transaction->store_for_reference_counting(tex.get());
  check_success(tex_tag.is_valid());
  group_icons->append(tex_tag, dice_transaction);

  // icon1
  {
    mi::base::Handle<nv::index::IIcon_3D> icon_3d(
      scene_edit->create_shape<nv::index::IIcon_3D>());
    check_success(icon_3d.is_valid_interface());
    const mi::math::Vector<mi::Float32, 3> position(-150.0f, 700.0f, 400.0f);
    const mi::math::Vector<mi::Float32, 3> right_vector(0.0f, -1.0f,  0.0f);
    const mi::math::Vector<mi::Float32, 3> up_vector   (0.0f,  0.0f, -1.0f);
    icon_3d->set_geometry(position, right_vector, up_vector, 44.0f, 180.0f);
```

```
    const mi::neuraylib::Tag icon_3d_tag =
      dice_transaction->store_for_reference_counting(icon_3d.get());
    check_success(icon_3d_tag.is_valid());
    group_icons->append(icon_3d_tag, dice_transaction);
  }

  // icon2
  {
    mi::base::Handle<nv::index::IIcon_2D> icon_2d(
      scene_edit->create_shape<nv::index::IIcon_2D>());
    check_success(icon_2d.is_valid_interface());
    const mi::math::Vector<mi::Float32, 3> position(520.0f, 0.0f, 80.0f);
    const mi::math::Vector<mi::Float32, 2> right_vector(1.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 2> up_vector   (0.0f, 1.0f);
    icon_2d->set_geometry(position, right_vector, up_vector, 44.0f, 180.0f);
    const mi::neuraylib::Tag icon_2d_tag =
      dice_transaction->store_for_reference_counting(icon_2d.get());
    check_success(icon_2d_tag.is_valid());
    group_icons->append(icon_2d_tag, dice_transaction);
  }

  const mi::neuraylib::Tag group_icons_tag =
    dice_transaction->store_for_reference_counting(group_icons.get());
  check_success(group_icons_tag.is_valid());
  scene_edit->append(group_icons_tag, dice_transaction);
}

void Multi_view_shape::setup_scene(mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(dice_transaction != 0);

  // Access the session instance from the database.
  mi::base::Handle<const nv::index::ISession> session(
    dice_transaction->access<const nv::index::ISession>(m_session_tag));
  check_success(session.is_valid_interface());

  // Access (edit mode) the scene instance from the database.
  mi::base::Handle<nv::index::IScene> scene_edit(
    dice_transaction->edit<nv::index::IScene>(session->get_scene()));
  check_success(scene_edit.is_valid_interface());

  std::map<std::string, mi::neuraylib::Tag> light_mat_tag_map;
 create_light_material_instance(scene_edit.get(), dice_transaction, light_mat_tag_map);

  // Add a scene elements to the scene
  add_scene_defaut_light(scene_edit.get(), light_mat_tag_map, dice_transaction);
 add_scene_group_simple_geometry(scene_edit.get(), light_mat_tag_map, dice_transaction);
  add_scene_group_planes(scene_edit.get(), light_mat_tag_map, dice_transaction);
  add_scene_group_raster(scene_edit.get(), light_mat_tag_map, dice_transaction);
  add_scene_group_paths( scene_edit.get(), light_mat_tag_map, dice_transaction);
  add_scene_group_labels(scene_edit.get(), light_mat_tag_map, dice_transaction);
  add_scene_group_icons( scene_edit.get(), light_mat_tag_map, dice_transaction);

  // Define a region of interest for the entire scene in the
  // scene's global coordinate system.
  const mi::math::Bbox<mi::Float32, 3> region_of_interest(
    -1000.0f, -1000.0f, -1000.0f,
```

```
    1300.0f,  1300.0f,  1300.0f
    );
  scene_edit->set_clipped_bounding_box(region_of_interest);

  // Set the scene global transformation matrix.
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f,  0.0f,  0.0f,  0.0f,
    0.0f,  1.0f,  0.0f,  0.0f,
    0.0f,  0.0f,  1.0f,  0.0f,
    0.0f,  0.0f,  0.0f,  1.0f
    );

  scene_edit->set_transform_matrix(transform_mat);

  // Create a camera
  mi::base::Handle< nv::index::IPerspective_camera > cam(
    scene_edit->create_camera<nv::index::IPerspective_camera>());
  check_success(cam.is_valid_interface());
  m_camera_tag = dice_transaction->store(cam.get());
  check_success(m_camera_tag.is_valid());

  // set up the camera (default camera parameter)
  setup_camera(m_camera_tag, dice_transaction);

  // ... and add the camera to the scene.
  scene_edit->set_camera(m_camera_tag);
}

bool Multi_view_shape::setup_main_host()
{
 // Access the IndeX rendering query interface for querying performance values and pick results
  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  // DiCE database access
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    //------------------------------------------------------------------------
    // Setup session information
    m_session_tag =
      m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<nv::index::ISession>(
        m_session_tag));
    check_success(session.is_valid_interface());
```

```
    // Setup the main multiple views, but it is empty at here.
    m_viewport_list = session->create_viewport_list();
    check_success(m_viewport_list.is_valid_interface());

    //------------------------------------------------------------------------
    // Scene setup in the global scope
    setup_scene(dice_transaction.get());
  }
  dice_transaction->commit();

  // set canvas size
  const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
  m_image_file_canvas->set_resolution(buffer_resolution);

  INFO_LOG << "Initialization complete.";

  return true;
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "4");                                    // log level
  sdict.insert("font_fpath", "/usr/share/fonts/dejavu/DejaVuSans.ttf"); // font path
  sdict.insert("iconfile", "../create_icons/nvidia_logo.ppm");
 sdict.insert("outfname", "frame_multi_view_shape");             // output file base name
  sdict.insert("verify_image_path_base", "");                        // for unit test
  sdict.insert("unittest", "0");                                    // unit test mode
 sdict.insert("enable_vidx_vec", "1 1 1 1");                  // enabled view indices vector

  // Load IndeX library via Index_connect
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io data_analysis_and_processing" );

  // Initialize application
  Multi_view_shape multi_view_shape;
  multi_view_shape.initialize(argc, argv, sdict);
  check_success(multi_view_shape.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = multi_view_shape.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.25 multi_view_trimesh.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/ifont.h>
#include <nv/index/ilabel.h>

#include <nv/index/iindex.h>
#include <nv/index/iindex_debug_configuration.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/itriangle_mesh_query_results.h>
#include <nv/index/itriangle_mesh_scene_element.h>
#include <nv/index/iviewport.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/app_rendering_context.h"
#include "utility/canvas_utility.h"

#include <sstream>
#include <iostream>

class Multi_view_trimesh:
    public nv::index::app::Index_connect
{
public:
  Multi_view_trimesh()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Multi_view_trimesh() ctor";
  }

  virtual ~Multi_view_trimesh()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Multi_view_trimesh() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
```

```
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Setup camera to see this example scene
  // \param[in] camera_tag camera tag
  // \param[in] dice_transaction dice transaction
  void setup_camera(
    const mi::neuraylib::Tag&         camera_tag,
    mi::neuraylib::IDice_transaction* dice_transaction) const;
  // Localize scene elements
  void localize_scene_element() const;
  // Filter enable viewport index list
  // \return newly created filtered viewport list
  nv::index::IViewport_list* filter_enabled_viewport_index_list();
  // Render one frame
  // \param[in] frame_idx        current frame index
  // \return true when success
  bool render_one_frame(mi::Sint32 frame_idx);
  // Create multiple viewports
  //
  // The viewport structure of this example
  //
  //        +-----------------+-----------------+(511,511)
  //        | viewport 0      | viewport 1      |
  // (0,256)| global scope    | camera change   |(511,256)
  //        +-----------------+-----------------+
  // (0,255)| viewport 2      | viewport 3      |(511,255)
  //        | camera change   | camera/color/   |
  //        |                 | opacity change  |
  //        +-----------------+-----------------+
  //       (0,0)       (255,0) (256,0)           (511,0)
  //
  //
 // \param[in,out] arc  application rendering context. multiple views will be updated.
  void create_views() const;
  // Add a label
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node   parent group node
  // \param[in] label_point  label's corner position in the object space
  // \param[in] label_str    label contents string
```

```
  // \param[in] label_height label height in the object space
  // \param[in] label_width  label width in the object space
  // \param[in] fg_col       label foreground color
  // \param[in] bg_col       label background color
  // \param[in] dice_transaction db transaction
  // \return created label tag
  mi::neuraylib::Tag create_append_label_to_group(
    nv::index::IScene*                      scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
    const mi::math::Vector<mi::Float32, 3>& label_point,
    const std::string&                      label_str,
    const mi::Float32                       label_height,
    const mi::Float32                       label_width,
    const mi::math::Color_struct&           fg_col,
    const mi::math::Color_struct&           bg_col,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // Set up the scene
  // Create a scene that contains a triangle mesh.
  // \param[in] dice_transaction db transaction
  void setup_scene(mi::neuraylib::IDice_transaction* dice_transaction);
  // Set up as the main host
  // \return true when success
  bool setup_main_host();

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                 m_verify_image_path_base;
  std::vector<bool>                                             m_enable_view_idx_vec;
  std::string                                                            m_font_fpath;
  std::string                                                              m_iconfile;
  std::string                                                             m_mesh_file;
  mi::Float32                                                  m_mesh_mat_opacity_0;
  mi::Float32                                                  m_mesh_mat_opacity_1;
  std::string                                                              m_mesh_bbox;
  mi::math::Bbox<mi::Float32, 3>                                                 m_roi;
  // camera tag
  mi::neuraylib::Tag                                                     m_camera_tag;
  // trimesh scene element tag
  mi::neuraylib::Tag                                                    m_trimesh_tag;
  // triangle mesh material tag (phong)
  mi::neuraylib::Tag                                               m_mesh_phong_1_tag;
  // static group node tag for trimesh
  mi::neuraylib::Tag                                            m_static_group_node_tag;
  // label group node tag
  mi::neuraylib::Tag                                            m_label_group_node_tag;
  // label tag vector
  std::vector<mi::neuraylib::Tag>                                     m_label_tag_vec;
  mi::base::Handle<nv::index::IViewport_list>                            m_viewport_list;
};
```

```
mi::Sint32 Multi_view_trimesh::launch()
{
  // setup
  {
    check_success(setup_main_host());

    // Create multiple views in the arc.
    create_views();

    mi::Sint32 frame_idx = 0;
    // Render multiple views for a single render call. Before localize.
    {
      // Render a frame and save the rendered image to a file.
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }

    // localize scene elements
    localize_scene_element();

    // Render multiple views for a single render call.
    {
      // Render a frame and save the rendered image to a file.
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }
  }
  return 0;
}

bool Multi_view_trimesh::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));
  if (m_is_unittest)
  {
    sdict.insert("dice::verbose", "2");
  }

  m_outfname               = sdict.get("outfname");
  m_verify_image_path_base = sdict.get("verify_image_path_base");
  m_enable_view_idx_vec    = get_bool_vec_from_string(sdict.get("enable_vidx_vec"));
  m_font_fpath             = sdict.get("font_fpath");
  m_iconfile               = sdict.get("iconfile");
  m_mesh_file              = sdict.get("mesh_file");
 m_mesh_mat_opacity_0    = nv::index::app::get_float32(sdict.get("mesh_mat_opacity_0"));
 m_mesh_mat_opacity_1    = nv::index::app::get_float32(sdict.get("mesh_mat_opacity_1"));
  m_mesh_bbox              = sdict.get("mesh_bbox");
 m_roi             = nv::index::app::get_bbox_from_string<mi::Float32,3>(sdict.get("roi"));

  info_cout("running " + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_path_base = [" + m_verify_image_path_base +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);
```

```
  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            print out this message\n"

      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-font_fpath FONT_FILE_PATH]\n"
      << "            font file path. (default: " << m_font_fpath << ")\n"

      << "        [-mesh_file MESH_FILE_PATH]\n"
    << "         name of the triangle mesh file (in .bin-format). When empty, the default triangle mesh
      << "            (default: [" << m_mesh_file << "])\n"

      << "        [-mesh_mat_opacity_0 float]\n"
      << "            mesh material opacity 0."
      << "(default: " << m_mesh_mat_opacity_0 << ")\n"

      << "        [-mesh_mat_opacity_1 float]\n"
      << "            mesh material opacity 1."
      << "(default: " << m_mesh_mat_opacity_1 << ")\n"

      << "        [-mesh_bbox \"float float float float float float\"]\n"
      << "            mesh bounding box."
      << "(default: " << m_mesh_bbox << ")\n"

      << "        [-roi \"float float float float float float\"]\n"
      << "            the bounding box representing the region of interest (roi).\n"
      << "            (default: [" << m_roi << "])\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_path_base image path basename]\n"
     << "          when image_fname path basename exist, verify the rendering images.\n"
      << "            (default: [" << m_verify_image_path_base << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode. "
      << "(default: [" << m_is_unittest << "])\n"

      << "        [-enable_vidx_vec enabled_vector]\n"
      << "            Specify which viewport is enabled by a bool vector.\n"
      << "            (default: [" << sdict.get("enable_vidx_vec") << "])"

      << std::endl;
    exit(1);
  }
  return true;
}
```

```
void Multi_view_trimesh::setup_camera(
  const mi::neuraylib::Tag&         camera_tag,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(camera_tag.is_valid());

  mi::base::Handle<nv::index::IPerspective_camera> cam(
    dice_transaction->edit<nv::index::IPerspective_camera>(camera_tag));
  check_success(cam.is_valid_interface());

  // Set the camera parameters to see the whole scene
  const mi::math::Vector<mi::Float32, 3> from( 450.0f,   800.0f,  1050.0f);
  mi::math::Vector<mi::Float32, 3>       dir (  0.0f,     0.0f,     -1.0f);
  const mi::math::Vector<mi::Float32, 3> up  (  0.0f,     1.0f,      0.0f);
  dir.normalize();

  cam->set(from, dir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(1000.0f);
}

void Multi_view_trimesh::localize_scene_element() const
{
  check_success(m_viewport_list.is_valid_interface());

  // Process local viewports first.

  // Set up viewport 1: enable the label 1 and change the camera
  {
    const mi::Size view_idx = 1;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "1");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      // localize labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        mi::Sint32 ret_localize = 1;
        ret_localize = dice_transaction->localize(m_label_tag_vec[i],
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }
```

```
    // localize camera
    {
      mi::Sint32 ret_localize = 1;
      ret_localize = dice_transaction->localize(m_camera_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
    }

    // edit labels
    for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
    {
      mi::base::Handle<nv::index::ILabel_3D> label(
        dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
      check_success(label.is_valid_interface());
      if (i == view_idx)
      {
        label->set_enabled(true);
        label->set_text("View 1: Left camera");
      }
      else
      {
        label->set_enabled(false);
      }
    }

    // edit camera
    {
      mi::base::Handle<nv::index::ICamera> cam(
        dice_transaction->edit<nv::index::ICamera>(m_camera_tag));
      check_success(cam.is_valid_interface());

      const mi::math::Vector<mi::Float32, 3> from( 100.0f,   800.0f,  1050.0f);
      mi::math::Vector<mi::Float32, 3>       dir (  0.7f,     0.0f,    -1.0f);
      const mi::math::Vector<mi::Float32, 3> up  (  0.0f,     1.0f,     0.0f);
      dir.normalize();

      cam->set(from, dir, up);
    }
  }
  dice_transaction->commit();
}

// Set up viewport 2: enable the label 2 and change the camera
{
  const mi::Size view_idx = 2;
  check_success(m_viewport_list->size() > view_idx);

  mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
  check_success(viewport != 0);

  mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
  check_success(cur_scope);
  check_success(std::string(cur_scope->get_id()) == "2");

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
```

```
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      // localize labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        mi::Sint32 ret_localize = 1;
        ret_localize = dice_transaction->localize(m_label_tag_vec[i],
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }

      // localize camera
      {
        mi::Sint32 ret_localize = 1;
        ret_localize = dice_transaction->localize(m_camera_tag,
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }

      // edit labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        mi::base::Handle<nv::index::ILabel_3D> label(
          dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
        check_success(label.is_valid_interface());
        if (i == view_idx)
        {
          label->set_enabled(true);
          label->set_text("View 2: right camera");
        }
        else
        {
          label->set_enabled(false);
        }
      }

      // edit camera
      {
        mi::base::Handle<nv::index::ICamera> cam(
          dice_transaction->edit<nv::index::ICamera>(m_camera_tag));
        check_success(cam.is_valid_interface());

        const mi::math::Vector<mi::Float32, 3> from( 800.0f,   800.0f,  1050.0f);
        mi::math::Vector<mi::Float32, 3>       dir (  -0.7f,     0.0f,    -1.0f);
        const mi::math::Vector<mi::Float32, 3> up  (   0.0f,     1.0f,     0.0f);
        dir.normalize();

        cam->set(from, dir, up);
      }
    }
    dice_transaction->commit();
  }

  // Set up viewport 3: enable the label 3 and change the camera
  {
```

```
    const mi::Size view_idx = 3;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "3");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      // localize labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        mi::Sint32 ret_localize = 1;
        ret_localize = dice_transaction->localize(m_label_tag_vec[i],
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }

      // localize camera
      {
        mi::Sint32 ret_localize = 1;
        ret_localize = dice_transaction->localize(m_camera_tag,
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }

      // localize material
      {
        check_success(m_mesh_phong_1_tag.is_valid());
        mi::Sint32 ret_localize = 1;
        ret_localize = dice_transaction->localize(m_mesh_phong_1_tag,
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }

      // edit labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        mi::base::Handle<nv::index::ILabel_3D> label(
          dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
        check_success(label.is_valid_interface());
        if (i == view_idx)
        {
          label->set_enabled(true);
          label->set_text("zoom + green");
        }
        else
        {
          label->set_enabled(false);
        }
```

```
      }

      // edit camera
      {
        mi::base::Handle<nv::index::ICamera> cam(
          dice_transaction->edit<nv::index::ICamera>(m_camera_tag));
        check_success(cam.is_valid_interface());

        const mi::math::Vector<mi::Float32, 3> from( 450.0f,   800.0f,   900.0f);
        mi::math::Vector<mi::Float32, 3>       dir (  0.0f,     0.0f,    -1.0f);
        const mi::math::Vector<mi::Float32, 3> up  (  0.0f,     1.0f,     0.0f);
        dir.normalize();

        cam->set(from, dir, up);
      }

      // edit material
      {
        mi::base::Handle<nv::index::IPhong_gl> phong_1(
          dice_transaction->edit<nv::index::IPhong_gl>(m_mesh_phong_1_tag));
        check_success(phong_1.is_valid_interface());
        phong_1->set_opacity(m_mesh_mat_opacity_1);
      phong_1->set_ambient( mi::math::Color(0.1f, 0.1f, 0.1f,  m_mesh_mat_opacity_1));
      phong_1->set_specular(mi::math::Color(0.4f, 0.4f, 0.75f, m_mesh_mat_opacity_1));
      phong_1->set_diffuse( mi::math::Color(0.1f, 0.8f, 0.1f,  m_mesh_mat_opacity_1));
      }
    }
    dice_transaction->commit();
  }

  // Set up viewport 0 with the global scope: disable labels
  {
    const mi::Size view_idx = 0;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "0");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // edit scene elements
    {
      // edit labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        mi::base::Handle<nv::index::ILabel_3D> label(
          dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
        check_success(label.is_valid_interface());

        if (i == view_idx)
        {
```

```
          // label->set_text("View 0");
        }
        else
        {
          label->set_enabled(false);
        }
      }
    }
    dice_transaction->commit();
  }
}

nv::index::IViewport_list* Multi_view_trimesh::filter_enabled_viewport_index_list()
{
  mi::base::Handle<nv::index::IViewport_list> new_viewport_list;
  {
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    assert(dice_transaction.is_valid_interface());
    {
      assert(m_session_tag.is_valid());
      mi::base::Handle<const nv::index::ISession> session(
        dice_transaction->access<nv::index::ISession>(m_session_tag));
      assert(session.is_valid_interface());

      // create viewport from a session
      new_viewport_list = session->create_viewport_list();
    }
    dice_transaction->commit();
  }

  check_success(new_viewport_list.is_valid_interface());
  const mi::Size nb_views = m_viewport_list->size();

  std::stringstream sstr;
  mi::Size nb_enabled = 0;
  for (mi::Size i = 0; i < nb_views; ++i)
  {
    if (m_enable_view_idx_vec[i])
    {
      sstr << i << " ";
      mi::base::Handle<nv::index::IViewport> vp (m_viewport_list->get(i));
      new_viewport_list->append(vp.get());
      ++nb_enabled;
    }
  }
  // copy advisory state
  new_viewport_list->set_advisory_enabled(m_viewport_list->get_advisory_enabled());

  if (nb_enabled == nb_views)
  {
    INFO_LOG << "All views are enabled.";
  }
  else
  {
    INFO_LOG << "Filtered views. Enabled: " << sstr.str();
  }
```

```
  new_viewport_list->retain();
  return new_viewport_list.get();
}

bool Multi_view_trimesh::render_one_frame(mi::Sint32 frame_idx)
{
  bool success = true;

  // Render a frame and save the rendered image to a file.
  // Only save the file at the end of the iteration.
  std::string fname = "";
  if (!(m_outfname.empty()))
  {
    fname = get_output_file_name(m_outfname, frame_idx);
  }
  check_success(m_index_rendering.is_valid_interface());

  // set up canvas. output_fname.empty() is valid (no output file)
  m_image_file_canvas->set_rgba_file_name(fname.c_str());

  // set advisory to see the multiview rendering details
  m_viewport_list->set_advisory_enabled(true);

  mi::base::Handle<nv::index::IViewport_list> cur_viewport_list(
    filter_enabled_viewport_index_list());
  check_success(cur_viewport_list.is_valid_interface());

  mi::base::Handle<nv::index::IFrame_results_list> frame_results_list(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      cur_viewport_list.get()));
  check_success(frame_results_list.is_valid_interface());

  if (frame_results_list->size() == 0)
  {
    ERROR_LOG << "IIndex_rendering rendering call has no results.";
    success = false;
  }
  else
  {
    for (mi::Size i = 0; i < frame_results_list->size(); ++i)
    {
      mi::base::Handle<nv::index::IFrame_results>
        frame_results(frame_results_list->get(i));
      check_success(frame_results.is_valid_interface());

    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      check_success(err_set.is_valid_interface());
      if (err_set->any_errors())
      {
        std::ostringstream os;

     const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
```

```
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

      ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        success = false;
       }
     }
   }

   // verify the generated frame
   if (!(m_verify_image_path_base.empty()))
   {
    const std::string ref_img_fpath = get_output_file_name(m_verify_image_path_base, frame_idx);
     if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), ref_img_fpath, get_options())))
     {
       success = false;
     }
   }

   return success;
}

void Multi_view_trimesh::create_views() const
{
   // Check session and multiple views in the arc
   check_success(m_session_tag.is_valid());
   check_success(m_viewport_list.is_valid_interface());
   check_success(m_viewport_list->size() == 0);

   // Multiple viewport itself lives in the global scope.
   mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
     m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
   check_success(dice_transaction.is_valid_interface());
   {
     mi::base::Handle<const nv::index::ISession> session(
       dice_transaction->access<nv::index::ISession>(m_session_tag));
     check_success(session.is_valid_interface());

     // 0. create a single viewport in the global scope.
     {
       mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
       check_success(viewport.is_valid_interface());

       const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0, 256);
       const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

       viewport->set_position(viewport_pos);
       viewport->set_size(viewport_size);
       viewport->set_scope(m_global_scope.get()); // ref count up

       m_viewport_list->append(viewport.get());
     }
```

```
    // 1. viewport
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256, 256);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }

    // 2. viewport
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0,   0);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }

    // 3. viewport
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256,   0);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);
```

```
      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }
  }
  dice_transaction->commit();
}


mi::neuraylib::Tag Multi_view_trimesh::create_append_label_to_group(
  nv::index::IScene*                      scene_edit,
  nv::index::ITransformed_scene_group*    group_node,
  const mi::math::Vector<mi::Float32, 3>& label_point,
  const std::string&                      label_str,
  const mi::Float32                       label_height,
  const mi::Float32                       label_width,
  const mi::math::Color_struct&           fg_col,
  const mi::math::Color_struct&           bg_col,
  mi::neuraylib::IDice_transaction*       dice_transaction)
{
  check_success(scene_edit != 0);
  check_success(group_node != 0);

  // Add font to the scene description
 mi::base::Handle<nv::index::IFont> font(scene_edit->create_attribute<nv::index::IFont>());
  check_success(font.is_valid_interface());

  if (font->set_file_name(m_font_fpath.c_str()))
  {
    INFO_LOG << "set the font path [" << m_font_fpath << "]";
  }
  else
  {
    ERROR_LOG << "Can not find the font path [" << m_font_fpath << "], "
         << "the rendering result may not correct.";
  }
  font->set_font_resolution(64.0f);
 const mi::neuraylib::Tag font_tag = dice_transaction->store_for_reference_counting(font.get());
  INFO_LOG << "Created a font: tag: " << font_tag;
  check_success(font_tag.is_valid());
  group_node->append(font_tag, dice_transaction);

  // Add a label
  mi::neuraylib::Tag label_tag;
  {
    mi::base::Handle<nv::index::ILabel_layout> label_layout(
      scene_edit->create_attribute<nv::index::ILabel_layout>());
    check_success(label_layout.is_valid_interface());
    const mi::Float32 padding = 20.0f;
    label_layout->set_padding(padding);
    label_layout->set_color(fg_col, bg_col);
    const mi::neuraylib::Tag label_layout_tag =
      dice_transaction->store_for_reference_counting(label_layout.get());
    INFO_LOG << "Created a label_layout: tag: " << label_layout_tag;
    check_success(label_layout_tag.is_valid());
```

```
    group_node->append(label_layout_tag, dice_transaction);

   mi::base::Handle<nv::index::ILabel_3D> label(scene_edit->create_shape<nv::index::ILabel_3D>());
    check_success(label.is_valid_interface());
    label->set_text(label_str.c_str());

    const mi::math::Vector<mi::Float32, 3> right(1.0f, 0.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 3> up   (0.0f, 1.0f, 0.0f);
    label->set_geometry(label_point, right, up, label_height, label_width);
    label_tag = dice_transaction->store_for_reference_counting(label.get());
    INFO_LOG << "Created a label: tag: " << label_tag;
    check_success(label_tag.is_valid());
    group_node->append(label_tag, dice_transaction);
  }
  return label_tag;
}

void Multi_view_trimesh::setup_scene(mi::neuraylib::IDice_transaction* dice_transaction)
{
  check_success(dice_transaction != 0);

  // Access the session instance from the database.
  mi::base::Handle<const nv::index::ISession> session(
    dice_transaction->access<const nv::index::ISession>(m_session_tag));
  check_success(session.is_valid_interface());

  // Access (edit mode) the scene instance from the database.
  mi::base::Handle<nv::index::IScene> scene_edit(
    dice_transaction->edit<nv::index::IScene>(session->get_scene()));
  check_success(scene_edit.is_valid_interface());

  // Create static group node for large data
  mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
    scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
  check_success(static_group_node.is_valid_interface());

  // Added a light and a material to the static group node for trimesh
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color color_intensity(1.0f, 1.0f, 1.0f, 1.0f);
    headlight->set_intensity(color_intensity);
    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));

   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());
    static_group_node->append(headlight_tag, dice_transaction);
    INFO_LOG << "Created a headlight: tag = " << headlight_tag.id;

    // Material for the trimesh
   mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_g
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(mi::math::Color(0.1f, 0.1f, 0.1f, 1.0f));
    phong_1->set_diffuse(mi::math::Color(0.45f, 0.3f, 0.3f, 1.0f));
    phong_1->set_specular(mi::math::Color(0.4f, 0.4f, 0.75f, 1.0f));
```

```
  if (m_mesh_mat_opacity_0 < 0.99f)
  {
   INFO_LOG << "Material opacity is less than 1.0 (" << m_mesh_mat_opacity_0 << ")";
  }
  phong_1->set_opacity(m_mesh_mat_opacity_0);
  phong_1->set_shininess(85);
  check_success(!m_mesh_phong_1_tag.is_valid());
m_mesh_phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.get());
  check_success(m_mesh_phong_1_tag.is_valid());

  scene_edit->append(m_mesh_phong_1_tag, dice_transaction);
  static_group_node->append(m_mesh_phong_1_tag, dice_transaction);
 INFO_LOG << "Created a phong material for the mesh: tag = " << m_mesh_phong_1_tag.id;
}

// Add a triangle mesh
{
  // triangle mesh creation parameter
  nv::index::app::String_dict triangle_mesh_opt;
  triangle_mesh_opt.insert("args::type",       "triangle_mesh");
 triangle_mesh_opt.insert("args::importer",  "nv::index::plugin::base_importer.Triangle_mesh_imp
  triangle_mesh_opt.insert("args::input_file", m_mesh_file);
  triangle_mesh_opt.insert("args::bbox",       m_mesh_bbox);
  nv::index::IDistributed_data_import_callback* importer_callback =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::base_importer.Triangle_mesh_importer",
      triangle_mesh_opt);

  const mi::math::Bbox<mi::Float32, 3> mesh_bbox =
    nv::index::app::get_bbox_from_string<mi::Float32,3>(m_mesh_bbox);
  INFO_LOG << "mesh_file: " << m_mesh_file;
  INFO_LOG << "mesh_bbox: " << mesh_bbox;

  // Create the triangle mesh scene element
  mi::base::Handle<nv::index::ITriangle_mesh_scene_element> mesh(
  scene_edit->create_triangle_mesh(mesh_bbox, importer_callback, dice_transaction));
  check_success(mesh != 0);

  // scene element properties
  mesh->set_enabled(true);

  // storing the mesh in the data (base) store
  m_trimesh_tag = dice_transaction->store_for_reference_counting(mesh.get());
  check_success(m_trimesh_tag.is_valid());

  // append mesh to the static scene group
  static_group_node->append(m_trimesh_tag, dice_transaction);

 m_static_group_node_tag = dice_transaction->store_for_reference_counting(static_group_node.get(
  check_success(m_static_group_node_tag.is_valid());

  // append the static scene group to the hierachical scene description.
  scene_edit->append(m_static_group_node_tag, dice_transaction);
}

// Create and add labels
```

```
{
  // Add a scene group where the shapes should be added
  mi::base::Handle<nv::index::ITransformed_scene_group> label_group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(label_group_node.is_valid_interface());

  // Add a material
  {
    // Add a fully ambient material for the planes, so that lighting doesn't matter
    mi::math::Color ambient_color (1.0f, 1.0f, 1.0f, 1.0f);
    mi::math::Color diffuse_color (0.0f, 0.0f, 0.0f, 1.0f);
    mi::math::Color specular_color(0.0f, 0.0f, 0.0f, 1.0f);
    mi::Float32     shiness = 100.0f;
    mi::base::Handle<nv::index::IPhong_gl> phong_1(
      scene_edit->create_attribute<nv::index::IPhong_gl>());
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(ambient_color);
    phong_1->set_diffuse(diffuse_color);
    phong_1->set_specular(specular_color);
    phong_1->set_shininess(shiness);

  mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.get())
    INFO_LOG << "Created phong for labels: tag: " << phong_1_tag;
    check_success(phong_1_tag.is_valid());
    label_group_node->append(phong_1_tag, dice_transaction);
  }

 mi::math::Color_struct fg_col; fg_col.r = 0.25f; fg_col.g = 0.95f; fg_col.b = 0.25f; fg_col.a = 1.0
 mi::math::Color_struct bg_col; bg_col.r = 0.4f;  bg_col.g = 0.5f;  bg_col.b = 0.4f;  bg_col.a = 0.5f

  m_label_tag_vec.clear();
  mi::math::Vector<mi::Float32, 3> label_pos (300.0f, 600.0f, 600.0f);
  std::string str = "View 0";
  const mi::Float32 height = 70.0f;
  const mi::Float32 width  = -1.0f; // automated width
  mi::neuraylib::Tag created_label =
  create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                  height, width, fg_col, bg_col, dice_transaction);
  check_success(created_label);
  m_label_tag_vec.push_back(created_label);

  label_pos.y += 100.0f;
  str = "View 1";
  created_label =
  create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                  height, width, fg_col, bg_col, dice_transaction);
  check_success(created_label.is_valid());
  m_label_tag_vec.push_back(created_label);

  label_pos.y += 100.0f;
  str = "View 2";
  created_label =
  create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                  height, width, fg_col, bg_col, dice_transaction);
  check_success(created_label.is_valid());
  m_label_tag_vec.push_back(created_label);
```

```
    label_pos.y += 100.0f;
    str = "View 3";
    created_label =
    create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                    height, width, fg_col, bg_col, dice_transaction);
    check_success(created_label.is_valid());
    m_label_tag_vec.push_back(created_label);

    // Finally append everything to the root of the hierachical scene description
    const mi::neuraylib::Tag label_group_node_tag =
      dice_transaction->store_for_reference_counting(label_group_node.get());
    check_success(label_group_node_tag.is_valid());
    INFO_LOG << "Created a label_group: " << label_group_node_tag;

    check_success(label_group_node_tag.is_valid());
    scene_edit->append(label_group_node_tag, dice_transaction);
    m_label_group_node_tag = label_group_node_tag;
  }

  // Define a region of interest for the entire scene in the
  // scene's global coordinate system.
  // Set the region of interest
  check_success(m_roi.is_volume());
  scene_edit->set_clipped_bounding_box(m_roi);

  // Set the scene global transformation matrix.
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f,  0.0f,  0.0f,  0.0f,
    0.0f,  1.0f,  0.0f,  0.0f,
    0.0f,  0.0f,  1.0f,  0.0f,
    0.0f,  0.0f,  0.0f,  1.0f
    );

  scene_edit->set_transform_matrix(transform_mat);

  // Create a camera
  mi::base::Handle< nv::index::IPerspective_camera > cam(
    scene_edit->create_camera<nv::index::IPerspective_camera>());
  check_success(cam.is_valid_interface());
  m_camera_tag = dice_transaction->store(cam.get());
  check_success(m_camera_tag.is_valid());

  // set up the camera
  setup_camera(m_camera_tag, dice_transaction);

  // ... and add the camera to the scene.
  scene_edit->set_camera(m_camera_tag);
}

bool Multi_view_trimesh::setup_main_host()
{
 // Access the IndeX rendering query interface for querying performance values and pick results
  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
```

```
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  // DiCE database access
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    //------------------------------------------------------------------------
    // Setup session information
    m_session_tag =
      m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<nv::index::ISession>(
        m_session_tag));
    check_success(session.is_valid_interface());

    // Setup the main multiple views, but it is empty at here.
    m_viewport_list = session->create_viewport_list();
    check_success(m_viewport_list.is_valid_interface());

    //------------------------------------------------------------------------
    // Scene setup in the global scope
    setup_scene(dice_transaction.get());
  }
  dice_transaction->commit();

  // Set canvas size
  const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
  m_image_file_canvas->set_resolution(buffer_resolution);

  INFO_LOG << "Initialization complete.";

  return true;
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "4");                                    // log level
  sdict.insert("font_fpath", "/usr/share/fonts/dejavu/DejaVuSans.ttf"); // font path
 sdict.insert("mesh_file", "../create_trianglemesh/jacket.bin");     // input triangle mesh file nar
 sdict.insert("mesh_bbox", "368.38 727.758 482.661 630.081 995.543 590.383"); // mesh bbox
 sdict.insert("mesh_mat_opacity_0", "1.0");                   // mesh material opacity 0 (default 1.
 sdict.insert("mesh_mat_opacity_1", "0.5");                   // mesh material opacity 1 (default 0.
  sdict.insert("roi", "0 0 0 650 1000 600");                           // input roi
 sdict.insert("outfname", "frame_multi_view_trimesh");            // output file base name
  sdict.insert("verify_image_path_base", "");                         // for unit test
  sdict.insert("unittest", "0");                                     // unit test mode
 sdict.insert("enable_vidx_vec", "1 1 1 1");                   // enabled view indices vector

  // Load IndeX library via Index_connect
```

```
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");
  sdict.insert("index::app::plugins::base_importer::enabled", "true");

  // Initialize application
  Multi_view_trimesh multi_view_trimesh;
  multi_view_trimesh.initialize(argc, argv, sdict);
  check_success(multi_view_trimesh.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = multi_view_trimesh.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.26 multi_view_volume.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/icolormap.h>
#include <nv/index/ifont.h>
#include <nv/index/iindex.h>
#include <nv/index/ilabel.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>
#include <nv/index/isparse_volume_rendering_properties.h>
#include <nv/index/iviewport.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/app_rendering_context.h"
#include "utility/canvas_utility.h"

#include <sstream>
#include <iostream>

class Multi_view_volume:
  public nv::index::app::Index_connect
{
public:
  Multi_view_volume()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Multi_view_volume() ctor";
  }

  virtual ~Multi_view_volume()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Multi_view_volume() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
```

```
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Setup camera to see this example scene
  // \param[in] camera_tag camera tag
  // \param[in] dice_transaction dice transaction
  void setup_camera(
    const mi::neuraylib::Tag&         camera_tag,
    mi::neuraylib::IDice_transaction* dice_transaction) const;
  // Localize scene elements
  void localize_scene_element();
  // Filter enable viewport index list
  // \return newly created filtered viewport list
  nv::index::IViewport_list* filter_enabled_viewport_index_list();
  // Render one frame
  // \param[in] frame_idx        current frame index
  // \return true when success
  bool render_one_frame(mi::Sint32 frame_idx);
  // Create multiple viewports
  //
  // The view structure of this example
  //
  //        +-----------------+-----------------+(512,512)
  //        | viewport 0      | viewport 1       |
  //        | global scope    | camera change    |
  // (0,256)+-----------------+-----------------+(512,256)
  //        | viewport 2      | viewport 3      |
  //        |                 |                 |
  //        +-----------------+-----------------+
  //       (0,0)              (256,0)           (512,0)
  //
  //
 // \param[in,out] arc  application rendering context. multiple views will be updated.
  void create_views();

  // Create a synthetic volume.
  //
  // \note The new volume is not added to the hierachical scene description.
  //
  // \param[in] scene            scene root object to update the scene
  // \param[in] volume_size      volume size to be created
  // \param[in] dice_transaction dice db transaction
  // \return tag of the newly created volume
```

```
  mi::neuraylib::Tag create_synthetic_sparse_volume(
    nv::index::IScene*                    scene,
    const mi::math::Vector<mi::Uint32, 3>& volume_size,
    mi::neuraylib::IDice_transaction*     dice_transaction) const;

  // Add a label
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node   parent group node
  // \param[in] label_point  label's corner position in the object space
  // \param[in] label_str    label contents string
  // \param[in] label_height label height in the object space
  // \param[in] label_width  label width in the object space
  // \param[in] fg_col       label foreground color
  // \param[in] bg_col       label background color
  // \param[in] opt_map command line option map
  // \param[in] dice_transaction db transaction
  // \return created label tag
  mi::neuraylib::Tag create_append_label_to_group(
    nv::index::IScene*                     scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
    const mi::math::Vector<mi::Float32, 3>& label_point,
    const std::string&                     label_str,
    const mi::Float32                      label_height,
    const mi::Float32                      label_width,
    const mi::math::Color_struct&          fg_col,
    const mi::math::Color_struct&          bg_col,
    mi::neuraylib::IDice_transaction*      dice_transaction) const;
  // Set up the scene
  // Create a scene that contains a synthetically generated volume.
  // \param[in] volume_size      volume size to be created
  // \param[in] dice_transaction DiCE db transaction
  void setup_scene(
    const mi::math::Vector<mi::Uint32, 3>& volume_size,
    mi::neuraylib::IDice_transaction*     dice_transaction);
  // Set up as the main host
  // \return true when success
  bool setup_main_host();

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                 m_verify_image_path_base;
  std::string                                                           m_font_fpath;
  std::string                                                             m_iconfile;
  std::vector<bool>                                             m_enable_view_idx_vec;
  // camera tag
  mi::neuraylib::Tag                                                    m_camera_tag;
  // volume scene element tag
  mi::neuraylib::Tag                                                    m_volume_tag;
  // colormap 1 tag
```

```
  mi::neuraylib::Tag                                              m_colormap_1_tag;
   // colormap 40 tag
  mi::neuraylib::Tag                                             m_colormap_40_tag;
   // static group node tag for volume
  mi::neuraylib::Tag                                          m_static_group_node_tag;
   // label group node tag
  mi::neuraylib::Tag                                          m_label_group_node_tag;
   // label tag vector
  std::vector<mi::neuraylib::Tag>                                   m_label_tag_vec;
  mi::base::Handle<nv::index::IViewport_list>                         m_viewport_list;
};

mi::Sint32 Multi_view_volume::launch()
{
  // Set up
  {
    // const std::string enable_vidx_str = sdict.get("enable_vidx_vec");
   // const std::vector<bool> enable_vidx_vec = get_bool_vec_from_string(enable_vidx_str);
    check_success(setup_main_host());

    // Create multiple views in the arc.
    create_views();

    mi::Sint32 frame_idx = 0;
    // Render multiple views for a single render call. Before localize.
    {
      // Render a frame and save the rendered image to a file.
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }

    // localize scene elements
    localize_scene_element();

    // Render multiple views for a single render call.
    {
      // Render a frame and save the rendered image to a file.
      const bool is_success = render_one_frame(frame_idx);
      check_success(is_success);
      ++frame_idx;
    }
  }
  return 0;
}

bool Multi_view_volume::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));
  if (m_is_unittest)
  {
    sdict.insert("dice::verbose", "2");
  }
  m_outfname               = sdict.get("outfname");
  m_verify_image_path_base = sdict.get("verify_image_path_base");
  m_enable_view_idx_vec    = get_bool_vec_from_string(sdict.get("enable_vidx_vec"));
```

```
  m_font_fpath             = sdict.get("font_fpath");
  m_iconfile               = sdict.get("iconfile");

  info_cout("running " + com_name, sdict);
  info_cout("outfname = ["              + m_outfname +
       "], verify_image_path_base = [" + m_verify_image_path_base +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            print out this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "         verbose severity level (3 is info). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-font_fpath FONT_FILE_PATH]\n"
      << "            font file path. (default: " << m_font_fpath << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"

      << "        [-verify_image_path_base image path basename]\n"
    << "          when image_fname path basename exist, verify the rendering images.\n"
      << "            (default: [" << m_verify_image_path_base << "])\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode. "
      << "(default: [" << m_is_unittest << "])\n"

      << "        [-enable_vidx_vec enabled_vector]\n"
      << "            Specify which viewport is enabled by a bool vector.\n"
      << "            (default: [" << sdict.get("enable_vidx_vec") << "])"

      << std::endl;
    exit(1);
  }
  return true;
}

void Multi_view_volume::setup_camera(
  const mi::neuraylib::Tag&         camera_tag,
  mi::neuraylib::IDice_transaction* dice_transaction) const
{
  check_success(camera_tag.is_valid());

  mi::base::Handle<nv::index::IPerspective_camera> cam(
    dice_transaction->edit<nv::index::IPerspective_camera>(camera_tag));
  check_success(cam.is_valid_interface());

  // Set the camera parameters to see the whole scene
  const mi::math::Vector<mi::Float32, 3> from(300.0f, 300.0f, 1000.0f);
```

```
  mi::math::Vector<mi::Float32, 3>       dir (  0.0f,   0.0f,   -1.0f);
  const mi::math::Vector<mi::Float32, 3> up  (  0.0f,   1.0f,    0.0f);
  dir.normalize();

  cam->set(from, dir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(1000.0f);
}

void Multi_view_volume::localize_scene_element()
{
  check_success(m_viewport_list.is_valid_interface());

  // Process local views first.

  // Set up viewport 1: enable the label 1 and change the camera
  {
    const mi::Size view_idx = 1;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "1");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize and edit scene elements
    {
      //----- localize
      // localize the scene group containing the volume
      mi::Sint32 ret_localize = 1;
      ret_localize = dice_transaction->localize(m_static_group_node_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      // localize labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        ret_localize = dice_transaction->localize(m_label_tag_vec[i],
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }

      // localize colormap
      ret_localize = dice_transaction->localize(m_colormap_1_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_colormap_40_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
```

```
check_success(ret_localize == 0);

// localize camera
ret_localize = dice_transaction->localize(m_camera_tag,
                    mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
check_success(ret_localize == 0);


//----- edit
// edit disable the scene group containing the volume
{
  // edit the scene group in order to disable it
  mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
    dice_transaction->edit<nv::index::IStatic_scene_group>(
      m_static_group_node_tag));
  check_success(static_group_node.is_valid_interface());

  static_group_node->set_enabled(false);
}

// edit labels
for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
{
  mi::base::Handle<nv::index::ILabel_3D> label(
    dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
  check_success(label.is_valid_interface());
  if (i == view_idx)
  {
    label->set_enabled(true);
    label->set_text("View 1: right");
  }
  else
  {
    label->set_enabled(false);
  }
}

// edit camera
{
  mi::base::Handle<nv::index::ICamera> cam(
    dice_transaction->edit<nv::index::ICamera>(m_camera_tag));
  check_success(cam.is_valid_interface());

  const mi::math::Vector<mi::Float32, 3> from(  0.0f,  300.0f,  1000.0f);
  mi::math::Vector<mi::Float32, 3>       dir (  0.5f,    0.0f,    -1.0f);
  const mi::math::Vector<mi::Float32, 3> up  (  0.0f,    1.0f,     0.0f);
  dir.normalize();

  cam->set(from, dir, up);
}

// edit colormap
{
  mi::base::Handle<nv::index::IColormap> colormap_1(
    dice_transaction->edit<nv::index::IColormap>(m_colormap_1_tag));
  colormap_1->set_enabled(true);
  mi::base::Handle<nv::index::IColormap> colormap_40(
```

```
        dice_transaction->edit<nv::index::IColormap>(m_colormap_40_tag));
      colormap_40->set_enabled(false);
    }
  }
  dice_transaction->commit();
}

// Set up viewport 2: enable the label 2 and change the camera and colormap
{
  const mi::Size view_idx = 2;
  check_success(m_viewport_list->size() > view_idx);

  mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
  check_success(viewport != 0);

  mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
  check_success(cur_scope);
  check_success(std::string(cur_scope->get_id()) == "2");

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());

  mi::base::Handle<const nv::index::ISession> session(
    dice_transaction->access<nv::index::ISession>(m_session_tag));
  check_success(session.is_valid_interface());

  // localize scene elements
  {
    //----- Localize
    // localize the scene root element
    mi::Sint32 ret_localize = 1;
    ret_localize = dice_transaction->localize(session->get_scene(),
                        mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
    check_success(ret_localize == 0);

    // localize labels
    for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
    {
      ret_localize = dice_transaction->localize(m_label_tag_vec[i],
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
    }

    // localize colormap
    ret_localize = dice_transaction->localize(m_colormap_1_tag,
                        mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
    check_success(ret_localize == 0);
    ret_localize = dice_transaction->localize(m_colormap_40_tag,
                        mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
    check_success(ret_localize == 0);

    // localize camera, volume scene element
    ret_localize = dice_transaction->localize(m_camera_tag,
                        mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
    check_success(ret_localize == 0);
```

```
// localize volume
ret_localize = dice_transaction->localize(m_volume_tag,
                     mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
check_success(ret_localize == 0);

//----- Edit
// change the region of interest of the scene
{
  mi::base::Handle<nv::index::IScene> scene(
    dice_transaction->edit<nv::index::IScene>(session->get_scene()));
  check_success(scene.is_valid_interface());

  // modify the region of interest
  mi::math::Bbox<mi::Float32, 3> roi(scene->get_clipped_bounding_box());
  roi.max.x = 700.f; // larger than before
  roi.min.y = 200.f; // smaller so that the label is visible
  scene->set_clipped_bounding_box(roi);
}

// edit labels
for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
{
  mi::base::Handle<nv::index::ILabel_3D> label(
    dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
  check_success(label.is_valid_interface());
  if (i == view_idx)
  {
    label->set_enabled(true);
    label->set_text("View 2: left");
  }
  else
  {
    label->set_enabled(false);
  }
}

// edit camera and volume scene element
{
  mi::base::Handle<nv::index::ICamera> cam(
    dice_transaction->edit<nv::index::ICamera>(m_camera_tag));
  check_success(cam.is_valid_interface());

  const mi::math::Vector<mi::Float32, 3> from(600.0f,  300.0f,  1000.0f);
  mi::math::Vector<mi::Float32, 3>       dir ( -0.5f,    0.0f,    -1.0f);
  const mi::math::Vector<mi::Float32, 3> up  (  0.0f,    1.0f,     0.0f);
  dir.normalize();

  cam->set(from, dir, up);
}

// edit colormap
{
  mi::base::Handle<nv::index::IColormap> colormap_1(
    dice_transaction->edit<nv::index::IColormap>(m_colormap_1_tag));
  colormap_1->set_enabled(false);
  mi::base::Handle<nv::index::IColormap> colormap_40(
    dice_transaction->edit<nv::index::IColormap>(m_colormap_40_tag));
```

```
        colormap_40->set_enabled(true);
      }
    }
    dice_transaction->commit();
  }

  // Set up viewport 3: enable the label 3 and change the camera
  {
    const mi::Size view_idx = 3;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "3");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // localize scene elements
    {
      //----- Localize
      mi::Sint32 ret_localize = 1;
      // localize labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        ret_localize = dice_transaction->localize(m_label_tag_vec[i],
                            mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
        check_success(ret_localize == 0);
      }

      // localize colormap
      ret_localize = dice_transaction->localize(m_colormap_1_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);
      ret_localize = dice_transaction->localize(m_colormap_40_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      // localize camera
      ret_localize = dice_transaction->localize(m_camera_tag,
                          mi::neuraylib::IDice_transaction::LOCAL_SCOPE);
      check_success(ret_localize == 0);

      //----- Edit
      // edit labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
        mi::base::Handle<nv::index::ILabel_3D> label(
          dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
        check_success(label.is_valid_interface());
        if (i == view_idx)
        {
          label->set_enabled(true);
```

```
          label->set_text("View 3: Zoom");
        }
        else
        {
          label->set_enabled(false);
        }
      }

      // edit camera
      {
        mi::base::Handle<nv::index::ICamera> cam(
          dice_transaction->edit<nv::index::ICamera>(m_camera_tag));
        check_success(cam.is_valid_interface());

        const mi::math::Vector<mi::Float32, 3> from(300.0f,  300.0f,   700.0f);
        mi::math::Vector<mi::Float32, 3>       dir (  0.0f,    0.0f,    -1.0f);
        const mi::math::Vector<mi::Float32, 3> up  (  0.0f,    1.0f,     0.0f);
        dir.normalize();

        cam->set(from, dir, up);
      }

      // edit colormap
      {
        mi::base::Handle<nv::index::IColormap> colormap_1(
          dice_transaction->edit<nv::index::IColormap>(m_colormap_1_tag));
        colormap_1->set_enabled(true);
        mi::base::Handle<nv::index::IColormap> colormap_40(
          dice_transaction->edit<nv::index::IColormap>(m_colormap_40_tag));
        colormap_40->set_enabled(false);
      }

    }
    dice_transaction->commit();
  }

  // Set up viewport 0 with the global scope: disable labels
  {
    const mi::Size view_idx = 0;
    check_success(m_viewport_list->size() > view_idx);

    mi::base::Handle<nv::index::IViewport> viewport(m_viewport_list->get(view_idx));
    check_success(viewport != 0);

    mi::base::Handle<mi::neuraylib::IScope> cur_scope(viewport->get_scope());
    check_success(cur_scope);
    check_success(std::string(cur_scope->get_id()) == "0");

    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      cur_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    check_success(dice_transaction.is_valid_interface());

    // edit scene elements
    {
      // edit labels
      for (mi::Size i = 0; i < m_label_tag_vec.size(); ++i)
      {
```

```
        mi::base::Handle<nv::index::ILabel_3D> label(
          dice_transaction->edit<nv::index::ILabel_3D>(m_label_tag_vec[i]));
        check_success(label.is_valid_interface());

        if (i == view_idx)
        {
          // label->set_text("View 0");
        }
        else
        {
          label->set_enabled(false);
        }
      }
    }
    // edit colormap
    {
      mi::base::Handle<nv::index::IColormap> colormap_1(
        dice_transaction->edit<nv::index::IColormap>(m_colormap_1_tag));
      colormap_1->set_enabled(true);
      mi::base::Handle<nv::index::IColormap> colormap_40(
        dice_transaction->edit<nv::index::IColormap>(m_colormap_40_tag));
      colormap_40->set_enabled(false);
    }

    dice_transaction->commit();
  }
}

nv::index::IViewport_list* Multi_view_volume::filter_enabled_viewport_index_list()
{
  mi::base::Handle<nv::index::IViewport_list> new_viewport_list;
  {
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
      m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
    assert(dice_transaction.is_valid_interface());
    {
      assert(m_session_tag.is_valid());
      mi::base::Handle<const nv::index::ISession> session(
        dice_transaction->access<nv::index::ISession>(m_session_tag));
      assert(session.is_valid_interface());

      // create viewport from a session
      new_viewport_list = session->create_viewport_list();
    }
    dice_transaction->commit();
  }

  check_success(new_viewport_list.is_valid_interface());
  const mi::Size nb_views = m_viewport_list->size();

  std::stringstream sstr;
  mi::Size nb_enabled = 0;
  for (mi::Size i = 0; i < nb_views; ++i)
  {
    if (m_enable_view_idx_vec[i])
    {
      sstr << i << " ";
```

```
      mi::base::Handle<nv::index::IViewport> vp (m_viewport_list->get(i));
      new_viewport_list->append(vp.get());
      ++nb_enabled;
    }
  }
  // copy advisory state
  new_viewport_list->set_advisory_enabled(m_viewport_list->get_advisory_enabled());

  if (nb_enabled == nb_views)
  {
    INFO_LOG << "All views are enabled.";
  }
  else
  {
    INFO_LOG << "Filtered views. Enabled: " << sstr.str();
  }

  new_viewport_list->retain();
  return new_viewport_list.get();
}

bool Multi_view_volume::render_one_frame(mi::Sint32 frame_idx)
{
  bool success = true;

  // Render a frame and save the rendered image to a file.
  // Only save the file at the end of the iteration.
  std::string fname = "";
  if (!(m_outfname.empty()))
  {
    fname = get_output_file_name(m_outfname, frame_idx);
  }
  check_success(m_index_rendering.is_valid_interface());

  // set up canvas. output_fname.empty() is valid (no output file)
  m_image_file_canvas->set_rgba_file_name(fname.c_str());

  // set advisory to see the multiview rendering details
  m_viewport_list->set_advisory_enabled(true);

  mi::base::Handle<nv::index::IViewport_list> cur_viewport_list(
    filter_enabled_viewport_index_list());
  check_success(cur_viewport_list.is_valid_interface());

  mi::base::Handle<nv::index::IFrame_results_list> frame_results_list(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      cur_viewport_list.get()));
  check_success(frame_results_list.is_valid_interface());

  if (frame_results_list->size() == 0)
  {
    ERROR_LOG << "IIndex_rendering rendering call has no results.";
    success = false;
  }
  else
```

```
{
  for (mi::Size i = 0; i < frame_results_list->size(); ++i)
  {
    mi::base::Handle<nv::index::IFrame_results>
      frame_results(frame_results_list->get(i));
    check_success(frame_results.is_valid_interface());

  const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
    check_success(err_set.is_valid_interface());
    if (err_set->any_errors())
    {
      std::ostringstream os;

   const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      const mi::Uint32 nb_err = err_set->get_nb_errors();
      for (mi::Uint32 e = 0; e < nb_err; ++e)
      {
        if (e != 0) os << '\n';
        const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
        os << err->get_error_string();
      }

    ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
          << os.str();
      success = false;
    }
    else
    {
      // print some performance values
      const mi::base::Handle<nv::index::IPerformance_values> performance_values(
        frame_results->get_performance_values());

    const mi::Float32 rendering_time  = performance_values->get_time("time_total_rendering") / 100
    const mi::Float32 compositing_time = performance_values->get_time("time_total_compositing") /


      INFO_LOG << "-------------------------------------------";
      INFO_LOG << " Frame: " << frame_idx << ", Viewport: " << i;
      INFO_LOG << "-------------------------------------------";
      INFO_LOG << "Rendering time     : " << rendering_time << " [sec]";
      INFO_LOG << "Compositing time   : " << compositing_time << " [sec]";
    }
  }
}

// verify the generated frame
if (!(m_verify_image_path_base.empty()))
{
 const std::string ref_img_fpath = get_output_file_name(m_verify_image_path_base, frame_idx);
  if (!(verify_canvas_result(get_application_layer_interface(),
               m_image_file_canvas.get(), ref_img_fpath, get_options())))
  {
    success = false;
  }
}

return success;
```

```
}

void Multi_view_volume::create_views()
{
  // Check session and multiple views in the arc
  check_success(m_session_tag.is_valid());
  check_success(m_viewport_list.is_valid_interface());
  check_success(m_viewport_list->size() == 0);

  // Multiple view itself lives in the global scope.
 mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
  check_success(dice_transaction.is_valid_interface());
  {
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<nv::index::ISession>(m_session_tag));
    check_success(session.is_valid_interface());

    // 0. create a single viewport in the global scope.
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0, 256);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(m_global_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }

    // 1. viewport
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256, 256);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }

    // 2. viewport
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());
```

```
      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos(   0,   0);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }

    // 3. viewport
    {
      mi::base::Handle<nv::index::IViewport> viewport(session->create_viewport());
      check_success(viewport.is_valid_interface());

      mi::neuraylib::IScope* parent = 0; // this means global scope in this context
      mi::Uint8 privacy_level = 1;
      bool      is_temp       = false;
    mi::base::Handle<mi::neuraylib::IScope> local_scope(m_database->create_scope(parent, privacy_l
      check_success(local_scope.is_valid_interface());

      const mi::math::Vector<mi::Sint32, 2> viewport_pos( 256,   0);
      const mi::math::Vector<mi::Sint32, 2> viewport_size(256, 256);

      viewport->set_position(viewport_pos);
      viewport->set_size(viewport_size);
      viewport->set_scope(local_scope.get()); // ref count up

      m_viewport_list->append(viewport.get());
    }
  }
  dice_transaction->commit();
}

mi::neuraylib::Tag Multi_view_volume::create_synthetic_sparse_volume(
  nv::index::IScene*                    scene,
  const mi::math::Vector<mi::Uint32, 3>& volume_size,
  mi::neuraylib::IDice_transaction*     dice_transaction) const
{
  mi::math::Bbox<mi::Float32, 3> ijk_bbox;
  ijk_bbox.min.x = 0.0f;
  ijk_bbox.min.y = 0.0f;
  ijk_bbox.min.z = 0.0f;
  ijk_bbox.max.x = static_cast<mi::Float32>(volume_size.x);
  ijk_bbox.max.y = static_cast<mi::Float32>(volume_size.y);
  ijk_bbox.max.z = static_cast<mi::Float32>(volume_size.z);

  // sparse volume creation parameter
  nv::index::app::String_dict sparse_volume_opt;
  sparse_volume_opt.insert("args::type",           "sparse_volume");
```

```
  sparse_volume_opt.insert("args::importer",       "synthetic");
  std::stringstream sstr;
  sstr << "0 0 0 " << volume_size.x << " " << volume_size.y << " " << volume_size.z;
  sparse_volume_opt.insert("args::bbox",     sstr.str());
  sparse_volume_opt.insert("args::voxel_format",   "uint8");
  sparse_volume_opt.insert("args::synthetic_type", "sphere_0");
  nv::index::IDistributed_data_import_callback* importer_callback =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::base_importer.Sparse_volume_generator_synthetic",
      sparse_volume_opt);

  // Create the sparse volume scene element
  mi::math::Matrix<mi::Float32, 4, 4>    transform_mat(1.0f); // Identity matrix
  const mi::math::Vector<mi::Float32, 3> scale(1.0f, 1.0f, 1.0f);
  const mi::Float32                      rotate_k = 0.0f;
  const mi::math::Vector<mi::Float32, 3> translate(0.0f, 0.0f, 0.0f);
  const bool                             is_rendering_enabled = true;

  mi::base::Handle<nv::index::ISparse_volume_scene_element> sparse_volume(
   scene->create_sparse_volume(ijk_bbox, transform_mat, importer_callback, dice_transaction));
  check_success(sparse_volume.is_valid_interface());
  sparse_volume->set_enabled(is_rendering_enabled);

  DEBUG_LOG << "setup_imported_volume: "
     << "size: "      << volume_size.x << " " << volume_size.y << " " << volume_size.z << " "
     << ", scale: "    << scale.x     << " " << scale.y     << " " << scale.z     << " "
       << ", rotate_k: "                   << rotate_k
     << ", translate: " << translate.x  << " "  << translate.y << " "  << translate.z  << " "
       << ", is_rendering_enabled: "       << is_rendering_enabled
    ;

  const mi::neuraylib::Tag sparse_volume_tag =
    dice_transaction->store_for_reference_counting(sparse_volume.get());
  check_success(sparse_volume_tag.is_valid());

  INFO_LOG << "Creating a synthetic volume: size = [" << volume_size.x
    << " " << volume_size.y << " " << volume_size.z << "], tag = " << sparse_volume_tag.id;
  INFO_LOG << "This will take a while...";

  return sparse_volume_tag;
}

mi::neuraylib::Tag Multi_view_volume::create_append_label_to_group(
  nv::index::IScene*                     scene_edit,
  nv::index::ITransformed_scene_group*   group_node,
  const mi::math::Vector<mi::Float32, 3>& label_point,
  const std::string&                     label_str,
  const mi::Float32                      label_height,
  const mi::Float32                      label_width,
  const mi::math::Color_struct&          fg_col,
  const mi::math::Color_struct&          bg_col,
  mi::neuraylib::IDice_transaction*      dice_transaction) const
{
  check_success(scene_edit != 0);
  check_success(group_node != 0);
```

```
  // Add font to the scene description
 mi::base::Handle<nv::index::IFont> font(scene_edit->create_attribute<nv::index::IFont>());
  check_success(font.is_valid_interface());

  if (font->set_file_name(m_font_fpath.c_str()))
  {
    INFO_LOG << "set the font path [" << m_font_fpath << "]";
  }
  else
  {
    ERROR_LOG << "Can not find the font path [" << m_font_fpath << "], "
         << "the rendering result may not correct.";
  }
  font->set_font_resolution(64.0f);
 const mi::neuraylib::Tag font_tag = dice_transaction->store_for_reference_counting(font.get());
  INFO_LOG << "Created a font: tag: " << font_tag;
  check_success(font_tag.is_valid());
  group_node->append(font_tag, dice_transaction);

  // Add a label
  mi::neuraylib::Tag label_tag;
  {
    mi::base::Handle<nv::index::ILabel_layout> label_layout(
      scene_edit->create_attribute<nv::index::ILabel_layout>());
    check_success(label_layout.is_valid_interface());
    const mi::Float32 padding = 20.0f;
    label_layout->set_padding(padding);
    label_layout->set_color(fg_col, bg_col);
    const mi::neuraylib::Tag label_layout_tag =
      dice_transaction->store_for_reference_counting(label_layout.get());
    INFO_LOG << "Created a label_layout: tag: " << label_layout_tag;
    check_success(label_layout_tag.is_valid());
    group_node->append(label_layout_tag, dice_transaction);

   mi::base::Handle<nv::index::ILabel_3D> label(scene_edit->create_shape<nv::index::ILabel_3D>());
    check_success(label.is_valid_interface());
    label->set_text(label_str.c_str());

    const mi::math::Vector<mi::Float32, 3> right(1.0f, 0.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 3> up   (0.0f, 1.0f, 0.0f);
    label->set_geometry(label_point, right, up, label_height, label_width);
    label_tag = dice_transaction->store_for_reference_counting(label.get());
    INFO_LOG << "Created a label: tag: " << label_tag;
    check_success(label_tag.is_valid());
    group_node->append(label_tag, dice_transaction);
  }
  return label_tag;
}

void Multi_view_volume::setup_scene(
  const mi::math::Vector<mi::Uint32, 3>& volume_size,
  mi::neuraylib::IDice_transaction*      dice_transaction)
{
  check_success(dice_transaction != 0);

  // Create a scene that contains a synthetically generated volume.
```

```
// Access the session instance from the database.
mi::base::Handle<const nv::index::ISession> session(
  dice_transaction->access<const nv::index::ISession>(m_session_tag));
check_success(session.is_valid_interface());

// Access (edit mode) the scene instance from the database.
mi::base::Handle<nv::index::IScene> scene_edit(
  dice_transaction->edit<nv::index::IScene>(session->get_scene()));
check_success(scene_edit.is_valid_interface());

// Create static group node for large data
mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
  scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
check_success(static_group_node.is_valid_interface());

// Added a light and a material to the static group node
{
  // Add a light
  mi::base::Handle<nv::index::IDirectional_headlight> headlight(
    scene_edit->create_attribute<nv::index::IDirectional_headlight>());
  check_success(headlight.is_valid_interface());
  const mi::math::Color color_intensity(1.0f, 1.0f, 1.0f, 1.0f);
  headlight->set_intensity(color_intensity);
  headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));

 const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
  check_success(headlight_tag.is_valid());
  static_group_node->append(headlight_tag, dice_transaction);
  INFO_LOG << "Created a headlight: tag = " << headlight_tag.id;

  // Add a sparse_volume_render_properties to the scene.
 mi::base::Handle<nv::index::ISparse_volume_rendering_properties> sparse_render_prop(
  scene_edit->create_attribute<nv::index::ISparse_volume_rendering_properties>());
  sparse_render_prop->set_filter_mode(nv::index::SPARSE_VOLUME_FILTER_NEAREST);
  sparse_render_prop->set_sampling_distance(             1.0);
  sparse_render_prop->set_reference_sampling_distance(   1.0);
 sparse_render_prop->set_voxel_offsets(           mi::math::Vector<mi::Float32, 3>(0.0f, 0.0f, 0
  sparse_render_prop->set_preintegrated_volume_rendering(false);
  sparse_render_prop->set_lod_rendering_enabled(         false);
  sparse_render_prop->set_lod_pixel_threshold(           2.0);
  sparse_render_prop->set_debug_visualization_option(    0);
  const mi::neuraylib::Tag sparse_render_prop_tag
    = dice_transaction->store_for_reference_counting(sparse_render_prop.get());
  check_success(sparse_render_prop_tag.is_valid());
  static_group_node->append(sparse_render_prop_tag, dice_transaction);
 INFO_LOG << "Created a sparse_render_prop_tag: tag = " << sparse_render_prop_tag;

  // Create two colormaps. To see the colormap 1 in the first
  // frame, the order is 40 -> 1. Because we render, localize & edit, render.
  {
  const mi::Sint32 colormap_entry_id = 40; // same as demo application's colormap file 40
    const mi::neuraylib::Tag colormap_40_tag =
      create_colormap(colormap_entry_id, scene_edit.get(), dice_transaction);
    check_success(colormap_40_tag.is_valid());
    check_success(((!m_colormap_40_tag.is_valid())));
    m_colormap_40_tag = colormap_40_tag; // record tag
    static_group_node->append(colormap_40_tag, dice_transaction);
```

```
      INFO_LOG << "Created colormap_40: " << colormap_40_tag;
    }
    {
    const mi::Sint32 colormap_entry_id = 1; // same as demo application's colormap file 1, orange
      const mi::neuraylib::Tag colormap_1_tag =
        create_colormap(colormap_entry_id, scene_edit.get(), dice_transaction);
      check_success(colormap_1_tag.is_valid());
      m_colormap_1_tag = colormap_1_tag; // record tag
      static_group_node->append(colormap_1_tag, dice_transaction);
      INFO_LOG << "Created colormap_1: " << colormap_1_tag;
    }
  }

  // Add a synthetic volume to the scene.
  {
    check_success(!m_volume_tag.is_valid())
    m_volume_tag =
      create_synthetic_sparse_volume(
        scene_edit.get(),
        volume_size,
        dice_transaction);
    check_success(m_volume_tag.is_valid());

    INFO_LOG << "Creating a synthetic volume: size = [" << volume_size.x
        << " " << volume_size.y << " " << volume_size.z << "], tag = "
        << m_volume_tag.id;
    INFO_LOG << "This would take a while...";

    // Append the volume to the scene group
    static_group_node->append(m_volume_tag, dice_transaction);


    mi::neuraylib::Tag static_group_node_tag =
      dice_transaction->store_for_reference_counting(static_group_node.get());
    check_success(static_group_node_tag.is_valid());
    check_success(!m_static_group_node_tag.is_valid());
    m_static_group_node_tag = static_group_node_tag;

    scene_edit->append(static_group_node_tag, dice_transaction);

    std::stringstream sstr;
    sstr << "Created an synthetic volume: size = "
      << volume_size << ", tag = " << m_volume_tag.id;
    INFO_LOG << sstr.str();
  }

  // Create and add labels
  {
    // Add a scene group where the shapes should be added
    mi::base::Handle<nv::index::ITransformed_scene_group> label_group_node(
      scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
    check_success(label_group_node.is_valid_interface());

    // Add a material
    {
      // Add a fully ambient material for the planes, so that lighting doesn't matter
      mi::math::Color ambient_color (1.0f, 1.0f, 1.0f, 1.0f);
```

```
    mi::math::Color diffuse_color (0.0f, 0.0f, 0.0f, 1.0f);
    mi::math::Color specular_color(0.0f, 0.0f, 0.0f, 1.0f);
    mi::Float32     shiness = 100.0f;
    mi::base::Handle<nv::index::IPhong_gl> phong_1(
      scene_edit->create_attribute<nv::index::IPhong_gl>());
    check_success(phong_1.is_valid_interface());
    phong_1->set_ambient(ambient_color);
    phong_1->set_diffuse(diffuse_color);
    phong_1->set_specular(specular_color);
    phong_1->set_shininess(shiness);

  mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.get())
    INFO_LOG << "Created phong for labels: tag: " << phong_1_tag;
    check_success(phong_1_tag.is_valid());
    label_group_node->append(phong_1_tag, dice_transaction);
  }

 mi::math::Color_struct fg_col; fg_col.r = 0.25f; fg_col.g = 0.95f; fg_col.b = 0.25f; fg_col.a = 1.0
 mi::math::Color_struct bg_col; bg_col.r = 0.4f;  bg_col.g = 0.5f;  bg_col.b = 0.4f;  bg_col.a = 0.5f

  m_label_tag_vec.clear();
  mi::math::Vector<mi::Float32, 3> label_pos (10.0f, -250.0f, 10.0f);
  std::string str = "View 0";
  const mi::Float32 width = -1.0f; // automated width
  mi::neuraylib::Tag created_label =
  create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                  100.0f, width, fg_col, bg_col, dice_transaction);
  check_success(created_label);
  m_label_tag_vec.push_back(created_label);

  label_pos.y += 150.0f;
  str = "View 1";
  created_label =
  create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                  100.0f, width, fg_col, bg_col, dice_transaction);
  check_success(created_label.is_valid());
  m_label_tag_vec.push_back(created_label);

  label_pos.y += 150.0f;
  str = "View 2";
  created_label =
  create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                  100.0f, width, fg_col, bg_col, dice_transaction);
  check_success(created_label.is_valid());
  m_label_tag_vec.push_back(created_label);

  label_pos.y += 150.0f;
  str = "View 3";
  created_label =
  create_append_label_to_group(scene_edit.get(), label_group_node.get(), label_pos, str,
                  100.0f, width, fg_col, bg_col, dice_transaction);
  check_success(created_label.is_valid());
  m_label_tag_vec.push_back(created_label);

  // Finally append everything to the root of the hierachical scene description
  const mi::neuraylib::Tag label_group_node_tag =
    dice_transaction->store_for_reference_counting(label_group_node.get());
```

```
    check_success(label_group_node_tag.is_valid());
    INFO_LOG << "Created a label_group: " << label_group_node_tag;

    check_success(label_group_node_tag.is_valid());
    scene_edit->append(label_group_node_tag, dice_transaction);
    m_label_group_node_tag = label_group_node_tag;
  }

  // Define a region of interest for the entire scene in the
  // scene's global coordinate system.
  const mi::math::Bbox<mi::Float32, 3> region_of_interest(
    0.0f, 0.0f, 0.0f,
    static_cast<mi::Float32>(volume_size.x),
    static_cast<mi::Float32>(volume_size.y),
    static_cast<mi::Float32>(volume_size.z));

  // Set the region of interest
  check_success(region_of_interest.is_volume());
  scene_edit->set_clipped_bounding_box(region_of_interest);

  // Set the scene global transformation matrix.
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f,  0.0f,  0.0f,  0.0f,
    0.0f,  1.0f,  0.0f,  0.0f,
    0.0f,  0.0f,  1.0f,  0.0f,
    0.0f,  0.0f,  0.0f,  1.0f
    );

  scene_edit->set_transform_matrix(transform_mat);

  // Create a camera
  mi::base::Handle< nv::index::IPerspective_camera > cam(
    scene_edit->create_camera<nv::index::IPerspective_camera>());
  check_success(cam.is_valid_interface());
  m_camera_tag = dice_transaction->store(cam.get());
  check_success(m_camera_tag.is_valid());

  // set up the camera
  setup_camera(m_camera_tag, dice_transaction);

  // ... and add the camera to the scene.
  scene_edit->set_camera(m_camera_tag);
}

bool Multi_view_volume::setup_main_host()
{
 // Access the IndeX rendering query interface for querying performance values and pick results
  m_cluster_configuration =
    get_index_interface()->get_api_component<nv::index::ICluster_configuration>();
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
```

```
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  // DiCE database access
  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    //------------------------------------------------------------------------
    // Setup session information
    m_session_tag =
      m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<nv::index::ISession>(
        m_session_tag));
    check_success(session.is_valid_interface());

    // Setup the main multiple views, but it is empty at here.
    m_viewport_list = session->create_viewport_list();
    check_success(m_viewport_list.is_valid_interface());

    // Enable monitoring of performance values.
    mi::base::Handle<nv::index::IConfig_settings> config_settings(
      dice_transaction->edit<nv::index::IConfig_settings>(session->get_config()));
    check_success(config_settings.is_valid_interface());
    config_settings->set_monitor_performance_values(true);
    config_settings->set_automatic_span_control(true);
    config_settings->set_max_spans_per_machine(8);


    //------------------------------------------------------------------------
    // Scene setup in the global scope
    mi::math::Vector<mi::Uint32, 3> volume_size(600, 600, 32);
    setup_scene(volume_size, dice_transaction.get());
  }
  dice_transaction->commit();

  // set canvas size
  const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
  m_image_file_canvas->set_resolution(buffer_resolution);

  INFO_LOG << "Initialization complete.";

  return true;
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "4");                          // log level
  sdict.insert("font_fpath", "/usr/share/fonts/dejavu/DejaVuSans.ttf"); // font path
  sdict.insert("outfname", "frame_multi_view_volume");    // output file base name
  sdict.insert("verify_image_path_base", "");                  // for unit test
  sdict.insert("unittest", "0");                          // unit test mode
 sdict.insert("enable_vidx_vec", "1 1 1 1");            // enabled view indices vector

  // Load IndeX library via Index_connect
```

```
  sdict.insert("dice::network::mode", "OFF");

  // index setting
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // application_layer component loading
  sdict.insert("index::app::components::application_layer::component_name_list",
         "canvas_infrastructure image io");
  sdict.insert("index::app::plugins::base_importer::enabled", "true");

  // Initialize application
  Multi_view_volume multi_view_volume;
  multi_view_volume.initialize(argc, argv, sdict);
  check_success(multi_view_volume.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = multi_view_volume.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.27 normal_recalculation.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"
#include "utility/index_resource.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/idata_distribution.h>
#include <nv/index/iindex.h>
#include <nv/index/ilight.h>
#include <nv/index/imaterial.h>
#include <nv/index/iregular_heightfield_patch.h>
#include <nv/index/iscene.h>
#include <nv/index/isession.h>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/app_rendering_context.h"
#include "utility/canvas_utility.h"

#include <iostream>
#include <sstream>

class Normal_recalculation:
  public nv::index::app::Index_connect
{
public:
  Normal_recalculation()
    :
    Index_connect(),
    m_heightfield_tag(mi::neuraylib::NULL_TAG),
    m_frame_idx(0)
  {
    // INFO_LOG << "DEBUG: Normal_recalculation() ctor";
  }

  virtual ~Normal_recalculation()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Normal_recalculation() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
```

```
    nv::index::app::String_dict&          options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Render one frame
  // \return true when test success.
  bool render_frame(const std::string& output_fname,
           const std::string& verify_fname);
  // Get next frame index. This index will be incremented when
  // render_frame() called.
  //
  // \return current frame index
  mi::Sint32 get_next_frame_idx() const
  {
    return m_frame_idx;
  }
  // Compute heightfield
  //
  // Note: Only the arguments is_normal_recalc and
  // is_set_constant_normal are considered for normal
  // computation. The value sdict.get("use_normal_recalculation") is
  // disregarded.
  //
  // \param[in] is_normal_recalc       recalculate normal on when true
 // \param[in] is_set_constant_normal set constant normal when true (if is_normal_recalc false).
  // \param[in] compute_id             user defined compute call id
  void compute_heightfield(
    bool       is_normal_recalc,
    bool       is_set_constant_normal,
    mi::Sint32 compute_id);
  // Create the scene.
  void create_scene();
  // Camera setup to see all the scene.
  // \param[in] cam     the camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;


  // This session tag
 mi::neuraylib::Tag                                                m_session_tag;
  // NVIDIA IndeX cluster configuration
 mi::base::Handle<nv::index::ICluster_configuration>                   m_cluster_configuration;
  // Application layer image file canvas (a render target)
 mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
```

```
  std::string                                                    m_outfbase;
  bool                                                         m_is_unittest;
  std::string                                          m_verify_image_path_base;
  mi::math::Vector<mi::Uint32, 2>                            m_heightfield_size;
  std::string                                             m_heightfield_infile;
  mi::math::Vector<mi::Float32, 2>                            m_heightfield_range;
  mi::math::Vector<mi::Sint32, 2>                             m_use_normal_recalc;
  mi::math::Vector<mi::Sint32, 2>                             m_set_normal_const;
  mi::neuraylib::Tag                                          m_heightfield_tag;
  mi::Sint32                                                        m_frame_idx;
  mi::math::Bbox<mi::Float32, 3>                                          m_roi;
};

mi::Sint32 Normal_recalculation::launch()
{
 m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configurat
  check_success(m_cluster_configuration.is_valid_interface());

  // create image canvas in application_layer
  m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
  check_success(m_image_file_canvas.is_valid_interface());

  // Verifying that local host has joined
  // This may fail when there is a license problem.
  check_success(is_local_host_joined(m_cluster_configuration.get()));

  mi::Sint32 exit_code = 0;
  create_scene();

  // render one frame
  {
    const mi::Sint32 fidx = get_next_frame_idx();
    const std::string outfbase     = get_output_file_name(m_outfbase, fidx);
   const std::string refimg_fname = get_output_file_name(m_verify_image_path_base, fidx);
    check_success(render_frame(outfbase, refimg_fname));
  }

  INFO_LOG << "use_normal_recalc: "  << m_use_normal_recalc
      << ", set_normal_const: " << m_set_normal_const;

  mi::Sint32 compute_id = 0;  // tell the compute task which compute this is.
  {
    const bool is_normal_recalc       = (m_use_normal_recalc.x == 1) ? true : false;
    const bool is_set_constant_normal = (m_set_normal_const.x  == 1) ? true : false;
    compute_heightfield(is_normal_recalc, is_set_constant_normal, compute_id);

    const mi::Sint32  fidx         = get_next_frame_idx();
    const std::string outfbase     = get_output_file_name(m_outfbase, fidx);
   const std::string refimg_fname = get_output_file_name(m_verify_image_path_base, fidx);
    check_success(render_frame(outfbase, refimg_fname));
  }

  // edit (restore) height values of the heightfield and render one frame
  ++compute_id;
  {
    const bool is_normal_recalc       = (m_use_normal_recalc.y == 1) ? true : false;
    const bool is_set_constant_normal = (m_set_normal_const.y  == 1) ? true : false;
```

```
    compute_heightfield(is_normal_recalc, is_set_constant_normal, compute_id);

    const mi::Sint32  fidx         = get_next_frame_idx();
    const std::string outfbase     = get_output_file_name(m_outfbase, fidx);
   const std::string refimg_fname = get_output_file_name(m_verify_image_path_base, fidx);
    check_success(render_frame(outfbase, refimg_fname));
  }

  return exit_code;
}

bool Normal_recalculation::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("dice::verbose", "2");
    sdict.insert("outfbase", "");
  }

  // Set own options
  m_outfbase               = sdict.get("outfbase", "");
  m_verify_image_path_base = sdict.get("verify_image_path_base", "");
 m_heightfield_size     = nv::index::app::get_vector_from_string<mi::Uint32,2> (sdict.get("heightf
  m_heightfield_infile     = sdict.get("heightfield_infile");
 m_heightfield_range    = nv::index::app::get_vector_from_string<mi::Float32,2>(sdict.get("height
 m_use_normal_recalc    = nv::index::app::get_vector_from_string<mi::Sint32,2> (sdict.get("use_nor
 m_set_normal_const     = nv::index::app::get_vector_from_string<mi::Sint32,2> (sdict.get("set_nor
  m_roi = nv::index::app::get_bbox_from_string<mi::Float32,3>(sdict.get("roi"));


  // Check necessary arguments for this example
  if (m_heightfield_infile.empty())
  {
    ERROR_LOG << "No -heightfield_infile option specified. Need -heightfield_file.";
    return false;
  }
  if (m_heightfield_size == mi::math::Vector<mi::Uint32, 2>(0, 0))
  {
    ERROR_LOG << "Invalid heightfield_size (0, 0). Need -heightfield_size.";
    return false;
  }
  if (m_heightfield_range == mi::math::Vector<mi::Float32,2>(0.0f, 0.0f))
  {
    ERROR_LOG << "Invalid heightfield_range (0, 0). Need -heightfield_range.";
    return false;
  }

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfbase = ["              + m_outfbase +
       "], verify_image_fname = [" + m_verify_image_path_base +
```

```
      "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            print out this message\n"
      << "        [-dice::verbose severity_level]\n"
      << "            verbose severity level (3 is info).\n"
      << "            (default: " << sdict.get("dice::verbose") << ")\n"

      << "        [-heightfield_size 'int int']\n"
      << "            Heightfield size heightfield.\n"
      << "            (default: [" << m_heightfield_size << "])\n"

      << "        [-heightfield_infile FILENAME]\n"
      << "            Heightfield input file path name.\n"
      << "            (default: [" << m_heightfield_infile << "])\n"

      << "        [-heightfield_range 'float float']\n"
      << "            Heightfield height range.\n"
      << "            (default: [" << m_heightfield_range << "])\n"

      << "        [-use_normal_recalc 'INT INT']\n"
      << "            Use normal recalculation of the height field when true.\n"
    << "        The first INT is for the first edit, the second INT is for the second edit.\n"
    << "        Use INT here since no BOOL vector. False when INT is 0, true when INT is 1.\n"
      << "            (default: [" << sdict.get("use_normal_recalc") << "])\n"

      << "        [-set_normal_const 'INT INT']\n"
      << "            Set normal a constant vector of the height field when true.\n"
    << "        The first INT is for the first edit, the second INT is for the second edit.\n"
    << "        This option is only effective when corresponding use_normal_recalc option is false.\n
    << "        Use INT here since no BOOL vector. False when INT is 0, true when INT is 1.\n"
      << "            (default: [" << sdict.get("set_normal_const") << "])\n"

      << "        [-outfbase FILENAME]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfbase << "])\n"

      << "        [-verify_image_path_base FILEBASE]\n"
    << "        when FILEBAS exists, verify the rendering image with adding frame number.\n"
      << "            (default: [" << m_verify_image_path_base << "])\n"

      << "\n"
      << "Test and debug option\n"
      << "\n"

      << "        [-unittest bool]\n"
      << "            when true, unit test mode.\n"
      << "            (default: [" << m_is_unittest << "])\n"

      << std::endl;
    exit(1);
```

```
  }

  return true;
}

class Example_height_fill_task :
  public mi::base::Interface_implement<nv::index::IRegular_heightfield_compute_task>
{
public:
  // Constructor
  //
  // You can use an arbitrary compute_id to specify the compute
  // task for test purpose. This is caller defined any value.
  //
  // \param[in] is_normal_recalc        normal recalculation when true
 // \param[in] is_set_constant_normal  set constant normal (0,1,0) to every normal value
  //                                    when is_normal_recalc is false.
  // \param[in] compute_id              compute id (for test)
  Example_height_fill_task(bool       is_normal_recalc,
              bool       is_set_constant_normal,
              mi::Sint32 compute_id)
    :
    m_is_normal_recalc(is_normal_recalc),
    m_is_set_constant_normal(is_set_constant_normal),
    m_compute_id(compute_id)
  {
    // empty
  }

  // implement IRegular_heightfield_compute_task

  virtual void get_region_of_interest_for_compute(
    mi::math::Bbox_struct<mi::Sint32, 2>& roi) const
  {
    // empty. This example applies whole region of interest.
  }

  // Compute height value with automatically recalculate the
  // normal value by the IndeX library.
  virtual bool compute(
    const mi::math::Bbox_struct<mi::Sint32, 2>& ij_patch_bbox,
    mi::Float32*                                elevation_values,
    mi::neuraylib::IDice_transaction*           dice_transaction) const
  {
   INFO_LOG << "Example_height_fill_task: called with automatic normal calculation. compute_id: "
        << m_compute_id;

    mi::math::Bbox<mi::Sint64, 2> ij_patch_bbox_s64;
    ij_patch_bbox_s64.min.x = static_cast< mi::Sint64 >(ij_patch_bbox.min.x);
    ij_patch_bbox_s64.min.y = static_cast< mi::Sint64 >(ij_patch_bbox.min.y);
    ij_patch_bbox_s64.max.x = static_cast< mi::Sint64 >(ij_patch_bbox.max.x);
    ij_patch_bbox_s64.max.y = static_cast< mi::Sint64 >(ij_patch_bbox.max.y);

    this->compute_elevation_value(ij_patch_bbox_s64, elevation_values);
    if (m_is_set_constant_normal)
    {
     WARN_LOG << "Turn off the is_set_constant_normal since is_normal_recalc is on.";
```

```
    }

    return true;
  }

  // Compute height value, but intentionally made the normal value
  // unchanged.
  //
  // If m_is_set_constant_normal is true, set the (0,1,0) normal
  // everywhere for test.
  virtual bool compute(
    const mi::math::Bbox_struct<mi::Sint32, 2>& ij_patch_bbox,
    mi::Float32*                                elevation_values,
    mi::math::Vector_struct<mi::Float32, 3>*    normal_values,
    mi::neuraylib::IDice_transaction*           dice_transaction) const
  {
    std::string normal_set_type = "";
    if (m_is_set_constant_normal)
    {
      normal_set_type = "set constant";
    }
    else
    {
      normal_set_type = "empty";
    }
    INFO_LOG << "Example_height_fill_task: called with (" << normal_set_type
        << ") user defined normal calculation. compute_id: " << m_compute_id;

    mi::math::Bbox<mi::Sint64, 2> ij_patch_bbox_s64;
    ij_patch_bbox_s64.min.x = static_cast< mi::Sint64 >(ij_patch_bbox.min.x);
    ij_patch_bbox_s64.min.y = static_cast< mi::Sint64 >(ij_patch_bbox.min.y);
    ij_patch_bbox_s64.max.x = static_cast< mi::Sint64 >(ij_patch_bbox.max.x);
    ij_patch_bbox_s64.max.y = static_cast< mi::Sint64 >(ij_patch_bbox.max.y);

    this->compute_elevation_value(ij_patch_bbox_s64, elevation_values);
    if (m_is_set_constant_normal)
    {
      this->compute_normal_value(ij_patch_bbox_s64, normal_values);
    }

    return true;
  }

  virtual bool user_defined_normal_computation() const
  {
    if (m_is_normal_recalc)
    {
      return false;
    }
    return true;
  }

private:
  // Compute elevation value
  //
  // \param[in]  ij_patch_bbox_s64 local patch bounding box
  // \param[out] elevation_values  (out) generated elevation value array
```

```
  void compute_elevation_value(
    const mi::math::Bbox<mi::Sint64, 2>& ij_patch_bbox_s64,
    mi::Float32 *                        elevation_values) const
  {
   INFO_LOG << "Compute_elevation_value (negation): ij patch bbox:" << ij_patch_bbox_s64;

    // Size of the heightfield patch in memory
   mi::math::Vector<mi::Sint64, 2> patch_raw_dim = ij_patch_bbox_s64.max - ij_patch_bbox_s64.min;
    inline_template_argument_ignore_fn(patch_raw_dim);
    assert((patch_raw_dim[0] > 0) && (patch_raw_dim[1] > 0));

    mi::math::Vector<mi::Sint64, 2> ij = ij_patch_bbox_s64.min;
    for(ij[0] = ij_patch_bbox_s64.min[0]; ij[0] < ij_patch_bbox_s64.max[0]; ++ij[0])
    {
     for(ij[1] = ij_patch_bbox_s64.min[1]; ij[1] < ij_patch_bbox_s64.max[1]; ++ij[1])
      {
        mi::Sint64 const ij_idx = get_heightfield_index(ij, ij_patch_bbox_s64);
        assert(ij_idx >= 0);
        assert(ij_idx <  patch_raw_dim[0] * patch_raw_dim[1]);

      if (!nv::index::IRegular_heightfield_patch::is_hole(elevation_values[ij_idx]))
        {
          elevation_values[ij_idx] = -1.0f * elevation_values[ij_idx];
        }
      }
    }
  }

  // Compute normal value ... set constant normal
  //
  // \param[in]  ij_patch_bbox_s64 local patch bounding box
  // \param[out] normal_data  (out) generated normal value array
 void compute_normal_value(const mi::math::Bbox<mi::Sint64, 2>&    ij_patch_bbox_s64,
              mi::math::Vector_struct<mi::Float32, 3>* normal_values) const
  {
    const mi::math::Vector<mi::Sint64, 2> patch_raw_dim =
      ij_patch_bbox_s64.max - ij_patch_bbox_s64.min;
    assert((patch_raw_dim[0] > 0) && (patch_raw_dim[1] > 0));
    mi::Sint64 const patch_raw_size = patch_raw_dim[0] * patch_raw_dim[1];

    const mi::math::Vector<mi::Float32, 3> normal_vec(0.0f, -1.0f, 0.0f);
    INFO_LOG << "Example_height_fill_task: set all normal to " << normal_vec;
    for(mi::Sint64 i = 0; i < patch_raw_size; i++)
    {
      normal_values[i] = normal_vec;
    }
  }

private:
  // recompute normal or not
  bool       m_is_normal_recalc;
  // set constant normal (0,1,0) for test.
  bool       m_is_set_constant_normal;
  // compute id for test
  mi::Sint32 m_compute_id;
};
```

```
class Example_heightfield_operation :
 public nv::index::Distributed_data_job<0xc1a29939,0x5853,0x4406,0x91,0x95,0xa6,0x9f,0xb3,0x28,0x
{
public:
  // Constructor
  //
  // \param[in] heightfield_tag        Tag of the heightfield scene element
  // \param[in] is_normal_recalc      switch for automatic normal recalculation or not
  // \param[in] is_set_constant_normal set constant normal (0,1,0) to every normal value
  //                                   when is_normal_recalc is false.
  // \param[in] compute_id             compute id for test (user defined value)
  Example_heightfield_operation(
    const mi::neuraylib::Tag&      heightfield_tag,
    bool                           is_normal_recalc,
    bool                           is_set_constant_normal,
    mi::Sint32                     compute_id)
   : m_heightfield_tag(heightfield_tag),
    m_is_normal_recalc(is_normal_recalc),
    m_is_set_constant_normal(is_set_constant_normal),
    m_compute_id(compute_id) {}

  // Default constructor
  Example_heightfield_operation()
   : m_heightfield_tag(mi::neuraylib::NULL_TAG),
    m_is_normal_recalc(true),
    m_is_set_constant_normal(false),
    m_compute_id(0) { }

 virtual nv::index::IDistributed_data_locality_query_mode* get_scheduling_mode() const
  {
   return new nv::index::Regular_heightfield_locality_query_mode(m_heightfield_tag, mi::math::Bbox<
  }

  virtual void execute_subset(
    mi::neuraylib::IDice_transaction*                  dice_transaction,
    const nv::index::IData_distribution*               data_distribution,
    nv::index::IData_subset_compute_task_processing*   data_subset_processing,
   mi::Size                             data_subset_index,    // not to confuse with subset data ID
    mi::Size                                           data_subset_count)
  {
   Example_height_fill_task heightfield_operation_task(m_is_normal_recalc, m_is_set_constant_normal
   data_subset_processing->execute_compute_task(&heightfield_operation_task, dice_transaction); //

    mi::base::Handle<nv::index::IRegular_heightfield_data_edit> compute_processing(
    data_subset_processing->get_interface<nv::index::IRegular_heightfield_data_edit>());
    if (compute_processing.is_valid_interface())
    {
      compute_processing->edit(&heightfield_operation_task, dice_transaction);
    }
  }

  virtual void execute_subset_remote(
    mi::neuraylib::ISerializer*                        serializer,
    mi::neuraylib::IDice_transaction*                  dice_transaction,
    const nv::index::IData_distribution*               data_distribution,
    nv::index::IData_subset_compute_task_processing*   data_subset_processing,
    mi::Size                                           data_subset_index,
```

```
    mi::Size                                          data_subset_count)
  {
   Example_height_fill_task heightfield_operation_task(m_is_normal_recalc, m_is_set_constant_normal
   data_subset_processing->execute_compute_task(&heightfield_operation_task, dice_transaction); //

    mi::base::Handle<nv::index::IRegular_heightfield_data_edit> compute_processing(
    data_subset_processing->get_interface<nv::index::IRegular_heightfield_data_edit>());
    if (compute_processing.is_valid_interface())
    {
      compute_processing->edit(&heightfield_operation_task, dice_transaction);
    }
  }

  virtual void receive_subset_result(
    mi::neuraylib::IDeserializer*           deserializer,
    mi::neuraylib::IDice_transaction*       dice_transaction,
    const nv::index::IData_distribution*    data_distribution,
    mi::Size                                data_subset_index,
    mi::Size                                data_subset_count) {}

  virtual void serialize(mi::neuraylib::ISerializer *serializer) const
  {
    serializer->write(&m_heightfield_tag.id,     1);
    serializer->write(&m_is_normal_recalc,       1);
    serializer->write(&m_is_set_constant_normal, 1);
    serializer->write(&m_compute_id,             1);
  }

  virtual void deserialize(mi::neuraylib::IDeserializer* deserializer)
  {
    deserializer->read(&m_heightfield_tag.id,     1);
    deserializer->read(&m_is_normal_recalc,       1);
    deserializer->read(&m_is_set_constant_normal, 1);
    deserializer->read(&m_compute_id,             1);
  }

private:
  mi::neuraylib::Tag      m_heightfield_tag;
  bool                    m_is_normal_recalc;
  bool                    m_is_set_constant_normal;
  mi::Sint32              m_compute_id;
};

void Normal_recalculation::create_scene()
{
 mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
  check_success(dice_transaction.is_valid_interface());
  {
    m_session_tag = m_index_session->create_session(dice_transaction.get());
    check_success(m_session_tag.is_valid());
    mi::base::Handle<const nv::index::ISession> session(
      dice_transaction->access<nv::index::ISession>(m_session_tag));
    check_success(session.is_valid_interface());

    // Access (edit mode) the scene instance from the database.
    mi::base::Handle<nv::index::IScene> scene_edit(
      dice_transaction->edit<nv::index::IScene>(session->get_scene()));
```

```
  check_success(scene_edit.is_valid_interface());

  nv::index::app::String_dict heightfield_opt;
  heightfield_opt.insert("args::type",          "heightfield");
 heightfield_opt.insert("args::importer",    "nv::index::plugin::legacy_importer.Raw_heightfield
  heightfield_opt.insert("args::input_file",    m_heightfield_infile);
 heightfield_opt.insert("args::size",       nv::index::app::to_string(m_heightfield_size));
  heightfield_opt.insert("args::range",         "0.1 1000");
  heightfield_opt.insert("args::cache",         "false");
  heightfield_opt.insert("args::serial_access", "false");
  heightfield_opt.insert("args::stats",         "false");

  nv::index::IDistributed_data_import_callback* importer_callback =
    get_importer_from_application_layer(
      get_application_layer_interface(),
      "nv::index::plugin::legacy_importer.Raw_heightfield_data_importer",
      heightfield_opt);

  const mi::Float32 rotate_k = 0.0f;
  const mi::math::Vector<mi::Float32, 3> translate(0.0f, 0.0f, 0.0f);
  const mi::math::Vector<mi::Float32, 3> scale(    1.0f, 1.0f, 1.0f);

  // Create a heightfield scene element via the raw heightfield importer
  mi::base::Handle<nv::index::IRegular_heightfield> heightfield(
    scene_edit->create_regular_heightfield(
      scale, rotate_k, translate,
      m_heightfield_size,
      m_heightfield_range,
      importer_callback,
      dice_transaction.get()));
  check_success(heightfield.is_valid_interface());

  // ... and store the heightfield scene element in the distributed database
 check_success(!m_heightfield_tag.is_valid()); // must not yet created before this initialization.
  m_heightfield_tag =
    dice_transaction->store_for_reference_counting(heightfield.get());
  check_success(m_heightfield_tag.is_valid()); // Now this should exist.

  // Create static group node for heightfield data
  mi::base::Handle<nv::index::IStatic_scene_group> static_group_node(
    scene_edit->create_scene_group<nv::index::IStatic_scene_group>());
  check_success(static_group_node.is_valid_interface());

  // Added a light and a material to the static group node
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
    headlight->set_intensity(color_intensity);
   headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
  const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headlig
    check_success(headlight_tag.is_valid());
    static_group_node->append(headlight_tag, dice_transaction.get());

    // Material for the heightfield
```

```
    mi::base::Handle<nv::index::IPhong_gl> phong_1(scene_edit->create_attribute<nv::index::IPhong_
      check_success(phong_1.is_valid_interface());

      // Define a more interesting greenish material
      phong_1->set_ambient(mi::math::Color(0.0f, 0.3f, 0.0f, 0.3f));
      phong_1->set_diffuse(mi::math::Color(0.0f, 0.8f, 0.2f, 0.3f));
      phong_1->set_specular(mi::math::Color(0.6f));
      phong_1->set_shininess(100.0f);

      const mi::neuraylib::Tag phong_1_tag
        = dice_transaction->store_for_reference_counting(phong_1.get());
      check_success(phong_1_tag.is_valid());
      static_group_node->append(phong_1_tag, dice_transaction.get());
    }

    // Append the heightfield to the scene group
    static_group_node->append(m_heightfield_tag, dice_transaction.get());
    mi::neuraylib::Tag static_group_node_tag =
      dice_transaction->store_for_reference_counting(static_group_node.get());
    check_success(static_group_node_tag.is_valid());

    // Append the static scene group to the scene.
    scene_edit->append(static_group_node_tag, dice_transaction.get());

    std::stringstream sstr;
    sstr << "Created a heightfield: size = "
      << m_heightfield_size << ", tag = " << m_heightfield_tag.id;
    INFO_LOG << sstr.str();

    // Define a region of interest
    scene_edit->set_clipped_bounding_box(m_roi);

    // Optionally adjust the scene's coordinate system
    mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
      1.0f,  0.0f,  0.0f,  0.0f,
      0.0f,  1.0f,  0.0f,  0.0f,
      0.0f,  0.0f, -1.0f,  0.0f, // adjust for coordinate system
      0.0f,  0.0f,  0.0f,  1.0f
      );

    scene_edit->set_transform_matrix(transform_mat);

    // Create a camera and set it to the scene.
    mi::base::Handle< nv::index::IPerspective_camera > cam(
      scene_edit->create_camera<nv::index::IPerspective_camera>());
    check_success(cam.is_valid_interface());
    setup_camera(cam.get());

    const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
    check_success(camera_tag.is_valid());

    scene_edit->set_camera(camera_tag);

    const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
    m_image_file_canvas->set_resolution(buffer_resolution);
  }
  dice_transaction->commit();
```

```
}

void Normal_recalculation::setup_camera(
  nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector<mi::Float32, 3> from(2959.61f, 1993.47f, -1653.08f);
  mi::math::Vector<mi::Float32, 3> dir (-0.5891f, -0.3783f,  0.7141f);
  mi::math::Vector<mi::Float32, 3> up  (-0.6795f, -0.2464f, -0.6911f);
  dir.normalize();

  cam->set(from, dir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(10000.0f);
}

bool Normal_recalculation::render_frame(const std::string& output_fname,
                    const std::string& verify_fname)
{
  bool success = true;

  // Rendering to a file by means of the file canvas
 m_image_file_canvas->set_rgba_file_name(output_fname.c_str()); // No output if empty string

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
  {
    m_index_session->update(m_session_tag, dice_transaction.get());
    mi::base::Handle<nv::index::IFrame_results> frame_results(
      m_index_rendering->render(
        m_session_tag,
        m_image_file_canvas.get(),
        dice_transaction.get()));
    check_success(frame_results.is_valid_interface());
  }
  dice_transaction->commit();

  if (!output_fname.empty())
  {
    INFO_LOG << "save the rendered image to: " << output_fname;
  }

  // verify the generated frame
  if (!verify_fname.empty())
  {
    if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), verify_fname, get_options())))
    {
      success = false;
    }
  }
```

```
  // increment the frame index.
  ++m_frame_idx;

  return success;
}

void Normal_recalculation::compute_heightfield(
  bool       is_normal_recalc,
  bool       is_set_constant_normal,
  mi::Sint32 compute_id)
{
  check_success(m_session_tag.is_valid());
  check_success(m_heightfield_tag.is_valid());

  // DiCE database access
 mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
  check_success(dice_transaction.is_valid_interface());
  {
    // Choose a distributed computing algorithm for heightfield creation
   mi::base::Handle<nv::index::IDistributed_data_job> distributed_compute_algorithm;
    // Edit the heightfield
    {
      INFO_LOG << "Normal recalculation: " << (is_normal_recalc ? "on" : "off");
      if (!is_normal_recalc)
      {
       INFO_LOG << "Set constant normal: " << (is_set_constant_normal ? "on" : "off");
      }

      mi::base::Handle< nv::index::IDistributed_data_job > eh_op(
        new Example_heightfield_operation(m_heightfield_tag,
                          is_normal_recalc,
                          is_set_constant_normal,
                          compute_id));
      distributed_compute_algorithm.swap(eh_op);
    }
   mi::base::Handle<const nv::index::ISession> session(dice_transaction->access<const nv::index::IS
    check_success(session.is_valid_interface());

    const mi::neuraylib::Tag dist_layout_tag = session->get_distribution_layout();
   mi::base::Handle<const nv::index::IData_distribution> distribution_layout(dice_transaction->acc

   distribution_layout->create_scheduler()->execute(distributed_compute_algorithm.get(), dice_tran

    INFO_LOG << "Done heightfield computation.";

  }
  dice_transaction->commit();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  // DiCE options
  sdict.insert("dice::verbose", "3");
  sdict.insert("dice::network::mode", "OFF");
  sdict.insert("dice::network::multicast_address", "224.1.3.2");
```

```
  // Test specific options
  sdict.insert("heightfield_name", "heightfield_1");
  sdict.insert("heightfield_infile", "");
  sdict.insert("heightfield_size",  "0 0");
  sdict.insert("heightfield_range", "0 0");
  sdict.insert("roi", "0 0 -1024 2040 2040 1024");
  sdict.insert("use_normal_recalc", "0 1");
  sdict.insert("set_normal_const", "1 0");
  sdict.insert("outfbase", "frame_normal_recalculation");    // output file base name

  // Unit test specific options
  sdict.insert("unittest", "0");                              // unit test mode
 sdict.insert("verify_image_path_base", "");           // reference image file base name
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  // IndeX settings
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // Application_layer settings
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io");
  // plugin: legacy_importer
  sdict.insert("index::app::plugins::legacy_importer::enabled", "true");

  // Initialize application
  Normal_recalculation normal_recalculation;
  normal_recalculation.initialize(argc, argv, sdict);
  check_success(normal_recalculation.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = normal_recalculation.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.28 render_frame.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/iindex.h>
#include <nv/index/isession.h>
#include <nv/index/iscene.h>
#include <nv/index/icamera.h>

#include <iostream>
#include <sstream>

#include <nv/index/app/application_layer_common_utility.h>
#include <nv/index/app/forwarding_logger.h>
#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>

class Render_frame:
    public nv::index::app::Index_connect
{
public:
  Render_frame()
    :
    Index_connect(),
    m_test_set_roi(true)
  {
    // INFO_LOG << "DEBUG: Render_frame() ctor";
  }

  virtual ~Render_frame()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Render_frame() dtor";
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  // override
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
```

```
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // Setup camera to see this example scene
  // \param[in] cam    a camera
  void setup_camera(nv::index::IPerspective_camera* cam) const;
  // set up a scene for this example.
  //   - Set region of interest to the scene
  //   - Set default transformation matrix to the scene
  //   - Set the current active camera to the scene
  void setup_render_frame_scene(
    nv::index::IScene*                              scene_edit,
    const mi::math::Bbox_struct< mi::Float32, 3 >& xyz_roi_st,
    const mi::neuraylib::Tag&                       camera_tag,
    mi::neuraylib::IDice_transaction*               dice_transaction) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                    m_is_unittest;
  bool                                                                   m_test_set_roi;
};

mi::Sint32 Render_frame::launch()
{
  mi::Sint32 exit_code = 0;
  {
   m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configura
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    // This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));

    // Setup scene
    {
      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());
      {
        // Setup session information
```

```
      m_session_tag = m_index_session->create_session(dice_transaction.get());
      check_success(m_session_tag.is_valid());
      mi::base::Handle< nv::index::ISession const > session(
        dice_transaction->access< nv::index::ISession const >(
          m_session_tag));
      check_success(session.is_valid_interface());

      mi::base::Handle< nv::index::IScene > scene_edit(
        dice_transaction->edit<nv::index::IScene>(session->get_scene()));
      check_success(scene_edit.is_valid_interface());

      //------------------------------------------------------------------------
      // Scene setup: Put nothing in the scene in this example
      //------------------------------------------------------------------------

      // Create and edit a camera. Data distribution is based on
      // the camera. (Because only visible massive data are
      // considered)
      mi::base::Handle< nv::index::IPerspective_camera > cam(
        scene_edit->create_camera<nv::index::IPerspective_camera>());
      check_success(cam.is_valid_interface());
      setup_camera(cam.get());
      const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
      check_success(camera_tag.is_valid());

      const mi::math::Vector<mi::Uint32, 2> buffer_resolution(512, 512);
      m_image_file_canvas->set_resolution(buffer_resolution);

      // Set up the scene
      if(m_test_set_roi)
      {
        mi::math::Bbox_struct< mi::Float32, 3 > const xyz_roi_st = {
          { 0.0f, 0.0f, 0.0f, },
          { 1.0f, 1.0f, 1.0f, },
        };
        setup_render_frame_scene(scene_edit.get(), xyz_roi_st, camera_tag,
                    dice_transaction.get());
      }
      else
      {
     ERROR_LOG << "The following error message about an invalid region of interest is expected";
        // test case: not valid ROI set
        mi::math::Bbox_struct< mi::Float32, 3 > const zero_xyz_roi_st = {
          { 0.0f, 0.0f, 0.0f, },
          { 0.0f, 0.0f, 0.0f, },
        };
        setup_render_frame_scene(scene_edit.get(), zero_xyz_roi_st, camera_tag,
                    dice_transaction.get());
      }
    }
    dice_transaction->commit();
  }

  // Rendering call
  {
    // Set up the canvas according to the camera setup. It is
    // still valid if output_fname.empty() == true (no output
```

```
      // file)
      const mi::Sint32  frame_idx = 0;
      const std::string output_fname = get_output_file_name(m_outfname, frame_idx);
      m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

      mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
        m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
      check_success(dice_transaction.is_valid_interface());

      m_index_session->update(m_session_tag, dice_transaction.get());

      mi::base::Handle<nv::index::IFrame_results> frame_results(
        m_index_rendering->render(
          m_session_tag,
          m_image_file_canvas.get(),
          dice_transaction.get()));
      check_success(frame_results.is_valid_interface());

      dice_transaction->commit();
    }
  }
  return exit_code;
}

bool Render_frame::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname     = sdict.get("outfname", "");
  m_test_set_roi = nv::index::app::get_bool(sdict.get("test_set_roi", "false"));

  info_cout(std::string("running ") + com_name, sdict);
  info_cout(std::string("outfname = [") + m_outfname +
       "], dice::verbose = "       + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if(sdict.is_defined("h"))
  {
    std::cout
      << "Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"
      << "        [-outfname string]\n"
```

```
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
      << "        [-test_set_roi bool]\n"
      << "            when false, no ROI set.\n"
      << "            (default: [" << m_test_set_roi << "])\n"
      << std::endl;
    exit(1);
  }
  return true;
}

void Render_frame::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene
  mi::math::Vector< mi::Float32, 3 > const from(-3.67f, 24.35f, -40.85f);
  mi::math::Vector< mi::Float32, 3 > const to  ( 9.5, 9.5, -9.5);
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
  cam->set_focal(0.03f);
  cam->set_clip_min(2.0f);
  cam->set_clip_max(420.0f);
}

void Render_frame::setup_render_frame_scene(
  nv::index::IScene*                              scene_edit,
  const mi::math::Bbox_struct< mi::Float32, 3 >& xyz_roi_st,
  const mi::neuraylib::Tag&                       camera_tag,
  mi::neuraylib::IDice_transaction*               dice_transaction) const
{
  check_success(dice_transaction != 0);

  // set the region of interest
  const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
  if(xyz_roi.is_volume())
  {
    scene_edit->set_clipped_bounding_box(xyz_roi_st);
  }
  else
  {
    INFO_LOG << "ROI has no volume, not set the ROI.";
  }

  // Set the scene transformation matrix.
  // only change the coordinate system
  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    1.0f,  0.0f,  0.0f,  0.0f,
    0.0f,  1.0f,  0.0f,  0.0f,
    0.0f,  0.0f, -1.0f,  0.0f,
    0.0f,  0.0f,  0.0f,  1.0f
```

```
    );
  scene_edit->set_transform_matrix(transform_mat);

  // Set the current camera to the scene.
  check_success(camera_tag.is_valid());
  scene_edit->set_camera(camera_tag);
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");             // log level
  sdict.insert("outfname", "frame_render_frame"); // output file base name
  sdict.insert("unittest", "0");                  // default mode
  sdict.insert("test_set_roi", "1");              // ROI set/no-set test
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  // DiCE settings
  sdict.insert("dice::network::mode", "OFF");

  // IndeX settings
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // Application_layer settings
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io");

  // Initialize application
  Render_frame render_frame;
  render_frame.initialize(argc, argv, sdict);
  check_success(render_frame.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = render_frame.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.29 scene_description_attribute.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>

#include "utility/example_shared.h"

#include <nv/index/icamera.h>
#include <nv/index/iconfig_settings.h>
#include <nv/index/idepth_offset.h>
#include <nv/index/idepth_test.h>
#include <nv/index/iindex.h>
#include <nv/index/ilabel.h>
#include <nv/index/ilight.h>
#include <nv/index/iline_set.h>
#include <nv/index/imaterial.h>
#include <nv/index/iplane.h>
#include <nv/index/ipoint_set.h>
#include <nv/index/ipolygon.h>
#include <nv/index/irendering_order.h>
#include <nv/index/iscene.h>
#include <nv/index/iscene_group.h>
#include <nv/index/isession.h>
#include <nv/index/isphere.h>
#include <nv/index/itexture_filter_mode.h>

#include "constant_color_mapping.h"

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>
#include <nv/index/app/time_functions.h>

#include "utility/app_rendering_context.h"
#include "utility/canvas_utility.h"

#include <algorithm>
#include <iostream>
#include <sstream>

class Scene_description_attribute:
    public nv::index::app::Index_connect
{
public:
  Scene_description_attribute()
    :
    Index_connect()
  {
    // INFO_LOG << "DEBUG: Scene_description_attribute() ctor";
  }

  virtual ~Scene_description_attribute()
  {
    // Note: Index_connect::~Index_connect() will be called after here.
    // INFO_LOG << "DEBUG: ~Scene_description_attribute() dtor";
```

```
  }

  // launch application
  mi::Sint32 launch();

protected:
  virtual bool evaluate_options(nv::index::app::String_dict& sdict) CPP11_OVERRIDE;
  virtual bool register_serializable_classes(
    mi::neuraylib::IDice_configuration* configuration_interface,
    nv::index::app::String_dict&        options)
  {
    bool is_registered = false;
   is_registered = configuration_interface->register_serializable_class<Distributed_compute_constar
    check_success(is_registered);
    return true;
  }
  virtual bool initialize_networking(
    mi::neuraylib::INetwork_configuration* network_configuration,
    nv::index::app::String_dict&           options) CPP11_OVERRIDE
  {
    check_success(network_configuration != 0);

    check_success(options.is_defined("unittest"));
    const bool is_unittest = nv::index::app::get_bool(options.get("unittest"));
    if (is_unittest)
    {
      info_cout("NETWORK: disabled networking mode.", options);
    network_configuration->set_mode(mi::neuraylib::INetwork_configuration::MODE_OFF);
      return true;
    }

    return initialize_networking_as_default_udp(network_configuration, options);
  }

private:
  // create a material to a group
  void create_append_material_to_group(
    nv::index::IScene*                   scene_edit,
    nv::index::ITransformed_scene_group* group_node,
    const mi::math::Color&               ambient_color,
    const mi::math::Color&               diffuse_color,
    const mi::math::Color&               specular_color,
    const mi::Float32                    shiness,
    mi::neuraylib::IDice_transaction*    dice_transaction);
  // create a plane and add to a group
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node parent group node of the depth offset attribute
  // \param[in] plane_point  plane corner point
  // \param[in] plane_normal plane normal vector
  // \param[in] plane_up     plane normal vector
  // \param[in] plane_extent plane extent (size of the plane)
  // \param[in] col          plane color
  // \param[in] dice_transaction db transaction
  void create_append_plane_to_group(
    nv::index::IScene*                     scene_edit,
    nv::index::ITransformed_scene_group*   group_node,
```

```
    const mi::math::Vector<mi::Float32, 3>& plane_point,
    const mi::math::Vector<mi::Float32, 3>& plane_normal,
    const mi::math::Vector<mi::Float32, 3>& plane_up,
    const mi::math::Vector<mi::Float32, 2>& plane_extent,
    const mi::math::Color&                  col,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // create a depth offset attribute and add to a group
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node parent group node of the depth offset attribute
  // \param[in] depth_offset depth offset value
 // \param[in] is_depth_offset_attribute_on true the attribute node enabled, false otherwise
  // \param[in] dice_transaction db transaction
  void create_append_depth_offset_attribute_to_group(
    nv::index::IScene*                   scene_edit,
    nv::index::ITransformed_scene_group* group_node,
    mi::Float32                          depth_offset,
    bool                                 is_depth_offset_attribute_on,
    mi::neuraylib::IDice_transaction*    dice_transaction);
  // create a depth test attribute and add to a group
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node parent group node of the depth offset attribute
  // \param[in] depth_test_operator depth test operator
 // \param[in] is_depth_test_attribute_on true the attribute node enabled, false otherwise
  // \param[in] dice_transaction db transaction
  void create_append_depth_test_attribute_to_group(
    nv::index::IScene*                      scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
    nv::index::IDepth_test::Depth_test_mode depth_test_operator,
    bool                                    is_depth_test_attribute_on,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // create a rendering order attribute and add to a group
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node parent group node of the depth offset attribute
  // \param[in] priority_offset rendering priority offset
 // \param[in] is_rendering_order_attrib_on true the attribute node enabled, false otherwise
  // \param[in] dice_transaction db transaction
  void create_append_rendering_order_test_attribute_to_group(
    nv::index::IScene*                   scene_edit,
    nv::index::ITransformed_scene_group* group_node,
    mi::Uint32                           priority_offset,
    bool                                 is_rendering_order_attrib_on,
    mi::neuraylib::IDice_transaction*    dice_transaction);
  // create a line and add to a group
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node parent group node of the depth offset attribute
  // \param[in] v0         line segment vertex 0
  // \param[in] v1         line segment vertex 1
  // \param[in] col        line segment color
  // \param[in] width      line width
  // \param[in] dice_transaction db transaction
  void create_append_line_to_group(
    nv::index::IScene*                      scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
```

```
    const mi::math::Vector<mi::Float32, 3>& v0,
    const mi::math::Vector<mi::Float32, 3>& v1,
    const mi::math::Color&                  col,
    const mi::Float32                       width,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // depth offset attribute test: labels
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node   parent group node
  // \param[in] label_point  label's corner position in the object space
  // \param[in] label_str    label contents string
  // \param[in] label_height label height in the object space
  // \param[in] label_width  label width in the object space
  // \param[in] dice_transaction db transaction
  // \return created group node tag
  void create_append_label_to_group(
    nv::index::IScene*                      scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
    const mi::math::Vector<mi::Float32, 3>& label_point,
    const std::string&                      label_str,
    const mi::Float32                       label_height,
    const mi::Float32                       label_width,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // depth offset attribute test: single point creation
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] group_node parent group node
  // \param[in] point_pos  point center position
  // \param[in] point_col  point color
  // \param[in] point_rad  point radius
  // \param[in] dice_transaction db transaction
  void create_append_point_to_group(
    nv::index::IScene*                      scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
    const mi::math::Vector<mi::Float32, 3>& point_pos,
    const mi::math::Color&                  point_col,
    mi::Float32                             point_rad,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // depth offset attribute test
  //
  // \param[in] scene_edit the editable IndeX scene
  // \param[in] dice_transaction db transaction
  // \return the local root group node of this scene
  mi::neuraylib::Tag create_depth_offset_attribute_test_scene(
    nv::index::IScene*                  scene_edit,
    mi::neuraylib::IDice_transaction*   dice_transaction);
  // create and append a sphere
  //
  // Sphere's color will be set by a material.
  //
  // \param[in] scene_edit scene to be edited
  // \param[in] group_node a parent group node
  // \param[in] diffuse_color sphere's diffuse color component
  // \param[in] center_pos center position of the sphere
  // \param[in] radius sphere radius
  // \param[in] dice_transaction db transaction
  void create_append_sphere(
```

```
    nv::index::IScene*                      scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
    const mi::math::Color&                  diffuse_color,
    const mi::math::Vector<mi::Float32, 3>& center_pos,
    const mi::Float32                       radius,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // create z-test scene
  //
  // \param[in] scene_edit scene to be edited
  // \param[in] dice_transaction db transaction
  // \return the local root group node of this scene
  mi::neuraylib::Tag create_z_test_attribute_test_scene(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction);
  // create and append a polygon
  //
  // \param[in] scene_edit scene to be edited
  // \param[in] group_node a parent group node
  // \param[in] diffuse_color sphere's diffuse color component
  // \param[in] vertex_pos_ary vertex position array
  // \param[in] vertex_count   number of vertices of the polygon
  // \param[in] polygon_center polygon center of 3D position
  // \param[in] dice_transaction db transaction
  void create_append_polygon(
    nv::index::IScene*                      scene_edit,
    nv::index::ITransformed_scene_group*    group_node,
    const mi::math::Color&                  diffuse_color,
    const mi::math::Vector<mi::Float32, 2>  vertex_pos_ary[],
    mi::Sint32                              vertex_count,
    const mi::math::Vector<mi::Float32, 3>& polygon_center,
    mi::neuraylib::IDice_transaction*       dice_transaction);
  // create rendering order (painter's algorithm) test scene
  //
  // \param[in] scene_edit scene to be edited
  // \param[in] dice_transaction db transaction
  // \return the local root group node of this scene
  mi::neuraylib::Tag create_rendering_order_attribute_test_scene(
    nv::index::IScene*                  scene_edit,
    mi::neuraylib::IDice_transaction*   dice_transaction);
  // create test scene
  void create_scene(
    nv::index::IScene*                scene_edit,
    mi::neuraylib::IDice_transaction* dice_transaction);
  // setup camera to see this example scene
  //
  // \param[in] cam    a camera
  void setup_camera(
    nv::index::IPerspective_camera* cam) const;
  // setup a far away camera for a extreme transformation handling test
  //
  // See Bugzilla 11567
  //
  // \param[in] cam    a camera
  void setup_extreme_transformed_camera(nv::index::IPerspective_camera* cam) const;
  // setup a large scene transform for the extreme
  // transformation handling test
  //
```

```
  // This value is based on Bugzilla 11567
  // Attachment: Two complete screen dump with 188853_8578 build case
  mi::math::Matrix<mi::Float32, 4, 4> get_extreme_transformed_matrix() const;
  // render a frame
  //
  // \param[in] index_connect access to the NVIDIA IndeX library
  // \param[in] arc              application rendering context
  // \param[in] output_fname     output rendering image filename
  // \return performance values
  nv::index::IFrame_results* render_frame(const std::string& output_fname) const;

  // This session tag
  mi::neuraylib::Tag                                                    m_session_tag;
  // NVIDIA IndeX cluster configuration
  mi::base::Handle<nv::index::ICluster_configuration>                      m_cluster_configuration;
  // Application layer image file canvas (a render target)
  mi::base::Handle<nv::index::app::canvas_infrastructure::IIndex_image_file_canvas> m_image_file_ca
  // Create_icons options
  std::string                                                              m_outfname;
  bool                                                                  m_is_unittest;
  std::string                                                        m_verify_image_fname;
  std::string                                                               m_font_fpath;
  bool                                                            m_is_depth_offset_node_on;
  bool                                                                m_is_z_test_attrib_on;
  bool                                                           m_is_rendering_order_attrib_on;
  mi::Sint32                                                           m_global_z_test_mode;
  bool                                                                 m_is_large_translate;
  const mi::Float32                                                 m_workaround_shift = 1.0f;
};

mi::Sint32 Scene_description_attribute::launch()
{
  mi::Sint32 exit_code = 0;

  {
   m_cluster_configuration = get_index_interface()->get_api_component<nv::index::ICluster_configura
    check_success(m_cluster_configuration.is_valid_interface());

    // create image canvas in application_layer
   m_image_file_canvas = create_image_file_canvas(get_application_layer_interface());
    check_success(m_image_file_canvas.is_valid_interface());

    // Verifying that local host has joined
    // This may fail when there is a license problem.
    check_success(is_local_host_joined(m_cluster_configuration.get()));

    // Scene setup
    {
    mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(create_transaction());
      check_success(dice_transaction.is_valid_interface());
      {
        // Setup session information
        m_session_tag =
          m_index_session->create_session(dice_transaction.get());
        check_success(m_session_tag.is_valid());

        mi::base::Handle<const nv::index::ISession> session(
```

```
  dice_transaction->access<nv::index::ISession>(
    m_session_tag));
check_success(session.is_valid_interface());

mi::base::Handle< nv::index::IScene > scene_edit(
  dice_transaction->edit< nv::index::IScene >(session->get_scene()));
check_success(scene_edit.is_valid_interface());

// Set global depth test mode for raster object and 3D object
{
  INFO_LOG << "global_z_test_mode = " << m_global_z_test_mode;
  mi::base::Handle<nv::index::IConfig_settings> config_settings(
dice_transaction->edit<nv::index::IConfig_settings>(session->get_config()));
  check_success(config_settings.is_valid_interface());

  config_settings->set_depth_test(
static_cast<nv::index::IDepth_test::Depth_test_mode>(m_global_z_test_mode));
  INFO_LOG << "Current global depth test mode = "
      << static_cast<mi::Sint32>(config_settings->get_depth_test());
}

//------------------------------------------------------------------------
// Scene setup: add textured planes
//------------------------------------------------------------------------
create_scene(scene_edit.get(), dice_transaction.get());

mi::base::Handle< nv::index::IPerspective_camera > cam(
  scene_edit->create_camera<nv::index::IPerspective_camera>());
check_success(cam.is_valid_interface());
setup_camera(cam.get());

if (m_is_large_translate)
{
  INFO_LOG << "large translate test mode.";
  setup_extreme_transformed_camera(cam.get());
}
else
{
  setup_camera(cam.get());
}
const mi::neuraylib::Tag camera_tag = dice_transaction->store(cam.get());
check_success(camera_tag.is_valid());

// Set up the scene and define the region of interest
const mi::math::Bbox_struct<mi::Float32, 3> xyz_roi_st = {
  { -1000.0f, -1000.0f, -500.0f, },
  {  1000.0f,  1000.0f,  500.0f, },
};

// set the region of interest
const mi::math::Bbox< mi::Float32, 3 > xyz_roi(xyz_roi_st);
check_success(xyz_roi.is_volume());
scene_edit->set_clipped_bounding_box(xyz_roi_st);

// Set the scene global transformation matrix.
// only change the coordinate system
mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
```

```
          1.0f,  0.0f,  0.0f,  0.0f,
          0.0f,  1.0f,  0.0f,  0.0f,
          0.0f,  0.0f, -1.0f,  0.0f,
          0.0f,  0.0f,  0.0f,  1.0f
          );

        mi::math::Vector<mi::Uint32, 2> buffer_resolution(1024, 1024);
        if (m_is_large_translate)
        {
          transform_mat = get_extreme_transformed_matrix();
          buffer_resolution = mi::math::Vector<mi::Uint32, 2>(1602, 934);
        }
        scene_edit->set_transform_matrix(transform_mat);
        m_image_file_canvas->set_resolution(buffer_resolution);

        // Set the current camera to the scene.
        check_success(camera_tag.is_valid());
        scene_edit->set_camera(camera_tag);
      }
      // Finalize scene setup transaction
      dice_transaction->commit();
    }

    // Render the frame to a file
    {
      const mi::Sint32  frame_idx = 0;
      const std::string fname = get_output_file_name(m_outfname, frame_idx);
     mi::base::Handle<nv::index::IFrame_results> frame_results(render_frame(fname));
    const mi::base::Handle<nv::index::IError_set> err_set(frame_results->get_error_set());
      if (err_set->any_errors())
      {
        std::ostringstream os;
        const mi::Uint32 nb_err = err_set->get_nb_errors();
        for (mi::Uint32 e = 0; e < nb_err; ++e)
        {
          if (e != 0) os << '\n';
          const mi::base::Handle<nv::index::IError> err(err_set->get_error(e));
          os << err->get_error_string();
        }

     ERROR_LOG << "IIndex_rendering rendering call failed with the following error(s): " << '\n'
              << os.str();
        exit_code = 1;
      }

      // verify the generated frame
      if (!(verify_canvas_result(get_application_layer_interface(),
                  m_image_file_canvas.get(), m_verify_image_fname, get_options())))
      {
        exit_code = 1;
      }
    }
  }

  return exit_code;
}
```

```
bool Scene_description_attribute::evaluate_options(nv::index::app::String_dict& sdict)
{
  const std::string com_name = sdict.get("command:", "<unknown_command>");
  m_is_unittest = nv::index::app::get_bool(sdict.get("unittest", "false"));

  if (m_is_unittest)
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("outfname", "");  // turn off file output in the unit test mode
    sdict.insert("dice::verbose", "2");
  }

  m_outfname                     = sdict.get("outfname");
  m_verify_image_fname           = sdict.get("verify_image_fname");
  m_font_fpath                   = sdict.get("font_fpath");
 m_is_depth_offset_node_on    = nv::index::app::get_bool(sdict.get("is_depth_offset_node_on", "fal
 m_is_z_test_attrib_on       = nv::index::app::get_bool(sdict.get("is_z_test_attrib_on", "false"))
 m_is_rendering_order_attrib_on = nv::index::app::get_bool(sdict.get("is_rendering_order_attrib_or
 m_global_z_test_mode       = nv::index::app::get_sint32(sdict.get("global_z_test_mode"));
 m_is_large_translate       = nv::index::app::get_bool(sdict.get("is_large_translate", "false"));

  info_cout(std::string("running ") + com_name, sdict);
  info_cout("outfname = ["        + m_outfname +
       "], dice::verbose = " + sdict.get("dice::verbose"), sdict);

  // print help and exit if -h
  if (sdict.is_defined("h"))
  {
    std::cout
      << "info: Usage: " << com_name << " [option]\n"
      << "Option: [-h]\n"
      << "            printout this message\n"
      << "        [-dice::verbose severity_level]\n"
    << "        verbose severity level (3 is info.). (default: " + sdict.get("dice::verbose")
      << ")\n"

      << "        [-font_fpath FONT_FILE_PATH]\n"
      << "            font file path. (default: " << m_font_fpath << ")\n"

      << "        [-is_depth_offset_node_on bool]\n"
      << "            on/off depth offset value on when true."
      << "(default: " << m_is_depth_offset_node_on << ")\n"

      << "        [-is_z_test_attrib_on bool]\n"
      << "            on/off z-test attribute on when true."
      << "(default: " << m_is_z_test_attrib_on << ")\n"

      << "        [-is_rendering_order_attrib_on bool]\n"
      << "            on/off rendering order attribute on when true."
      << "(default: " << m_is_rendering_order_attrib_on << ")\n"

      << "        [-global_z_test_mode depth_test_mode]\n"
      << "            set depth_test_mode, see IConfig_settings documentation."
      << "(default: " << m_global_z_test_mode << ")\n"
```

```
      << "        [-is_large_translate bool]\n"
      << "            on/off large translation mode."
      << "(default: " << m_is_large_translate << ")\n"

      << "        [-outfname string]\n"
      << "            output ppm file base name. When empty, no output.\n"
      << "            A frame number and extension (.ppm) will be added.\n"
      << "            (default: [" << m_outfname << "])\n"
      << "        [-verify_image_fname [image_fname]]\n"
     << "           when image_fname exist, verify the rendering image. (default: ["
      << m_verify_image_fname << "])\n"
      << "        [-unittest bool]\n"
      << "            when true, unit test mode. "
      << m_is_unittest << "])"
      << std::endl;
    exit(1);
  }
  return true;
}

void Scene_description_attribute::create_append_material_to_group(
  nv::index::IScene*                   scene_edit,
  nv::index::ITransformed_scene_group* group_node,
  const mi::math::Color&               ambient_color,
  const mi::math::Color&               diffuse_color,
  const mi::math::Color&               specular_color,
  const mi::Float32                    shiness,
  mi::neuraylib::IDice_transaction*    dice_transaction)
{
  check_success(scene_edit != 0);
  check_success(group_node != 0);
  check_success(dice_transaction != 0);

  // Add a fully ambient material for the planes, so that lighting doesn't matter
  mi::base::Handle<nv::index::IPhong_gl> phong_1(
    scene_edit->create_attribute<nv::index::IPhong_gl>());

  check_success(phong_1.is_valid_interface());
  phong_1->set_ambient(ambient_color);
  phong_1->set_diffuse(diffuse_color);
  phong_1->set_specular(specular_color);
  phong_1->set_shininess(shiness);

 mi::neuraylib::Tag phong_1_tag = dice_transaction->store_for_reference_counting(phong_1.get());
  check_success(phong_1_tag.is_valid());
  group_node->append(phong_1_tag, dice_transaction);
}

void Scene_description_attribute::create_append_plane_to_group(
  nv::index::IScene*                      scene_edit,
  nv::index::ITransformed_scene_group*    group_node,
  const mi::math::Vector<mi::Float32, 3>& plane_point,
  const mi::math::Vector<mi::Float32, 3>& plane_normal,
  const mi::math::Vector<mi::Float32, 3>& plane_up,
  const mi::math::Vector<mi::Float32, 2>& plane_extent,
  const mi::math::Color&                  col,
```

```
  mi::neuraylib::IDice_transaction*       dice_transaction)
{
  mi::base::Handle<nv::index::ITexture_filter_mode> tex_filter(
   scene_edit->create_attribute<nv::index::ITexture_filter_mode_nearest_neighbor>());
  check_success(tex_filter.is_valid_interface());
 mi::neuraylib::Tag tex_filter_tag = dice_transaction->store_for_reference_counting(tex_filter.get
  check_success(tex_filter_tag.is_valid());
  group_node->append(tex_filter_tag, dice_transaction);

 mi::base::Handle<Distributed_compute_constant_color_mapping> mapping(new Distributed_compute_cons
  check_success(mapping.is_valid_interface());
 const mi::neuraylib::Tag mapping_tag = dice_transaction->store_for_reference_counting(mapping.get
  check_success(mapping_tag.is_valid());
  group_node->append(mapping_tag, dice_transaction);

 mi::base::Handle<nv::index::IPlane> plane(scene_edit->create_shape<nv::index::IPlane>());
  check_success(plane.is_valid_interface());
  plane->set_point(plane_point);
  plane->set_normal(plane_normal);
  plane->set_up(plane_up);
  plane->set_extent(plane_extent);

 const mi::neuraylib::Tag plane_tag = dice_transaction->store_for_reference_counting(plane.get());
  check_success(plane_tag.is_valid());
  group_node->append(plane_tag, dice_transaction);
}

void Scene_description_attribute::create_append_depth_offset_attribute_to_group(
  nv::index::IScene*                   scene_edit,
  nv::index::ITransformed_scene_group* group_node,
  mi::Float32                          depth_offset,
  bool                                 is_depth_offset_attribute_on,
  mi::neuraylib::IDice_transaction*    dice_transaction)
{
  mi::base::Handle<nv::index::IDepth_offset> depth_offset_attr(
    scene_edit->create_attribute<nv::index::IDepth_offset>());
  check_success(depth_offset_attr.is_valid_interface());

  depth_offset_attr->set_depth_offset(depth_offset);
  depth_offset_attr->set_enabled(is_depth_offset_attribute_on);

  const mi::neuraylib::Tag depth_offset_attr_tag =
    dice_transaction->store_for_reference_counting(depth_offset_attr.get());
  check_success(depth_offset_attr_tag.is_valid());
  group_node->append(depth_offset_attr_tag, dice_transaction);
}

void Scene_description_attribute::create_append_depth_test_attribute_to_group(
  nv::index::IScene*                      scene_edit,
  nv::index::ITransformed_scene_group*    group_node,
  nv::index::IDepth_test::Depth_test_mode depth_test_operator,
  bool                                    is_depth_test_attribute_on,
  mi::neuraylib::IDice_transaction*       dice_transaction)
{
  mi::base::Handle<nv::index::IDepth_test> depth_test_attr(
    scene_edit->create_attribute<nv::index::IDepth_test>());
  check_success(depth_test_attr.is_valid_interface());
```

```
  depth_test_attr->set_depth_test(depth_test_operator);
  depth_test_attr->set_enabled(is_depth_test_attribute_on);

  const mi::neuraylib::Tag depth_test_attr_tag =
    dice_transaction->store_for_reference_counting(depth_test_attr.get());
  check_success(depth_test_attr_tag.is_valid());
  group_node->append(depth_test_attr_tag, dice_transaction);
}

void Scene_description_attribute::create_append_rendering_order_test_attribute_to_group(
  nv::index::IScene*                   scene_edit,
  nv::index::ITransformed_scene_group* group_node,
  mi::Uint32                           priority_offset,
  bool                                 is_rendering_order_attrib_on,
  mi::neuraylib::IDice_transaction*    dice_transaction)
{
  mi::base::Handle<nv::index::IRendering_order> rendering_order_attr(
    scene_edit->create_attribute<nv::index::IRendering_order>());
  check_success(rendering_order_attr.is_valid_interface());

  rendering_order_attr->set_order(priority_offset);

  const mi::neuraylib::Tag rendering_order_attr_tag =
    dice_transaction->store_for_reference_counting(rendering_order_attr.get());
  check_success(rendering_order_attr_tag.is_valid());
  group_node->append(rendering_order_attr_tag, dice_transaction);
}

void Scene_description_attribute::create_append_line_to_group(
  nv::index::IScene*                     scene_edit,
  nv::index::ITransformed_scene_group*   group_node,
  const mi::math::Vector<mi::Float32, 3>& v0,
  const mi::math::Vector<mi::Float32, 3>& v1,
  const mi::math::Color&                 col,
  const mi::Float32                      width,
  mi::neuraylib::IDice_transaction*      dice_transaction)
{
  std::vector< mi::math::Vector< mi::Float32, 3> > lseg_vec;
  std::vector< mi::math::Color > color_vec;
  std::vector< mi::Float32 >     width_vec;

  lseg_vec.push_back(v0);
  lseg_vec.push_back(v1);
  color_vec.push_back(col);
  width_vec.push_back(width);

  // FIXME workaround
  lseg_vec.push_back(v0);
  lseg_vec.push_back(v1);
  color_vec.push_back(col);
  width_vec.push_back(width);

  // POD type conversion
  std::vector< mi::math::Vector_struct< mi::Float32, 3> > lseg_vec_st;
  std::vector< mi::math::Color_struct > color_vec_st;
  std::transform(lseg_vec.begin(), lseg_vec.end(),
```

```
          std::back_inserter(lseg_vec_st), convert_to_vector_float32_3_st);
  std::transform(color_vec.begin(), color_vec.end(),
          std::back_inserter(color_vec_st), convert_to_color_st);

 mi::base::Handle<nv::index::ILine_set> line_set(scene_edit->create_shape<nv::index::ILine_set>()
  check_success(line_set.is_valid_interface());
  check_success((lseg_vec_st.size()) > 0);

  // line_set->set_lines(&(lseg_vec_st[0]), lseg_vec_st.size());
  line_set->set_lines(&(lseg_vec_st[0]),   lseg_vec_st.size());
  line_set->set_colors(&(color_vec_st[0]), color_vec_st.size());
  line_set->set_widths(&(width_vec[0]),    width_vec.size());
  line_set->set_line_type (nv::index::ILine_set::LINE_TYPE_SEGMENTS);
  line_set->set_line_style(nv::index::ILine_set::LINE_STYLE_SOLID);

 const mi::neuraylib::Tag line_set_tag = dice_transaction->store_for_reference_counting(line_set.g
  check_success(line_set_tag.is_valid());
  group_node->append(line_set_tag, dice_transaction);
}

void Scene_description_attribute::create_append_label_to_group(
  nv::index::IScene*                      scene_edit,
  nv::index::ITransformed_scene_group*    group_node,
  const mi::math::Vector<mi::Float32, 3>& label_point,
  const std::string&                      label_str,
  const mi::Float32                       label_height,
  const mi::Float32                       label_width,
  mi::neuraylib::IDice_transaction*       dice_transaction)
{
  // Add font to the scene description
 mi::base::Handle<nv::index::IFont> font(scene_edit->create_attribute<nv::index::IFont>());
  check_success(font.is_valid_interface());

  if (font->set_file_name(m_font_fpath.c_str()))
  {
    INFO_LOG << "set the font path [" << m_font_fpath << "]";
  }
  else
  {
    ERROR_LOG << "Can not find the font path [" << m_font_fpath << "], "
         << "the rendering result may not correct.";
  }
  font->set_font_resolution(64.0f);
 const mi::neuraylib::Tag font_tag = dice_transaction->store_for_reference_counting(font.get());
  check_success(font_tag.is_valid());
  group_node->append(font_tag, dice_transaction);

  // Add a label
  {
    mi::base::Handle<nv::index::ILabel_layout> label_layout(
      scene_edit->create_attribute<nv::index::ILabel_layout>());
    check_success(label_layout.is_valid_interface());
    const mi::Float32 padding = 10.0f;
    label_layout->set_padding(padding);
   mi::math::Color_struct fg_col; fg_col.r = 0.9f; fg_col.g = 0.95f; fg_col.b = 0.99f; fg_col.a = 1.0f
   mi::math::Color_struct bg_col; bg_col.r = 0.4f; bg_col.g = 0.5f;  bg_col.b = 0.4f;  bg_col.a = 0.5f;
    label_layout->set_color(fg_col, bg_col);
```

```
    const mi::neuraylib::Tag label_layout_tag =
      dice_transaction->store_for_reference_counting(label_layout.get());
    check_success(label_layout_tag.is_valid());
    group_node->append(label_layout_tag, dice_transaction);

   mi::base::Handle<nv::index::ILabel_3D> label(scene_edit->create_shape<nv::index::ILabel_3D>());
    check_success(label.is_valid_interface());
    label->set_text(label_str.c_str());

    const mi::math::Vector<mi::Float32, 3> right(1.0f, 0.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 3> up   (0.0f, 1.0f, 0.0f);
    label->set_geometry(label_point, right, up, label_height, label_width);
   const mi::neuraylib::Tag label_tag = dice_transaction->store_for_reference_counting(label.get())
    check_success(label_tag.is_valid());
    group_node->append(label_tag, dice_transaction);
  }
}

void Scene_description_attribute::create_append_point_to_group(
  nv::index::IScene*                      scene_edit,
  nv::index::ITransformed_scene_group*    group_node,
  const mi::math::Vector<mi::Float32, 3>& point_pos,
  const mi::math::Color&                  point_col,
  mi::Float32                             point_rad,
  mi::neuraylib::IDice_transaction*       dice_transaction)
{
  std::vector< mi::math::Vector< mi::Float32, 3> > point_pos_vec;
  std::vector< mi::math::Color > point_col_vec;
  std::vector< mi::Float32 >     point_rad_vec;

  point_pos_vec.push_back(point_pos);
  point_col_vec.push_back(point_col);
  point_rad_vec.push_back(point_rad);

  // Create attribute_point_set scene element and add it to the scene
 const nv::index::IPoint_set::Point_style style = nv::index::IPoint_set::FLAT_CIRCLE;
 mi::base::Handle<nv::index::IPoint_set> point_set(scene_edit->create_shape<nv::index::IPoint_set>
  check_success(point_set.is_valid_interface());
  point_set->set_point_style(style);

  // POD type conversion
  std::vector< mi::math::Vector_struct< mi::Float32, 3> > point_pos_vec_st;
  std::vector< mi::math::Color_struct > point_col_vec_st;
  std::transform(point_pos_vec.begin(), point_pos_vec.end(),
          std::back_inserter(point_pos_vec_st), convert_to_vector_float32_3_st);
  std::transform(point_col_vec.begin(), point_col_vec.end(),
          std::back_inserter(point_col_vec_st), convert_to_color_st);

  point_set->set_vertices(&point_pos_vec_st[0], point_pos_vec_st.size());
  point_set->set_colors(  &point_col_vec_st[0], point_col_vec_st.size());
  point_set->set_radii(   &point_rad_vec[0],    point_rad_vec.size());

 const mi::neuraylib::Tag point_set_tag = dice_transaction->store_for_reference_counting(point_set
  check_success(point_set_tag.is_valid());
  group_node->append(point_set_tag, dice_transaction);
}
```

```
mi::neuraylib::Tag Scene_description_attribute::create_depth_offset_attribute_test_scene(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  // create a group node for this test
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Add depth offset attribute node for the plane
  const mi::Float32 depth_offset = 0.0f; // base
 create_append_depth_offset_attribute_to_group(scene_edit, group_node.get(), depth_offset,
                        m_is_depth_offset_node_on, dice_transaction);

  // Add a plane
  const mi::math::Vector<mi::Float32, 2> plane_extent(1200.0f, 400.0f);
  const mi::math::Vector<mi::Float32, 3> plane_normal(0.0f, 0.0f, 1.0f);
  const mi::math::Vector<mi::Float32, 3> plane_up    (0.0f, 1.0f, 0.0f);
  const mi::math::Vector<mi::Float32, 3> plane_right (1.0f, 0.0f, 0.0f);
 const mi::math::Vector<mi::Float32, 3> plane_point (-600.0f, 400.0f, 0.0f + m_workaround_shift);
  {
    const mi::Float32 plane_depth_offset = 0.0f;
    create_append_depth_offset_attribute_to_group(scene_edit, group_node.get(),
                          plane_depth_offset, m_is_depth_offset_node_on,
                          dice_transaction);
    create_append_plane_to_group(scene_edit, group_node.get(),
                 plane_point, plane_normal, plane_up, plane_extent,
                 mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f), // red
                 dice_transaction);

    // plane annotation
    const mi::math::Vector<mi::Float32, 3> label_pos = plane_point +
      1.05f * plane_extent.x * plane_right +
      0.45f * plane_extent.y * plane_up;

    std::stringstream sstr;
   sstr << "Plane offset: " << (m_is_depth_offset_node_on ? plane_depth_offset : 0.0f);
    create_append_label_to_group(scene_edit, group_node.get(), label_pos, sstr.str(),
                 64.0f, 330.0f, dice_transaction);
  }

  const mi::math::Vector<mi::Float32, 3> plane_x_span(plane_extent.x, 0.0f, 0.0f);
  const mi::math::Vector<mi::Float32, 3> plane_y_span(0.0f, plane_extent.y, 0.0f);

  // Lines on the edges
  {
    const mi::Sint32 line_count = 2;
    const mi::Float32 ext_coeff = 0.05f; // 5%
    const mi::Float32 depth_offset_ary[line_count] = { 40.0f, -40.0f, };

    const mi::math::Vector< mi::Float32, 3> v0_ary[line_count] = {
      plane_point - ext_coeff * plane_x_span,
      plane_point - ext_coeff * plane_x_span + plane_y_span,
    };
    const mi::math::Vector< mi::Float32, 3> v1_ary[line_count] = {
      plane_point + (1.0f + ext_coeff) * plane_x_span,
      plane_point + (1.0f + ext_coeff) * plane_x_span + plane_y_span,
```

```
    };
    const mi::math::Color col_ary[line_count]        = {
      mi::math::Color(0.0f, 1.0f, 0.0f, 1.0f), // green
      mi::math::Color(0.0f, 0.0f, 1.0f, 1.0f), // blue
    };
    const mi::Float32     line_width_ary[line_count] = { 20.0f, 20.0f, };

    for(mi::Sint32 i = 0; i < line_count; ++i){
      create_append_depth_offset_attribute_to_group(scene_edit, group_node.get(),
                            depth_offset_ary[i], m_is_depth_offset_node_on,
                            dice_transaction);
    create_append_line_to_group(scene_edit, group_node.get(), v0_ary[i], v1_ary[i], col_ary[i],
                    line_width_ary[i], dice_transaction);
    }

    // line annotation
    const mi::Sint32 label_count = 2;
    const mi::math::Vector<mi::Float32, 3> label_point_ary[label_count] = {
      plane_point - 0.35f * plane_x_span + -0.1f * plane_y_span,
      plane_point - 0.35f * plane_x_span +  0.9f * plane_y_span,
    };
    const mi::Float32 label_height_ary[label_count] = {  64.0f,  64.0f, };
    const mi::Float32 label_width_ary [label_count] = { 350.0f, 350.0f, };

    for(mi::Sint32 i = 0; i < label_count; ++i){
      std::stringstream sstr;
    sstr << "Line offset: " << (m_is_depth_offset_node_on ? depth_offset_ary[i] : 0.0f);
    create_append_label_to_group(scene_edit, group_node.get(), label_point_ary[i], sstr.str(),
                    label_height_ary[i], label_width_ary[i],
                    dice_transaction);
    }
  }

  // Labels in the center
  {
    const mi::Sint32  label_count = 2;
    const mi::Float32 depth_offset_ary[label_count] = { 10.0f, 5.0f };
    const mi::math::Vector<mi::Float32, 3> label_point_ary[label_count] = {
      mi::math::Vector<mi::Float32, 3>(-500.0f, 600.0f, 0.0f + m_workaround_shift),
      mi::math::Vector<mi::Float32, 3>(-200.0f, 560.0f, 0.0f + m_workaround_shift),
    };
    const mi::Float32 label_height_ary[label_count] = {  64.0f,  64.0f, };
    const mi::Float32 label_width_ary [label_count] = { 640.0f, 640.0f, };

    for(mi::Sint32 i = 0; i < label_count; ++i){
      create_append_depth_offset_attribute_to_group(scene_edit, group_node.get(),
                            depth_offset_ary[i], m_is_depth_offset_node_on,
                            dice_transaction);

      std::stringstream sstr;
    sstr << "label 3D: depth offset: " << (m_is_depth_offset_node_on ? depth_offset_ary[i] : 0.0f);
    create_append_label_to_group(scene_edit, group_node.get(), label_point_ary[i], sstr.str(),
                    label_height_ary[i], label_width_ary[i],
                    dice_transaction);
    }
  }
```

```
  // Points on the edges
  {
    const mi::Sint32 point_count = 6;
    const mi::Float32 depth_offset_ary[point_count] = {
      60.0f,  20.0f, -60.0f,
      60.0f, -20.0f, -60.0f,
    };
    const mi::math::Vector<mi::Float32, 3> full_x(plane_extent.x, 0.0f, 0.0f);
   const mi::math::Vector<mi::Float32, 3> half_x(0.5f * plane_extent.x, 0.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 3> full_y(0.0f, plane_extent.y, 0.0f);
    const mi::math::Vector<mi::Float32, 3> point_pos_ary[point_count] = {
      plane_point,
      plane_point + half_x,
      plane_point + full_x,
      plane_point + full_y,
      plane_point + half_x + full_y,
      plane_point + full_x + full_y,
    };
    const mi::math::Color point_col_ary[point_count] = {
     mi::math::Color(1.0f, 1.0f, 0.0f, 1.0f), mi::math::Color(0.0f, 1.0f, 1.0f, 1.0f),
     mi::math::Color(1.0f, 0.0f, 1.0f, 1.0f), mi::math::Color(1.0f, 1.0f, 1.0f, 1.0f),
      mi::math::Color(0.5f, 0.5, 0.0f, 1.0f),  mi::math::Color(0.0f, 0.5, 0.5f, 1.0f),
    };
    const mi::Float32 point_rad_ary[point_count] = {
      19.0f, 19.0f, 19.0f, 19.0f, 19.0f, 19.0f,
    };

    for(mi::Sint32 i = 0; i < point_count; ++i){
      create_append_depth_offset_attribute_to_group(scene_edit, group_node.get(),
                            depth_offset_ary[i], m_is_depth_offset_node_on,
                            dice_transaction);
      create_append_point_to_group(scene_edit, group_node.get(),
                    point_pos_ary[i], point_col_ary[i], point_rad_ary[i],
                    dice_transaction);
    }


    // point annotation
    const mi::Sint32  label_count = point_count;
   const mi::Float32 label_height_ary[label_count] = {  64.0f,  64.0f,  64.0f,  64.0f,  64.0f,  64.0f,
   const mi::Float32 label_width_ary [label_count] = { 400.0f, 400.0f, 400.0f, 400.0f, 400.0f, 400.0f
   const mi::Float32 label_oc_ary[label_count] = {    -3.0f,  -3.0f,  -3.0f,   2.0f,   2.0f,   2.0f, };

    for(mi::Sint32 i = 0; i < label_count; ++i){
      std::stringstream sstr;
    sstr << "Point offset: " << (m_is_depth_offset_node_on ? depth_offset_ary[i] : 0.0f);
      const mi::math::Vector<mi::Float32, 3> label_pos = point_pos_ary[i] +
     label_oc_ary[i] * label_height_ary[i] * plane_up - 0.5f * label_width_ary[i] * plane_right;
    create_append_label_to_group(scene_edit, group_node.get(), label_pos, sstr.str(),
                    label_height_ary[i], label_width_ary[i],
                    dice_transaction);
    }
  }

  // put to db
 const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_no
  check_success(group_node_tag.is_valid());
```

```
  return group_node_tag;
}

void Scene_description_attribute::create_append_sphere(
  nv::index::IScene*                      scene_edit,
  nv::index::ITransformed_scene_group*    group_node,
  const mi::math::Color&                  diffuse_color,
  const mi::math::Vector<mi::Float32, 3>& center_pos,
  const mi::Float32                       radius,
  mi::neuraylib::IDice_transaction*       dice_transaction)
{
  // A material for the sphere
  create_append_material_to_group(scene_edit, group_node,
                  mi::math::Color(0.3f, 0.3f, 0.3f, 1.0f),
                  diffuse_color,
                  mi::math::Color(0.4f),
                  100.0f,
                  dice_transaction);

  {
   mi::base::Handle<nv::index::ISphere> sphere_1(scene_edit->create_shape<nv::index::ISphere>());
    check_success(sphere_1.is_valid_interface());
    sphere_1->set_center(center_pos);
    sphere_1->set_radius(radius);
   const mi::neuraylib::Tag sphere_1_tag = dice_transaction->store_for_reference_counting(sphere_1.
    check_success(sphere_1_tag.is_valid());
    group_node->append(sphere_1_tag, dice_transaction);
  }
}

mi::neuraylib::Tag Scene_description_attribute::create_z_test_attribute_test_scene(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  const mi::math::Vector<mi::Float32, 3> plane_up    (0.0f, 1.0f, 0.0f);
  const mi::math::Vector<mi::Float32, 3> plane_right (1.0f, 0.0f, 0.0f);

  // create a group node for this test
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Spheres
  {
    const mi::Sint32 sphere_count = 4;
    const mi::math::Color diffuse_color_ary[sphere_count] = {
      mi::math::Color(1.0f, 0.1f, 0.1f, 1.0f),
      mi::math::Color(0.1f, 1.0f, 0.1f, 1.0f),
      mi::math::Color(1.0f, 0.1f, 0.1f, 1.0f),
      mi::math::Color(0.1f, 1.0f, 0.1f, 1.0f),
    };
    const mi::math::Vector<mi::Float32, 3> center_position_ary[sphere_count] = {
      mi::math::Vector<mi::Float32, 3>(-450.0f,    0.0f, 0.0f),
      mi::math::Vector<mi::Float32, 3>(-450.0f,    0.0f, 0.0f),
      mi::math::Vector<mi::Float32, 3>(-450.0f, -200.0f, 0.0f),
      mi::math::Vector<mi::Float32, 3>(-450.0f, -200.0f, 0.0f),
```

```
    };
    const mi::Float32 radius_ary[sphere_count]  = {
      80.0f, 80.0f, 80.0f, 80.0f,
    };
    const nv::index::IDepth_test::Depth_test_mode ztest_mode_ary[sphere_count]  = {
      nv::index::IDepth_test::TEST_LESS_EQUAL,
      nv::index::IDepth_test::TEST_LESS_EQUAL,
      nv::index::IDepth_test::TEST_LESS,
      nv::index::IDepth_test::TEST_LESS,
    };

    for (mi::Sint32 i = 0; i < sphere_count; ++i)
    {
      create_append_depth_test_attribute_to_group(scene_edit, group_node.get(),
                           ztest_mode_ary[i],
                           m_is_z_test_attrib_on,
                           dice_transaction);

      create_append_sphere(scene_edit, group_node.get(), diffuse_color_ary[i],
                center_position_ary[i], radius_ary[i], dice_transaction);

      // sphere annotation
      const mi::Sint32  label_count = 2;
      const mi::Float32 label_height_ary[label_count] = {  64.0f,  64.0f, };
      const mi::Float32 label_width_ary [label_count] = { 420.0f, 420.0f, };
    const mi::Float32 label_y_oc_ary[label_count] = {   -0.7f,  -0.7f, }; // y offset coefficient
    const mi::Float32 label_x_oc_ary[label_count] = {   -1.3f,  -1.3f, }; // x offset coefficient
      const std::string ztest_mode[label_count] = {         "<=",    "<", };
      std::vector< mi::math::Vector<mi::Float32, 3> > sphere_center_vec;
      sphere_center_vec.push_back(center_position_ary[0]);
      sphere_center_vec.push_back(center_position_ary[2]);

      for(mi::Sint32 i = 0; i < label_count; ++i){
        std::stringstream sstr;
     sstr << "Z-test op: " << ztest_mode[i] << ": " << (m_is_z_test_attrib_on ? "on" : "off");
        const mi::math::Vector<mi::Float32, 3> label_pos = sphere_center_vec[i] +
          label_y_oc_ary[i] * label_height_ary[i] * plane_up +
          label_x_oc_ary[i] * label_width_ary[i]  * plane_right;
     create_append_label_to_group(scene_edit, group_node.get(), label_pos, sstr.str(),
                    label_height_ary[i], label_width_ary[i],
                    dice_transaction);
      }
    }
  }

  // Planes
  {
    const mi::math::Vector<mi::Float32, 3> plane_normal(0.0f, 0.0f, 1.0f);
    const mi::Sint32 plane_count = 4;

    create_append_material_to_group(scene_edit, group_node.get(),
                mi::math::Color(1.0f), mi::math::Color(0.0f), mi::math::Color(0.0f),
                    100.0f, dice_transaction);


    const mi::math::Vector<mi::Float32, 2> plane_extent_ary[plane_count] = {
      mi::math::Vector<mi::Float32, 2>( 700.0f,  80.0f),
```

```
      mi::math::Vector<mi::Float32, 2>( 700.0f,  80.0f),
      mi::math::Vector<mi::Float32, 2>( 400.0f, 130.0f),
      mi::math::Vector<mi::Float32, 2>( 400.0f, 130.0f),
    };
    const mi::math::Vector<mi::Float32, 3> plane_point_ary[plane_count] = {
      mi::math::Vector<mi::Float32, 3>(-150.0f, -100.0f, 0.0f + m_workaround_shift),
      mi::math::Vector<mi::Float32, 3>(-150.0f, -300.0f, 0.0f + m_workaround_shift),
      mi::math::Vector<mi::Float32, 3>(-200.0f, - 60.0f, 0.0f + m_workaround_shift),
      mi::math::Vector<mi::Float32, 3>(-200.0f, -260.0f, 0.0f + m_workaround_shift),
    };
    const mi::math::Color plane_color_ary[plane_count] = {
      mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f),
      mi::math::Color(1.0f, 0.0f, 0.0f, 1.0f),
      mi::math::Color(0.0f, 1.0f, 0.0f, 1.0f),
      mi::math::Color(0.0f, 1.0f, 0.0f, 1.0f),
    };
    const nv::index::IDepth_test::Depth_test_mode ztest_mode_ary[plane_count]  = {
      nv::index::IDepth_test::TEST_LESS_EQUAL,
      nv::index::IDepth_test::TEST_LESS,
      nv::index::IDepth_test::TEST_LESS_EQUAL,
      nv::index::IDepth_test::TEST_LESS,
    };

    for(mi::Sint32 i = 0; i < plane_count; ++i){
      // Add ambient only material
      create_append_depth_test_attribute_to_group(scene_edit, group_node.get(),
                           ztest_mode_ary[i],
                           m_is_z_test_attrib_on,
                           dice_transaction);
      create_append_plane_to_group(scene_edit, group_node.get(),
                    plane_point_ary[i], plane_normal, plane_up, plane_extent_ary[i],
                    plane_color_ary[i], dice_transaction);
    }
  }

 const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_no
  check_success(group_node_tag.is_valid());

  return group_node_tag;
}

void Scene_description_attribute::create_append_polygon(
  nv::index::IScene*                      scene_edit,
  nv::index::ITransformed_scene_group*    group_node,
  const mi::math::Color&                  diffuse_color,
  const mi::math::Vector<mi::Float32, 2>  vertex_pos_ary[],
  mi::Sint32                              vertex_count,
  const mi::math::Vector<mi::Float32, 3>& polygon_center,
  mi::neuraylib::IDice_transaction*       dice_transaction)
{
 mi::base::Handle<nv::index::IPolygon> poly(scene_edit->create_shape<nv::index::IPolygon>());
  check_success(poly.is_valid_interface());
  poly->set_geometry(vertex_pos_ary, vertex_count, polygon_center);
 poly->set_fill_style(convert_to_color_st(diffuse_color), nv::index::IPolygon::FILL_SOLID);

 const mi::neuraylib::Tag poly_tag = dice_transaction->store_for_reference_counting(poly.get());
  check_success(poly_tag.is_valid());
```

```
  group_node->append(poly_tag, dice_transaction);
}

mi::neuraylib::Tag Scene_description_attribute::create_rendering_order_attribute_test_scene(
  nv::index::IScene*                  scene_edit,
  mi::neuraylib::IDice_transaction*   dice_transaction)
{
  // create a group node for this test
  mi::base::Handle<nv::index::ITransformed_scene_group> group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(group_node.is_valid_interface());

  // Polygons (rectangles)
  {
    const mi::math::Vector<mi::Float32, 2> rect_vertex_base_ary[4] = {
      mi::math::Vector<mi::Float32, 2>(  0.0f,   0.0f),
      mi::math::Vector<mi::Float32, 2>(150.0f,   0.0f),
      mi::math::Vector<mi::Float32, 2>(150.0f, 100.0f),
      mi::math::Vector<mi::Float32, 2>(  0.0f, 100.0f),
    };

    const mi::Sint32 poly_count = 6;
    const mi::math::Color diffuse_color_ary[poly_count] = {
      mi::math::Color(1.0f, 0.1f, 0.1f, 1.0f), // red
      mi::math::Color(0.1f, 1.0f, 0.1f, 1.0f), // green
      mi::math::Color(0.1f, 0.1f, 1.0f, 1.0f), // blue

      mi::math::Color(1.0f, 0.1f, 0.1f, 1.0f), // red
      mi::math::Color(0.1f, 1.0f, 0.1f, 1.0f), // green
      mi::math::Color(0.1f, 0.1f, 1.0f, 1.0f), // blue
    };
    const mi::Uint32 priority_ary[poly_count]  = {
      0, 1, 2,
      2, 1, 0,
    };

    const mi::math::Vector<mi::Float32, 3> base_0(-850.0f, -800.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 3> base_1( 100.0f, -800.0f, 0.0f);
    const mi::math::Vector<mi::Float32, 3> rect_position_origin_ary[poly_count] = {
      base_0, base_0, base_0,
      base_1, base_1, base_1,
    };
    const mi::math::Vector<mi::Float32, 2> rect_vertex_offset_ary[poly_count] = {
      mi::math::Vector<mi::Float32, 2>(  0.0f,    0.0f),
      mi::math::Vector<mi::Float32, 2>( 35.0f,  -35.0f),
      mi::math::Vector<mi::Float32, 2>( 70.0f,  -70.0f),
      mi::math::Vector<mi::Float32, 2>(  0.0f,    0.0f),
      mi::math::Vector<mi::Float32, 2>( 35.0f,  -35.0f),
      mi::math::Vector<mi::Float32, 2>( 70.0f,  -70.0f),
    };

    mi::math::Vector<mi::Float32, 2> rect_vertex_ary[4];
    for (mi::Sint32 i = 0; i < poly_count; ++i)
    {
    create_append_rendering_order_test_attribute_to_group(scene_edit, group_node.get(),
                                priority_ary[i],
                                m_is_rendering_order_attrib_on,
```

```
                                dice_transaction);

      for (mi::Sint32 j = 0; j < 4; ++j)
      {
        rect_vertex_ary[j] = rect_vertex_base_ary[j] + rect_vertex_offset_ary[i];
      }

      create_append_polygon(scene_edit, group_node.get(), diffuse_color_ary[i],
                 rect_vertex_ary, 4,
                 rect_position_origin_ary[i],
                 dice_transaction);
    }


    // Rendering order annotation
    {
     const mi::math::Vector<mi::Float32, 3> main_label_pos(-1000.0f, -500.0f, 1.0f);
      std::stringstream sstr;
    sstr << "Rendering order attrib: " << (m_is_rendering_order_attrib_on ? "on" : "off");
    create_append_label_to_group(scene_edit, group_node.get(), main_label_pos, sstr.str(),
                   64.0f, 600.0f, dice_transaction);

      const mi::Sint32  label_count  = 6;
      const mi::Float32 label_height = 64.0f;
      const mi::Float32 label_width  = 300.0f;
      const mi::math::Vector<mi::Float32, 3> label_pos_offset(500.0f, 0.0f, 50.0f);

      for (mi::Sint32 i = 0; i < label_count; ++i)
      {
        std::stringstream sstr;
        sstr << "Priority: " << priority_ary[i];
       const mi::math::Vector<mi::Float32, 3> label_pos = rect_position_origin_ary[i]
          + mi::math::Vector<mi::Float32, 3>(2.0f * rect_vertex_offset_ary[i].x,
                            2.0f * rect_vertex_offset_ary[i].y,
                            0.0f)
          + label_pos_offset ;
      create_append_label_to_group(scene_edit, group_node.get(), label_pos, sstr.str(),
                      label_height, label_width, dice_transaction);
      }
    }
  }

 const mi::neuraylib::Tag group_node_tag = dice_transaction->store_for_reference_counting(group_no
  check_success(group_node_tag.is_valid());

  return group_node_tag;
}

void Scene_description_attribute::create_scene(
  nv::index::IScene*                scene_edit,
  mi::neuraylib::IDice_transaction* dice_transaction)
{
  // Add a scene group where the shapes should be added
  mi::base::Handle<nv::index::ITransformed_scene_group> main_group_node(
    scene_edit->create_scene_group<nv::index::ITransformed_scene_group>());
  check_success(main_group_node.is_valid_interface());
```

```
  // Add a light and a material
  {
    // Add a light
    mi::base::Handle<nv::index::IDirectional_headlight> headlight(
      scene_edit->create_attribute<nv::index::IDirectional_headlight>());
    check_success(headlight.is_valid_interface());
    const mi::math::Color_struct color_intensity = { 1.0f, 1.0f, 1.0f, 1.0f, };
    headlight->set_intensity(color_intensity);
    headlight->set_direction(mi::math::Vector<mi::Float32, 3>(1.0f, -1.0f, -1.0f));
   const mi::neuraylib::Tag headlight_tag = dice_transaction->store_for_reference_counting(headligh
    check_success(headlight_tag.is_valid());
    main_group_node->append(headlight_tag, dice_transaction);

    // Add a fully ambient material for the planes, so that lighting doesn't matter
    create_append_material_to_group(scene_edit, main_group_node.get(),
                    mi::math::Color(1.0f, 1.0f, 1.0f, 1.0f),
                    mi::math::Color(0.0f),
                    mi::math::Color(0.0f),
                    100.0f,
                    dice_transaction);
  }

  // depth-offset attribute test scene
  const mi::neuraylib::Tag depth_offset_group_tag = create_depth_offset_attribute_test_scene(
    scene_edit, dice_transaction);
  main_group_node->append(depth_offset_group_tag, dice_transaction);

  // 3D Z-test
  const mi::neuraylib::Tag z_test_group_tag = create_z_test_attribute_test_scene(
    scene_edit, dice_transaction);
  main_group_node->append(z_test_group_tag, dice_transaction);

  // Painters algorithm test
  const mi::neuraylib::Tag rendering_order_group_tag = create_rendering_order_attribute_test_scene(
    scene_edit, dice_transaction);
  main_group_node->append(rendering_order_group_tag, dice_transaction);

  // Finally append everything to the root of the hierachical scene description
  const mi::neuraylib::Tag main_group_tag = dice_transaction->store_for_reference_counting(main_grou
  check_success(main_group_tag.is_valid());
  scene_edit->append(main_group_tag, dice_transaction);
}

void Scene_description_attribute::setup_camera(nv::index::IPerspective_camera* cam) const
{
  check_success(cam != 0);

  // Set the camera parameters to see the whole scene.
  const mi::math::Vector<mi::Float32, 3> from(0.0f, 0.0f, 2000.0f);
  const mi::math::Vector<mi::Float32, 3> to  (0.0f, 0.0f,    0.0f);
  const mi::math::Vector<mi::Float32, 3> up  (0.0f, 1.0f,    0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.0f);
```

```
  cam->set_focal(0.03f);
  cam->set_clip_min(80.0f);
  cam->set_clip_max(5000.0f);
}

void Scene_description_attribute::setup_extreme_transformed_camera(nv::index::IPerspective_camera*
{
  check_success(cam != 0);

  // Adjusted the camera position for p = 23+1.
 mi::math::Vector< mi::Float32, 3 > const from( 1805060.125f + 5120.0f, 9978520.0f + 0.0f,   -30720.0
 mi::math::Vector< mi::Float32, 3 > const to  ( 1805060.125f + 0.0f,   9978520.0f + 2560.0f,    0.0f)
  mi::math::Vector< mi::Float32, 3 > const up  ( 0.0f,   1.0f,   0.0f);
  mi::math::Vector<mi::Float32, 3> viewdir = to - from;
  viewdir.normalize();

  cam->set(from, viewdir, up);
  cam->set_aperture(0.033f);
  cam->set_aspect(1.71520f); // not 1.7152034261242f, see IEEE754
  cam->set_focal(0.03f);
  cam->set_clip_min(124.485f); // not 124.485130310059f, see IEEE754
  cam->set_clip_max(17585.6f); // not 17585.55078125f,   see IEEE754

  INFO_LOG << "Set camera extreme mode";
}

mi::math::Matrix<mi::Float32, 4, 4> Scene_description_attribute::get_extreme_transformed_matrix() c
{
  // transform =
  //  [ 20.6100006103516   0                0       1805060
  //    0                 20.6100006103516  0       9978520
  //    0                  0               -4       0
  //    0                  0                0       1       ]

  mi::math::Matrix<mi::Float32, 4, 4> transform_mat(
    20.0f,       0.0f,       0.0f,  0.0f,
    0.0f,        20.0f,      0.0f,  0.0f,
    0.0f,        0.0f,      -20.0f,  0.0f,
    1805060.0f,  9978520.0f, 0.0f,  1.0f
    );
  return transform_mat;
}

nv::index::IFrame_results* Scene_description_attribute::render_frame(
  const std::string& output_fname) const
{
  check_success(m_index_rendering.is_valid_interface());

  // set output filename, empty string is valid
  m_image_file_canvas->set_rgba_file_name(output_fname.c_str());

  check_success(m_session_tag.is_valid());

  mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(
    m_global_scope->create_transaction<mi::neuraylib::IDice_transaction>());
  check_success(dice_transaction.is_valid_interface());
```

```
  m_index_session->update(m_session_tag, dice_transaction.get());

  mi::base::Handle<nv::index::IFrame_results> frame_results(
    m_index_rendering->render(
      m_session_tag,
      m_image_file_canvas.get(),
      dice_transaction.get()));
  check_success(frame_results.is_valid_interface());

  dice_transaction->commit();

  frame_results->retain();
  return frame_results.get();
}

int main(int argc, const char* argv[])
{
  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");               // log level
 sdict.insert("font_fpath", "/usr/share/fonts/liberation/LiberationSerif-Italic.ttf");
 sdict.insert("outfname", "frame_scene_description_attribute");    // output file base name
  sdict.insert("verify_image_fname", "");            // for unit test
  sdict.insert("unittest", "0");                     // default mode
 sdict.insert("is_large_translate", "0");        // large translation mode (default 0)
 sdict.insert("is_depth_offset_node_on", "1");    // depth offset node enabled (default 1)
 sdict.insert("is_z_test_attrib_on", "1");         // z-test node enabled (default 1)
 sdict.insert("global_z_test_mode", "1");          // global z-test mode (default 1)
 sdict.insert("is_rendering_order_attrib_on", "1"); // rendering order attribute enabled (default 1)
 sdict.insert("is_dump_comparison_image_when_failed", "1");  // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  // DiCE settings
  sdict.insert("dice::network::mode", "OFF");

  // IndeX settings
  sdict.insert("index::config::set_monitor_performance_values", "true");
  sdict.insert("index::service", "rendering_and_compositing");
  sdict.insert("index::cuda_debug_checks", "false");

  // Application_layer settings
  sdict.insert("index::app::components::application_layer::component_name_list",
        "canvas_infrastructure image io data_analysis_and_processing");

  // Initialize application
  Scene_description_attribute scene_description_attribute;
  scene_description_attribute.initialize(argc, argv, sdict);
  check_success(scene_description_attribute.is_initialized());

  // launch the application. creating the scene and rendering.
  const mi::Sint32 exit_code = scene_description_attribute.launch();
  INFO_LOG << "Shutting down ...";

  return exit_code;
}
```

9.30 session.cpp

```
/******************************************************************************
* Copyright 2023 NVIDIA Corporation. All rights reserved.
*****************************************************************************/

#include <mi/dice.h>
#include <sstream>
#include <map>

#include <nv/index/app/index_connect.h>
#include <nv/index/app/string_dict.h>

#include "utility/example_shared.h"

#include <nv/index/iindex.h>
#include <nv/index/isession.h>

int main(int argc, const char* argv[])
{
  const std::string com_name(argv[0]);

  nv::index::app::String_dict sdict;
  sdict.insert("dice::verbose", "3");                          // log level
  sdict.insert("unittest", "0");                               // default mode
 sdict.insert("is_dump_comparison_image_when_failed", "1"); // default: dump images when failed.
 sdict.insert("is_call_from_test", "0");              // default: not call from make check.

  if (nv::index::app::get_bool(sdict.get("unittest", "false")))
  {
    if (nv::index::app::get_bool(sdict.get("is_call_from_test", "false")))
    {
      sdict.insert("is_dump_comparison_image_when_failed", "0");
    }
    sdict.insert("dice::verbose", "2");
  }
  info_cout(std::string("running ") + com_name, sdict);

  {
    // This is a index_connect scope.

    // Load IndeX library via Index_connect
    nv::index::app::Index_connect index_connect;
    index_connect.initialize(argc, argv, sdict);

    // This scope is for handles
    {
    info_cout(std::string("NVIDIA IndeX version: ") + index_connect.get_index_interface()->get_vers

      // initialize_log_module(index_connect, opt_map);

    // Define service mode of the cluster machine. This suppress the following message.
      //   info: No particular service role has been assigned to the cluster machine.
    //  The IndeX library assigns it to service the 'rendering and compositing' by default.
     mi::base::Handle<nv::index::ICluster_configuration> rendering_properties_query(
        index_connect.get_index_interface()->
        get_api_component<nv::index::ICluster_configuration>());
```

```
  check_success(rendering_properties_query.is_valid_interface());
  rendering_properties_query->set_service_mode("rendering_and_compositing");
}

// session
{
  if (!index_connect.is_initialized())
  {
    printf("error: Fatal error! Initialization of the IndeX library failed.\n");
    return 1;
  }

  // Begin of internal application scope
  //   Note: index_connect has database, global_scope,
  //   index_session.  Here we demonstrate to create
  //   own. When you want to use index_connect's database,
  //   global_scope, index_session, you need to derive a own
  //   index_connect application class. See other API examples.
  mi::base::Handle<mi::neuraylib::IDatabase> database(
 index_connect.get_index_interface()->get_api_component<mi::neuraylib::IDatabase>());
  check_success(database.is_valid_interface());

mi::base::Handle<mi::neuraylib::IScope> global_scope(database->get_global_scope());
  check_success(global_scope.is_valid_interface());

mi::base::Handle<mi::neuraylib::IDice_transaction> dice_transaction(index_connect.create_trans
  check_success(dice_transaction.is_valid_interface());
  {
    // This is a dice_transaction scope.

    // IIndex_session to create an session
    mi::base::Handle<nv::index::IIndex_session> iindex_session(
   index_connect.get_index_interface()->get_api_component<nv::index::IIndex_session>());
    check_success(iindex_session.is_valid_interface());

    // Setup session information
 mi::neuraylib::Tag session_tag = iindex_session->create_session(dice_transaction.get());
    check_success(session_tag.is_valid());

    // Access the session
    mi::base::Handle<nv::index::ISession const> session(
      dice_transaction->access<nv::index::ISession const>(session_tag));
    check_success(session);

    std::stringstream sstr;

    // How to check the reference count status (create_session
    // makes one reference by the IndeX library. The handle
    // session has one reference. Totally two object reference this object.)
    session.get()->retain(); // manually reference count up (+1).
 const mi::Uint32 ref_count = session.get()->release(); // count down (-1), returns ref count.
    sstr << "Current reference count of session: " << ref_count;
    info_cout(sstr.str(), sdict);
    check_success(ref_count == 2);

    sstr.str("");
    sstr << "Created an session: tag = " << session_tag.id;
```

```
        info_cout(sstr.str(), sdict);
      }
      dice_transaction->commit(); // Finish this transaction
    }
    // shutdown when index_connect out of scope
  }

  return EXIT_SUCCESS;
}
```