Nodal Scene Interface

A flexible, modern API for renderers

Authors

Olivier Paquet, Aghiles Kheffache, François Colbert, Berj Bannayan

January 22, 2018

Contents

Chapter 1

Background

The Nodal Scene Interface (NSI) was developed to replace existing APIs in our renderer which are showing their age. Having been designed in the 80s and extended several times since, they include features which are no longer relevant and design decisions which do not reflect modern needs. This makes some features more complex to use than they should be and prevents or greatly increases the complexity of implementing other features.

The design of the NSI was shaped by multiple goals:

Simplicity The interface itself should be simple to understand and use, even if complex things can be done with it. This simplicity is carried into everything which derives from the interface.

Interactive rendering and scene edits Scene edit operations should not be a special case. There should be no difference between scene *description* and scene *edits*. In other words, a scene description is a series of edits and vice versa.

Tight integration with *Open Shading Language* OSL integration is not superficial and affects scene definition. For example, there are no explicit light sources in NSI: light sources are created by connected shaders with an `emission()` closure to a geometry.

Scripting The interface should be accessible from a platform independent, efficient and easily accessible scripting language. Scripts can be used to add render time intelligence to a given scene description.

Performance and multi-threading All API design decisions are made with performance in mind and this includes the possibility to run all API calls in a concurrent, multi-threaded environment. Nearly all software today which deals with large data sets needs to use multiple threads at some point. It is important for the interface to support this directly so it does not become a single thread communication bottleneck. This is why commands are self-contained and do not rely

on a current state. Everything which is needed to perform an action is passed in on every call.

Support for serialization The interface calls should be serializable. This implies a mostly unidirectional dataflow from the client application to the renderer and allows greater implementation flexibility.

Extensibility The interface should have as few assumptions as possible built-in about which features the renderer supports. It should also be abstract enough that new features can be added without looking out of place.

Chapter 2

The Interface

2.1 The interface abstraction

The Nodal Scene Interface is built around the concept of nodes. Each node has a unique handle to identify it and a type which describes its intended function in the scene. Nodes are abstract containers for data for which the interpretation depends on the node type. Nodes can also be connected to each other to express relationships.

Data is stored on nodes as attributes. Each attribute has a name which is unique on the node and a type which describes the kind of data it holds (strings, integer numbers, floating point numbers, etc).

Relationships and data flow between nodes are represented as connections. Connections have a source and a destination. Both can be either a node or a specific attribute of a node. There are no type restrictions for connections in the interface itself. It is acceptable to connect attributes of different types or even attributes to nodes. The validity of such connections depends on the types of the nodes involved.

What we refer to as the NSI has two major components:

- Methods to create nodes, attributes and their connections.
- Node types understood by the renderer. These are described in chapter 3.

Much of the complexity and expressiveness of the interface comes from the supported nodes. The first part was kept deliberately simple to make it easy to support multiple ways of creating nodes. We will list a few of those in the following sections but this list is not meant to be final. New languages and file formats will undoubtedly be supported in the future.

2.2 The C API

This section will describe in detail the C implementation of the NSI, as provided in the `nsi.h` file. This will also be a reference for the interface in other languages as all

concepts are the same.

```
#define NSI_VERSION 1
```

The `NSI_VERSION` macro exists in case there is a need at some point to break source compatibility of the C interface.

```
#define NSI_SCENE_ROOT ".root"
```

The `NSI_SCENE_ROOT` macro defines the handle of the root node.

```
#define NSI_ALL_NODES ".all"
```

The `NSI_ALL_NODES` macro defines a special handle to refer to all nodes in some contexts, such as the removing connections.

2.2.1 Context handling

```
NSIContext_t NSIBegin(
    int nparams,
    const NSIParam_t *params );

void NSIEnd( NSIContext_t ctx );
```

These two functions control creation and destruction of a NSI context, identified by a handle of type `NSIContext_t`. A context must be given explicitly when calling all other functions of the interface. Contexts may be used in multiple threads at once. The `NSIContext_t` is a convenience typedef and is defined as such:

```
typedef int NSIContext_t;
```

If `NSIBegin` fails for some reason, it returns `NSI_BAD_CONTEXT` which is defined in nsi.h:

```
#define NSI_BAD_CONTEXT ((NSIContext_t)0)
```

Optional parameters may be given to `NSIBegin()` to control the creation of the context:

`type` ..... `string` (`render`)
: Sets the type of context to create. The possible types are:

  `render` → To execute the calls directly in the renderer.

  `apistream` → To write the interface calls to a stream, for later execution. The target for writing the stream must be specified in another parameter.

`streamfilename` ..... `string`
: The file to which the stream is to be output, if the context type is apistream. Specify `"stdout"` to write to standard output and `"stderr"` to write to standard error.

`streamformat` ..... `string`
: The format of the command stream to write. Possible formats are:

`nsi` → Produces a NSI stream.

`binarynsi` → Produces a binary encoded NSI stream.

`lua` → Produces Lua API calls (refer to section 2.3).

`streamcompression` ..... `string`
The type of compression to apply to the written command stream.

`errorhandler` ..... `pointer`
A function which is to be called by the renderer to report errors. The default handler will print messages to the console.

`errorhandlerdata` ..... `pointer`
The `userdata` parameter of the error reporting function.

2.2.2 Passing optional parameters

```
struct NSIParam_t
{
    const char *name;
    const void *data;
    int type;
    int arraylength;
    size_t count;
    int flags;
};
```

This structure is used to pass variable parameter lists through the C interface. Most functions accept an array of the structure in a `params` parameter along with its length in a `nparams` parameter. The meaning of these two parameters will not be documented for every function. Instead, they will document the parameters which can be given in the array.

The `name` member is a C string which gives the parameter's name. The `type` member identifies the parameter's type, using one of the following constants:

- `NSITypeFloat` for a single 32-bit floating point value.
- `NSITypeDouble` for a single 64-bit floating point value.
- `NSITypeInteger` for a single 32-bit integer value.
- `NSITypeString` for a string value, given as a pointer to a C string.
- `NSITypeColor` for a color, given as three 32-bit floating point values.
- `NSITypePoint` for a point, given as three 32-bit floating point values.
- `NSITypeVector` for a vector, given as three 32-bit floating point values.
- `NSITypeNormal` for a normal vector, given as three 32-bit floating point values.

- `NSITypeMatrix` for a transformation matrix, given as 16 32-bit floating point values.
- `NSITypeDoubleMatrix` for a transformation matrix, given as 16 64-bit floating point values.
- `NSITypePointer` for a C pointer.

Array types are specified by setting the bit defined by the `NSIParamIsArray` constant in the `flags` member and the length of the array in the `arraylength` member. The `count` member gives the number of data items given as the value of the parameter. The `data` member is a pointer to the data for the parameter. The `flags` member is a bit field with a number of constants defined to communicate more information about the parameter:

- `NSIParamIsArray` to specify that the parameter is an array type, as explained previously.
- `NSIParamPerFace` to specify that the parameter has different values for every face of a geometric primitive, where this might be ambiguous.
- `NSIParamPerVertex` to specify that the parameter has different values for every vertex of a geometric primitive, where this might be ambiguous.
- `NSIParamInterpolateLinear` to specify that the parameter is to be interpolated linearly instead of using some other default method.
- `NSIParamIndirect` to specify that the parameter is read through an indirect lookup prior to interpolation. An integer parameter of the same name, with the `.indices` suffix added, is read to know which values of this parameter to use.

2.2.3 Node creation

```
void NSICreate(
    NSIContext_t context,
    NSIHandle_t handle,
    const char *type,
    int nparams,
    const NSIParam_t *params );
```

This function is used to create a new node. Its parameters are:

`context`
: The context returned by `NSIBegin`. See subsection 2.2.1

`handle`
: A node handle. This string will uniquely identify the node in the scene.

  If the supplied handle matches an existing node, the function does nothing if all other parameters match the call which created that node. Otherwise, it emits an error. Note that handles need only be unique within a given interface context. It

is acceptable to reuse the same handle inside different contexts. The `NSIHandle_t` typedef is defined in nsi.h:

```
typedef const char * NSIHandle_t;
```

`type`
: The type of node to create. See chapter 3.

`nparams, params`
: This pair describes a list of optional parameters. *There are no optional parameters defined as of now.* The `NSIParam_t` type is described in subsection 2.2.2.

```
void NSIDelete(
    NSIContext_t ctx,
    NSIHandle_t handle,
    int nparams,
    const NSIParam_t *params );
```

This function deletes a node from the scene. All connections to and from the node are also deleted. Note that it is not possible to delete the root or the global node. Its parameters are:

`context`
: The context returned by `NSIBegin`. See subsection 2.2.1

`handle`
: A node handle. It identifies the node to be deleted.

It accepts the following optional parameters:

`recursive` ..... `int (0)`
: Specifies whether deletion is recursive. By default, only the specified node is deleted. If a value of 1 is given, then nodes which connect to the specified node are recursively removed, unless they also have connections which do not eventually lead to the specified node. This allows, for example, deletion of an entire shader network in a single call.

```
void NSIRename(
    NSIContext_t ctx,
    NSIHandle_t old_handle,
    NSIHandle_t new_handle,
    int nparams,
    const NSIParam_t *params );
```

This function changes the handle of an existing node. The node's attributes remain unchanged and all connections to and from the node are kept. However, if the new handle already refers to a node, that one will be deleted, along with its connections, before being replaced. Its parameters are:

`context`
: The context returned by `NSIBegin`. See subsection 2.2.1

`old_handle`
: Identifies the node to be renamed.

`new_handle`
: Specifies the new handle to use for the node.

2.2.4 Setting attributes

```
void NSISetAttribute(
    NSIContext_t ctx,
    NSIHandle_t object,
    int nparams,
    const NSIParam_t *params );
```

This functions sets attributes on a previously created node. All optional parameters of the function become attributes of the node. On a shader node, this function is used to set the implicitly defined shader parameters. Setting an attribute using this function replaces any value previously set by `NSISetAttribute` or `NSISetAttributeAtTime`. To reset an attribute to its default value, use `NSIDeleteAttribute`.

```
void NSISetAttributeAtTime(
    NSIContext_t ctx,
    NSIHandle_t object,
    float time,
    int nparams,
    const NSIParam_t *params );
```

This function sets time-varying attributes (i.e. motion blurred). The `time` parameter specifies at which time the attribute is being defined. It is not required to set time-varying attributes in any particular order. In most uses, attributes that are motion blurred must have the same specification throughout the time range. A notable exception is the `P` attribute on particles which can be of different size for each time step because of appearing or disappearing particles. Setting an attribute using this function replaces any value previously set by `NSISetAttribute`.

```
void NSIDeleteAttribute(
    NSIContext_t ctx,
    NSIHandle_t object,
    const char *name );
```

This function deletes any attribute with a name which matches the `name` parameter on the specified object. There is no way to delete an attribute only for a specific time value.

Deleting an attribute resets it to its default value. For example, after deleting the `transformationmatrix` attribute on a transform node, the transform will be an identity. Deleting a previously set attribute on a shader node will default to whatever is declared inside the shader.

2.2.5 Making connections

```
void NSIConnect(
    NSIContext_t ctx,
    NSIHandle_t from,
    const char *from_attr,
    NSIHandle_t to,
    const char *to_attr,
    int nparams,
    const NSIParam_t *params );

void NSIDisconnect(
    NSIContext_t ctx,
    NSIHandle_t from,
    const char *from_attr,
    NSIHandle_t to,
    const char *to_attr );
```

These two functions respectively create or remove a connection between two elements. It is not an error to create a connection which already exists or to remove a connection which does not exist but the nodes on which the connection is performed must exist. The parameters are:

`from`
: The handle of the node from which the connection is made.

`from_attr`
: The name of the attribute from which the connection is made. If this is an empty string then the connection is made from the node instead of from a specific attribute of the node.

`to`
: The handle of the node to which the connection is made.

`to_attr`
: The name of the attribute to which the connection is made. If this is an empty string then the connection is made to the node instead of to a specific attribute of the node.

`NSIConnect` accepts additional optional parameters. Refer to section 5.8 for more about their utility.

With `NSIDisconnect`, the handle for either node may be the special value `".all"`. This will remove all connections which match the other three parameters. For example, to disconnect everything from the scene root:

```
NSIDisconnect( NSI_ALL_NODES, "", NSI_SCENE_ROOT, "objects" );
```

2.2.6 Evaluating procedurals

```
void NSIEvaluate(
    NSIContext_t ctx,
    int nparams,
    const NSIParam_t *params );
```

This function includes a block of interface calls from an external source into the current scene. It blends together the concepts of a straight file include, commonly known as an archive, with that of procedural include which is traditionally a compiled executable. Both are really the same idea expressed in a different language (note that for delayed procedural evaluation one should use the procedural node).

The NSI adds a third option which sits in-between—Lua scripts (section 2.3). They are much more powerful than a simple included file yet they are also much easier to generate as they do not require compilation. It is, for example, very realistic to export a whole new script for every frame of an animation. It could also be done for every character in a frame. This gives great flexibility in how components of a scene are put together.

The optional parameters accepted by this function are:

`filename` ..... `string`
: The name of the file which contains the interface calls to include.

`type` ..... `string`
: The type of file which will generate the interface calls. This can be one of:

  `apistream` → To read in a NSI stream.

  `lua` → To execute a Lua script, either from file or inline. See section 2.3 and more specifically subsection 2.3.3.

  `dynamiclibrary` → To execute native compiled code in a loadable library.

`script` ..... `string`
: A valid Lua script to execute when `type` is set to `"lua"`.

`parameters` ..... `string`
Optional procedural parameters.

`backgroundload` ..... `int` (0)
If this is nonzero, the object may be loaded in a separate thread, at some later time. This requires that further interface calls not directly reference objects defined in the included file. The only guarantee is that the file will be loaded before rendering begins.

2.2.7 Error reporting

```
enum NSIErrorLevel
{
    NSIErrMessage = 0,
    NSIErrInfo = 1,
    NSIErrWarning = 2,
    NSIErrError = 3
};

typedef void (*NSIErrorHandler_t)(
    void *userdata, int level, int code, const char *message );
```

This defines the type of the error handler callback given to the `NSIBegin` function. When it is called, the `level` parameter is one of the values defined by the `NSIErrorLevel` enum. The `code` parameter is a numeric identifier for the error message, or 0 when irrelevant. The `message` parameter is the text of the message.

The text of the message will not contain the numeric identifier nor any reference to the error level. It is usually desirable for the error handler to present these values together with the message. The identifier exists to provide easy filtering of messages.

The intended meaning of the error levels is as follows:

- `NSIErrMessage` for general messages, such as may be produced by printf in shaders. The default error handler will print this type of messages without an EOL terminator as it's the duty of the caller to format the message.
- `NSIErrInfo` for messages which give specific information. These might simply inform about the state of the renderer, files being read, settings being used and so on.
- `NSIErrWarning` for messages warning about potential problems. These will generally not prevent producing images and may not require any corrective action. They can be seen as suggestions of what to look into if the output is broken but no actual error is produced.
- `NSIErrError` for error messages. These are for problems which will usually break the output and need to be fixed.

2.2.8 Rendering

```
void NSIRenderControl(
    NSIContext_t ctx,
    int nparams,
    const NSIParam_t *params );
```

This function is the only control function of the API. It is responsible of starting, suspending and stopping the render. It also allows for synchronizing the render with interactive calls that might have been issued. The function accepts optional parameters:

`action` ..... `string`
Specifies the operation to be performed, which should be one of the following:

- `start` → This starts rendering the scene in the provided context. The render starts in parallel and the control flow is not blocked.
- `wait` → Wait for a render to finish.
- `synchronize` → For an interactive render, apply all the buffered calls to scene's state.
- `suspend` → Suspends render in the provided context.
- `resume` → Resumes a previously suspended render.
- `stop` → Stops rendering in the provided context without destroying the scene

`progressive` ..... `int (0)`
If set to 1, render the image in a progressive fashion.

`interactive` ..... `int (0)`
If set to 1, the renderer will accept commands to edit scene's state while rendering. The difference with a normal render is that the render task will not exit even if rendering is finished. Interactive renders are by definition progressive.

`frame` ..... `int`
Specifies the frame number of this render.

`stoppedcallback` ..... `pointer`
A pointer to a user function that should be called once rendering has stopped, either because a non-interactive render is complete, or because the `stop` action has been triggered. This function must have no return value and accept a single pointer argument.

`stoppedcallbackdata` ..... `pointer`
A pointer that will be passed back to the `stoppedcallback` function.

```
nsi.Create( "lambert", "shader" );
nsi.SetAttribute(
    "lambert",
    {name="filename", data="lambert_material.oso"},
    {name="Kd", data=.55},
    {name="albedo", data={1, 0.5, 0.3}, type=nsi.TypeColor} );

nsi.Create( "ggx", "shader" );
nsi.SetAttribute(
    "ggx",
    {
        {name="filename", data="ggx_material.oso"},
        {name="anisotropy_direction", data={0.13, 0 ,1}, type=nsi.TypeVector}
    } );
```

Listing 2.1: Shader creation example in Lua

2.3 The Lua API

The scripted interface is slightly different than its C counterpart since it has been adapted to take advantage of the niceties of Lua. The main differences with the C API are:

- No need to pass a NSI context to function calls since it's already embodied in the NSI Lua table (which is used as a class).
- The `type` parameter specified can be omitted if the parameter is an integer, real or string (as with the `Kd` and `filename` in the example below).
- NSI parameters can either be passed as a variable number of arguments or as a single argument representing an array of parameters (as in the `"ggx"` shader below)
- There is no need to call `NSIBegin` and `NSIEnd` equivalents since the Lua script is run in a valid context.

Listing 2.1 shows an example shader creation logic in Lua.

2.3.1 API calls

All useful (in a scripting context) NSI functions are provided and are listed in Table 2.1. There is also a `nsi.utilities` class which, for now, only contains a method to print errors. See subsection 2.3.5.

2.3.2 Function parameters format

Each single parameter is passed as a Lua table containing the following key values:

Lua Function	C equivalent
nsi.SetAttribute	NSISetAttribute
nsi.SetAttributeAtTime	NSISetAttributeAtTime
nsi.Create	NSICreate
nsi.Delete	NSIDelete
nsi.Rename	NSIRename
nsi.DeleteAttribute	NSIDeleteAttribute
nsi.Connect	NSIConnect
nsi.Disconnect	NSIDisconnect
Evaluate	NSIEvaluate

Table 2.1: NSI functions

- name - contains the name of the parameter.
- data - The actual parameter data. Either a value (integer, float or string) or an array.
- type - specifies the type of the parameter. Possible values are shown in Table 2.2.

Lua Type	C equivalent
nsi.TypeFloat	NSITypeFloat
nsi.TypeInteger	NSITypeInteger
nsi.TypeString	NSITypeString
nsi.TypeNormal	NSITypeNormal
nsi.TypeVector	NSITypeVector
nsi.TypePoint	NSITypePoint
nsi.TypeMatrix	NSITypeMatrix

Table 2.2: NSI types

- arraylength - specifies the length of the array for each element.

  NOTE — There is no count parameter in Lua since it can be obtained from the size of the provided data, its type and array length.

Here are some example of well formed parameters:

```
--[[ strings, floats and integers do not need a 'type' specifier ]] --
p1 = {name="shaderfilename", data="emitter"};
p2 = {name="power", data=10.13};
p3 = {name="toggle", data=1};
```

```
--[[ All other types, including colors and points, need a
    type specified for disambiguation. ]]--
p4 = {name="Cs", data={1, 0.9, 0.7}, type=nsi.TypeColor};

--[[ An array of 2 colors ]] --
p5 = {name="vertex_color", arraylength=2,
   data={1, 1, 1, 0, 0, 0}, type=nsi.TypeColor};

--[[ Create a simple mesh and connect it root ]] --
nsi.Create( "floor", "mesh" )
nsi.SetAttribute( "floor",
   {name="nvertices", data=4},
   {name="P", type=nsi.TypePoint,
      data={-2, -1, -1, 2, -1, -1, 2, 0, -3, -2, 0, -3}} )
nsi.Connect( "floor", "", ".root", "objects" )
```

2.3.3 Evaluating a Lua script

Script evaluation is started using `NSIEvaluate` in C, NSI stream or even another Lua script. Here is an example using NSI stream:

```
Evaluate
    "filename" "string" 1 ["test.nsi.lua"]
    "type" "string" 1 ["lua"]
```

It is also possible to evaluate a Lua script *inline* using the `script` parameter. For example:

```
Evaluate
    "script" "string" 1 ["nsi.Create(\"light\", \"shader\");"]
    "type" "string" 1 ["lua"]
```

Both "filename" and "script" can be specified to `NSIEvaluate` in one go, in which case the inline script will be evaluated before the file and both scripts will share the same NSI and Lua contexts. Any error during script parsing or evaluation will be sent to NSI's error handler. Note that all Lua scripts are run in a sandbox in which all Lua system libraries are disabled. Some utilities, such as error reporting, are available through the `nsi.utilities` class.

2.3.4 Passing parameters to a Lua script

All parameters passed to `NSIEvaluate` will appear in the `nsi.scriptparameters` table. For example, the following call:

```
Evaluate
    "filename" "string" 1 ["test.lua"]
    "type" "string" 1 ["lua"]
    "userdata" "color[2]" 1 [1 0 1 2 3 4]
```

Will register a `userdata` entry in the `nsi.scriptparameters` table. So executing the following line in `test.lua`:

```
print( nsi.scriptparameters.userdata.data[5] );
```

Will print 3.0.

2.3.5 Reporting errors from a Lua script

Use `nsi.utilities.ReportError` to send error messages to the error handler defined in the current NSI context. For example:

```
nsi.utilities.ReportError( nsi.ErrWarning, "Watch out!" );
```

The error codes are the same as in the C API and are shown in Table 2.3.

Lua Error Codes	C equivalent
nsi.ErrMessage	NSIErrMessage
nsi.ErrWarning	NSIErrMessage
nsi.ErrInfo	NSIErrInfo
nsi.ErrError	NSIErrError

Table 2.3: NSI error codes

2.4 The C++ API wrappers

The `nsi.hpp` file provides C++ wrappers which are less tedious to use than the low level C interface. All the functionality is inline so no additional libraries are needed and there are no ABI issues to consider.

To be continued ...

2.5 The interface stream

It is important for a scene description API to be streamable. This allows saving scene description into files, communicating scene state between processes and provide extra flexibility when sending commands to the renderer[1].

Instead of re-inventing the wheel, the authors have decided to use exactly the same format as is used by the *RenderMan* Interface Bytestream (RIB). This has several advantages:

- Well defined ASCII and binary formats.
- The ASCII format is human readable and easy to understand.
- Easy to integrate into existing renderers (writers and readers already available).

Note that since Lua is part of the API, one can use Lua files for API streaming[2].

[1] The streamable nature of the *RenderMan* API, through RIB, is an undeniable advantage. RenderMan® is a registered trademark of Pixar.

[2] Preliminary tests show that the Lua parser is as fast as an optimized ASCII RIB parser.

Chapter 3

Nodes

The following sections describe available nodes in technical terms. Refer to chapter 5 for usage details.

Node	Function	Reference
root	Scene's root	section 3.1
global	Global settings node	section 3.2
set	To express relationships to groups of nodes	section 3.3
shader	OSL shader or layer in a shader group	section 3.10
attributes	Container for generic attributes (e.g. visibility)	section 3.11
transform	Transformation to place objects in the scene	section 3.12
mesh	Polygonal mesh or subdivision surface	section 3.4
faceset	Assign attributes to part of a mesh	section 3.5
cubiccurves	B-spline and Catmull-Rom curves	section 3.6
particles	Collection of particles	section 3.7
procedural	Geometry to be loaded in delayed fashion	section 3.8
environment	Geometry type to define environment lighting	section 3.9
*camera	Set of nodes to create viewing cameras	section 3.16
outputdriver	Location where to output rendered pixels	section 3.13
outputlayer	Describes one render layer to be connected to an `outputdriver` node	section 3.14
screen	Describes how the view from a camera will be rasterized into an `outputlayer` node	section 3.15

Table 3.1: NSI nodes overview

3.1 The root node

The root node is much like a transform node with the particularity that it is the end connection for all renderable scene elements (see section 5.1). A node can exist in an NSI context without being connected to the root note but in that case it won't affect the render in any way. The root node has the reserved handle name `.root` and doesn't need to be created using `NSICreate`. The root node has two defined attributes: `objects` and `geometryattributes`. Both are explained in section 3.12.

3.2 The global node

This node contains various global settings for a particular NSI context. Note that these attributes are for the most case implementation specific. This node has the reserved handle name `.global` and doesn't need to be created using `NSICreate`. The following attributes are recognized by *3Delight*:

`numberofthreads` ..... `int` (`0`)
Specifies the total number of threads to use for a particular render:

- A value of zero lets the render engine choose an optimal thread value. This is the default behaviour.
- Any positive value directly sets the total number of render threads.
- A negative value will start as many threads as optimal *plus* the specified value. This allows for an easy way to decrease the total number of render threads.

`texturememory` ..... `int` (`250`)
Specifies the approximate memory size, in megabytes, the renderer will allocate to accelerate texture access.

`networkcache.size` ..... `int` (`15`)
Specifies the maximum network cache size, in gigabytes, the renderer will use to cache textures on a local drive to accelerate data access.

`networkcache.directory` ..... `string`
Specifies the directory in which textures will be cached. A good default value is /var/tmp/3DelightCache on Linux systems.

`license.server` ..... `string`
Specifies the name or address of the license server to be used.

`license.wait` ..... `int` (`1`)
When no license is available for rendering, the renderer will wait until a license is available if this attribute is set to `1`, but will stop immediately if it's set to `0`.

The latter setting is useful when managing a renderfarm and other work could be scheduled instead.

`license.hold` ..... `int` (`0`)
By default, the renderer will get new licenses for every render and release them once it's done. This can be undesirable if several frames are rendered in sequence from the same process. If this option is set to `1`, the licenses obtained for the first frame are held until the last frame is finished.

`renderatlowpriority` ..... `int` (`0`)
If set to 1, start the render with a lower process priority. This can be useful if there are other applications that must run during rendering.

`bucketorder` ..... `string` (`horizontal`)
Specifies in what order the buckets are rendered. The available values are:

`horizontal` → row by row, left to right and top to bottom.

`vertical` → column by column, top to bottom and left to right.

`zigzag` → row by row, left to right on even rows and right to left on odd rows.

`spiral` → in a clockwise spiral from the centre of the image.

`circle` → in concentric circles from the centre of the image.

`maximumraydepth.diffuse` ..... `int` (`1`)
Specifies the maximum bounce depth a diffuse ray can reach. A depth of 1 specifies one additional bounce compared to purely local illumination.

`maximumraydepth.hair` ..... `int` (`4`)
Specifies the maximum bounce depth a hair ray can reach. Note that hair are akin to volumetric primitives and might need elevated ray depth to properly capture the illumination.

`maximumraydepth.reflection` ..... `int` (`1`)
Specifies the maximum bounce depth a reflection ray can reach. Setting the reflection depth to 0 will only compute local illumination meaning that only emissive surfaces will appear in the reflections.

`maximumraydepth.refraction` ..... `int` (`4`)
Specifies the maximum bounce depth a refraction ray can reach. A value of 4 allows light to shine through a properly modeled object such as a glass.

`maximumraydepth.volume` ..... `int` (`0`)
Specifies the maximum bounce depth a volume ray can reach.

`maximumraylength.diffuse` ..... `double` (`-1`)
Limits the distance a ray emitted from a diffuse material can travel. Using a

relatively low value for this attribute might improve performance without significantly affecting the look of the resulting image, as it restrains the extent of global illumination. Setting it to a negative value disables the limitation.

`maximumraylength.hair` ..... `double` (`-1`)
Limits the distance a ray emitted from a hair closure can travel. Setting it to a negative value disables the limitation.

`maximumraylength.reflection` ..... `double` (`-1`)
Limits the distance a ray emitted from a reflective material can travel. Setting it to a negative value disables the limitation.

`maximumraylength.refraction` ..... `double` (`-1`)
Limits the distance a ray emitted from a refractive material can travel. Setting it to a negative value disables the limitation.

`maximumraylength.specular` ..... `double` (`-1`)
Limits the distance a ray emitted from a specular (glossy) material can travel. Setting it to a negative value disables the limitation.

`maximumraylength.volume` ..... `double` (`-1`)
Limits the distance a ray emitted from a volume can travel. Setting it to a negative value disables the limitation.

`quality.shadingsamples` ..... `int` (`1`)
Controls the quality of BSDF sampling. Larger values give less visible noise.

`quality.volumesamples` ..... `int` (`1`)
Controls the quality of volume sampling. Larger values give less visible noise.

`show.displacement` ..... `int` (`1`)
When set to `1`, enables displacement shading. Otherwise, it must be set to `0`, which forces the renderer to ignore any displacement shader in the scene.

`show.osl.subsurface` ..... `int` (`1`)
When set to `1`, enables the `subsurface()` OSL closure. Otherwise, it must be set to `0`, which will ignore this closure in OSL shaders.

`statistics.progress` ..... `int` (`0`)
When set to `1`, prints rendering progress as a percentage of completed pixels.

`statistics.filename` ..... `string` (`null`)
Full path of the file where rendering statistics will be written. An empty string will write statistics to standard output. The name `null` will not output statistics.

3.3 The set node

This node can be used to express relationships between objects. An example is to connect many lights to such a node to create a *light set* and then to connect this node to `outputlayer.lightset` (section 3.14 and section 5.7). It has the following attributes:

`objects` ..... `<connection>`
This connection accepts all nodes that are members of the set.

3.4 The mesh node

This node represents a polygon mesh. It has the following required attributes:

`P` ..... `point`
The positions of the object's vertices. Typically, this attribute will have the `NSIParamIndirect` flag and will be addressed indirectly through a `P.indices` attribute.

`nvertices` ..... `int`
The number of vertices for each face of the mesh. The number of values for this attribute specifies total face number (unless `nholes` is defined).

It also has optional attributes:

`nholes` ..... `int`
The number of holes in the polygons. When this attribute is defined, the total number of faces in the mesh is defined by the number of values for `nholes` rather than for `nvertices`. For each face, there should be (`nholes`+1) values in `nvertices`: the respective first value specifies the number of vertices on the outside perimeter of the face, while additional values describe the number of vertices on perimeters of holes in the face. Listing 3.1 shows the definition of a polygon mesh consisting of 3 square faces, with one triangular hole in the first one and square holes in the second one.

`clockwisewinding` ..... `int` (0)
A value of 1 specifies that polygons with a clockwise winding order are front facing. The default is 0, making counterclockwise polygons front facing.

`subdivision.scheme` ..... `string`
A value of `"catmull-clark"` will cause the mesh to render as a Catmull-Clark subdivision surface.

`subdivision.cornervertices` ..... `int`
This attribute is a list of vertices which are sharp corners. The values are indices into the `P` attribute, like `P.indices`.

```
Create "holey" "mesh"
SetAttribute "holey"
  "nholes" "int" 3 [ 1 2 0 ]
  "nvertices" "int" 6 [
    4 3               # Square with 1 triangular hole
    4 4 4             # Square with 2 square holes
    4 ]               # Square with 0 hole
  "P" "point" 23 [
    0 0 0  3 0 0  3 3 0  0 3 0
    1 1 0  2 1 0  1 2 0

    4 0 0  9 0 0  9 3 0  4 3 0
    5 1 0  6 1 0  6 2 0  5 2 0
    7 1 0  8 1 0  8 2 0  7 2 0

    10 0 0  13 0 0  13 3 0  10 3 0 ]
```

Listing 3.1: Definition of a polygon mesh with holes

`subdivision.cornersharpness` ..... `float`
This attribute is the sharpness of each specified sharp corner. It must have a value for each value given in `subdivision.cornervertices`.

`subdivision.creasevertices` ..... `int`
This attribute is a list of crease edges. Each edge is specified as a pair of indices into the `P` attribute, like `P.indices`.

`subdivision.creasesharpness` ..... `float`
This attribute is the sharpness of each specified crease. It must have a value for each pair of values given in `subdivision.creasevertices`.

3.5 The faceset node

This node is used to provide a way to attach attributes to some faces of another geometric primitive, such as the `mesh` node, as shown in Listing 3.2. It has the following attributes:

`faces` ..... `int`
This attribute is a list of indices of faces. It identifies which faces of the original geometry will be part of this face set.

3.6 The cubiccurves node

This node represents a group of cubic curves. It has the following required attributes:

```
Create "subdiv" "mesh"
SetAttribute "subdiv"
  "nvertices" "int" 4 [ 4 4 4 4 ]
  "P" "i point" 9 [
    0 0 0   1 0 0   2 0 0
    0 1 0   1 1 0   2 1 0
    0 2 0   1 2 0   2 2 2 ]
  "P.indices" "int" 16 [
    0 1 4 3   2 3 5 4   3 4 7 6   4 5 8 7 ]
  "subdivision.scheme" "string" 1 "catmull-clark"

Create "set1" "faceset"
SetAttribute "set1"
  "faces" "int" 2 [ 0 3 ]
Connect "set1" "" "subdiv" "facesets

Connect "attributes1" "" "subdiv" "geometryattributes"
Connect "attributes2" "" "set1" "geometryattributes"
```

Listing 3.2: Definition of a face set on a subdivision surface

`nvertices` ..... `int`
The number of vertices for each curve. This must be at least 4. There can be either a single value or one value per curve.

`P` ..... `point`
The positions of the curve vertices. The number of values provided, divided by `nvertices`, gives the number of curves which will be rendered.

`width` ..... `float`
The width of the curves.

`basis` ..... `string` (`catmull-rom`)
The basis functions used for curve interpolation. Possible choices are:

`b-spline` → B-spline interpolation.

`catmull-rom` → Catmull-Rom interpolation.

Attributes may also have a single value, one value per curve, one value per vertex or one value per vertex of a single curve, reused for all curves. Attributes which fall in that last category must always specify `NSIParamPerVertex`. Note that a single curve is considered a face as far as use of `NSIParamPerFace` is concerned.

3.7 The particles node

This geometry node represents a collection of *tiny* particles. Particles are repensted by either a disk or a sphere. This primitive is not suitable to render large particles as

these should be represented by other means (e.g. instancing).

`P` ..... `point`
A mandatory attribute that specifies the center of each particle.

`width` ..... `float`
A mandatory attribute that specifies the width of each particle. It can be specified for the entire particles node (only one value provided), per-particle or indirectly by using the `NSIParamIndirect` flag.

`N` ..... `normal`
The presence of a normal indicates that each particle is to be rendered as an oriented disk. The orientation of each disk is defined by the provided normal which can be constant or a per-particle attribute. Each particle is assumed to be a sphere if a normal is not provided.

`id` ..... `int`
This attribute, of the same size as `P`, assigns a unique identifier to each particle which must be constant throughout the entire shutter range. Its presence is necessary in the case where particles are motion blurred and some of them could appear or disappear during the motion interval. Having such identifiers allows the renderer to properly render such transient particles. This implies that the number of *id*s might vary for each time step of a motion-blurred particle cloud so the use of `NSISetAttributeAtTime` is mandatory by definition.

3.8 The procedural node

This node defines geometry that will be loaded in a delayed fashion. The natural parameter of such a construct is a bounding volume that strictly includes the geometric primitive:

`boundingbox` ..... `point[2]`
Specifies a bounding box for the geometry where

$$(boundingbox[0], boundingbox[1]) = (min, max).$$

In addition to this parameter, the procedural node accepts all the parameters of the `NSIEvaluate` API call, meaning that file formats accepted by that API call (NSI archives, dynamic libraries, LUA scripts) are also accepted by this node.

3.9 The environment node

This geometry node defines a sphere of infinite radius. Its only purpose is to render environment lights, solar lights and directional lights; lights which cannot be efficiently modeled using area lights. In practical terms, this node is no different than a geometry

node with the exception of shader execution semantics: there is no surface position `P`, only a direction `I` (refer to section 5.5 for more practical details). The following node attribute is recognized:

`angle` ..... `double` (360)
: Specifies the cone angle representing the region of the sphere to be sampled. The angle is measured around the Z+ axis[1]. If the angle is set to 0, the environment describes a directional light. Refer to section 5.5 for more about how to specify light sources.

3.10 The shader node

This node represents an OSL shader, also called layer when part of a shader group. It has the following required attribute:

`shaderfilename` ..... `string`
: This is the name of the file which contains the shader's compiled code.

All other attributes on this node are considered parameters of the shader. They may either be given values or connected to attributes of other shader nodes to build shader networks. OSL shader networks must form acyclic graphs or they will be rejected. Refer to section 5.4 for instructions on OSL network creation and usage.

3.11 The attributes node

This node is a container for various geometry related rendering attributes that are not *intrinsic* to a particular node (for example, one can't set the topology of a polygonal mesh using this attributes node). Instances of this node must be connected to the `geometryattributes` attribute of either geometric primitives or transform nodes (to build attributes hierarchies). Attribute values are gathered along the path starting from the geometric primitive, through all the transform nodes it is connected to, until the scene root is reached.

When an attribute is defined multiple times along this path, the definition with the highest priority is selected. In case of conflicting priorities, the definition that is the closest to the geometric primitive (i.e. the furthest from the root) is selected. Connections (for shaders, essentially) can also be assigned priorities, which are used in the same way as for regular attributes. Multiple attributes nodes can be connected to the same geometry or transform nodes (e.g. one attributes node can set object visibility and another can set the surface shader) and will all be considered.

This node has the following attributes:

[1] To position the environment dome one must connect the node to a transform node and apply the desired rotation.

`surfaceshader` ..... `<connection>`
The shader node which will be used to shade the surface is connected to this attribute. A priority (useful for overriding a shader from higher in the scene graph) can be specified by setting the `priority` attribute of the connection itself.

`displacementshader` ..... `<connection>`
The shader node which will be used to displace the surface is connected to this attribute. A priority (useful for overriding a shader from higher in the scene graph) can be specified by setting the `priority` attribute of the connection itself.

`volumeshader` ..... `<connection>`
The shader node which will be used to shade the volume inside the primitive is connected to this attribute.

`ATTR.priority` ..... `int (0)`
Sets the priority of attribute `ATTR` when gathering attributes in the scene hierarchy.

`visibility.camera` ..... `int (1)`
`visibility.diffuse` ..... `int (1)`
`visibility.hair` ..... `int (1)`
`visibility.reflection` ..... `int (1)`
`visibility.refraction` ..... `int (1)`
`visibility.shadow` ..... `int (1)`
`visibility.specular` ..... `int (1)`
`visibility.volume` ..... `int (1)`

These attributes set visibility for each ray type specified in OSL. The same effect could be achieved using shader code (using the `raytype()` function) but it is much faster to filter intersections at trace time. A value of `1` makes the object visible to the corresponding ray type, while `0` makes it invisible.

`visibility` ..... `int (1)`
This attribute sets the default visibility for all ray types. When visibility is set both per ray type and with this default visibility, the attribute with the highest priority is used. If their priority is the same, the more specific attribute (i.e. per ray type) is used.

`matte` ..... `int (0)`
If this attribute is set to 1, the object becomes a matte for camera rays. Its transparency is used to control the matte opacity and all other shading components are ignored.

3.12 The transform node

This node represents a geometric transformation. Transform nodes can be chained together to express transform concatenation, hierarchies and instances. Transform nodes also accept attributes to implement hierarchical attribute assignment and overrides. It has the following attributes:

`transformationmatrix` ..... `doublematrix`
This is a 4x4 matrix which describes the node's transformation. Matrices in NSI post-multiply column vectors so are of the form:

$$\begin{bmatrix} w_{1_1} & w_{1_2} & w_{1_3} & 0 \\ w_{2_1} & w_{2_2} & w_{2_3} & 0 \\ w_{3_1} & w_{3_2} & w_{3_3} & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$$

`objects` ..... `<connection>`
This is where the transformed objects are connected to. This includes geometry nodes, other transform nodes and camera nodes.

`geometryattributes` ..... `<connection>`
This is where attributes nodes may be connected to affect any geometry transformed by this node. Refer to section 5.2 and section 5.3 for explanation on how this connection is used.

3.13 The outputdriver node

An output driver defines how an image is transferred to an output destination. The destination could be a file (e.g. "exr" output driver), frame buffer or a memory address. It can be connected to the `outputdrivers` attribute of an output layer node. It has the following attributes:

`drivername` ..... `string`
This is the name of the driver to use. The API of the driver is implementation specific and is not covered by this documentation.

`imagefilename` ..... `string`
Full path to a file for a file-based output driver or some meaningful identifier depending on the output driver.

`embedstatistics` ..... `int (1)`
A value of 1 specifies that statistics will be embedded into the image file.

Any extra attributes are also forwarded to the output driver which may interpret them however it wishes.

3.14 The outputlayer node

This node describes one specific layer of render output data. It can be connected to the `outputlayers` attribute of a screen node. It has the following attributes:

`variablename` ..... `string`
This is the name of a variable to output.

`variablesource` ..... `string` (`shader`)
Indicates where the variable to be output is read from. Possible values are:

`shader` → computed by a shader and output through an OSL closure (such as `outputvariable()` or `debug()`) or the `Ci` global variable.

`attribute` → retrieved directly from an attribute with a matching name attached to a geometric primitive.

`builtin` → generated automatically by the renderer (e.g. "z", "alpha", "N" or "P").

`layername` ..... `string`
This will be name of the layer as written by the output driver. For example, if the output driver writes to an EXR file then this will be the name of the layer inside that file.

`scalarformat` ..... `string` (`uint8`)
Specifies the format in which data will be encoded (quantized) prior to passing it to the output driver. Possible values are:

`int8` → signed 8-bit integer

`uint8` → unsigned 8-bit integer

`int16` → signed 16-bit integer

`uint16` → unsigned 16-bit integer

`int32` → signed 32-bit integer

`uint32` → unsigned 32-bit integer

`half` → IEEE 754 half-precision binary floating point (binary16)

`float` → IEEE 754 single-precision binary floating point (binary32)

`layertype` ..... `string` (`color`)
Specifies the type of data that will be written to the layer. Possible values are:

`scalar` → A single quantity. Useful for opacity ("alpha") or depth ("Z") information.

`color` → A 3-component color.

`vector` → A 3D point or vector. This will help differentiate the data from a color in further processing.

`quad` → A sequence of 4 values, where the fourth value is not an alpha channel.

Each component of those types is stored according to the `scalarformat` attribute set on the same `outputlayer` node.

`colorprofile` ..... `string`
The name of an OCIO color profile to apply to rendered image data prior to quantization.

`dithering` ..... `integer (0)`
If set to 1, dithering is applied to integer scalars[2]. Otherwise, it must be set to 0.

`withalpha` ..... `integer (0)`
If set to 1, an alpha channel is included in the output layer. Otherwise, it must be set to 0.

`sortkey` ..... `integer`
This attribute is used as a sorting key when ordering multiple output layer nodes connected to the same output driver node. Layers with the lowest `sortkey` attribute appear first.

`lightset` ..... `<connection>`
This connection accepts either light sources or set nodes to which lights are connected. In this case only listed lights will affect the render of the output layer. If nothing is connected to this attribute then all lights are rendered.

`outputdrivers` ..... `<connection>`
This connection accepts output driver nodes to which the layer's image will be sent.

`filter` ..... `string (gaussian)`
The type of filter to use when reconstructing the final image from sub-pixel samples. Possible values are: "box", "triangle", "catmull-rom", "bessel", "gaussian", "sinc", "mitchell", "blackman-harris", "zmin" and "zmax".

`filterwidth` ..... `double (2.0)`
Diameter in pixels of the reconstruction filter. It is not applied when filter is "box" or "zmin".

Any extra attributes are also forwarded to the output driver which may interpret them however it wishes.

[2]It is sometimes desirable to turn off dithering, for example, when outputting object IDs.

3.15 The screen node

This node describes how the view from a camera node will be rasterized into an output layer node. It can be connected to the `screens` attribute of a camera node.

`outputlayers` ..... `<connection>`
: This connection accepts output layer nodes which will receive a rendered image of the scene as seen by the camera.

`resolution` ..... `integer[2]`
: Horizontal and vertical resolution of the rendered image, in pixels.

`oversampling` ..... `integer`
: The total number of samples (i.e. camera rays) to be computed for each pixel in the image.

`crop` ..... `float[2]`
: The region of the image to be rendered. It's defined by a list of exactly 2 pairs of floating-point number. Each pair represents a point in NDC space:

  - `Top-left` corner of the crop region
  - `Bottom-right` corner of the crop region

`prioritywindow` ..... `int[2]`
: For progressive renders, this is the region of the image to be rendered first. It is two pairs of integers. Each represents pixel coordinates:

  - `Top-left` corner of the high priority region
  - `Bottom-right` corner of the high priority region

`screenwindow` ..... `double[2]`
: Specifies the screen space region to the rendered. Each pair represents a 2D point in `screen` space:

  - `Bottom-left` corner of the region
  - `Top-right` corner of the region

  Note that the default screen window is set implicitely by the frame aspect ratio:

$$screenwindow = \left[-f \quad -1\right], \left[f \quad 1\right] \text{ for } f = \frac{xres}{yres}$$

`pixelaspectratio` ..... `float`
: Ratio of the physical width to the height of a single pixel. A value of 1.0 corresponds to square pixels.

3.16 Camera Nodes

All camera nodes share a set of common attributes. These are listed below.

`screens` ..... `<connection>`
This connection accepts screen nodes which will rasterize an image of the scene as seen by the camera. Refer to section 5.6 for more information.

`shutterrange` ..... `double`
Time interval during which the camera shutter is at least partially open. It's defined by a list of exactly two values:

- Time at which the shutter starts `opening`.
- Time at which the shutter finishes `closing`.

`shutteropening` ..... `double`
A *normalized* time interval indicating the time at which the shutter is fully open (a) and the time at which the shutter starts to close (b). These two values define the top part of a trapezoid filter. The end goal of this feature it to simulate a mechanical shutter on which open and close movements are not instantaneous. Figure 3.1 shows the geometry of such a trapezoid filter.

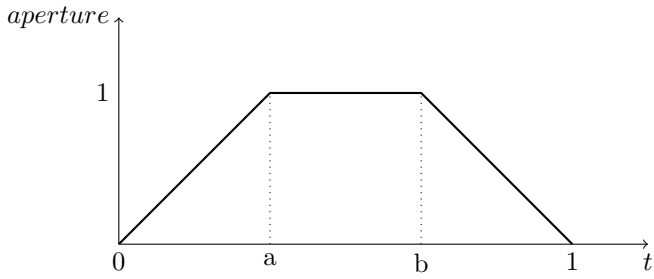

Figure 3.1: An example shutter opening configuration with a=1/3 and b=2/3.

`clippingrange` ..... `double`
Distance of the near and far clipping planes from the camera. It's defined by a list of exactly two values:

- Distance to the `near` clipping plane, in front of which scene objects are clipped.
- Distance to the `far` clipping plane, behind which scene objects are clipped.

3.16.1 The orthographiccamera node

This node defines an orthographic camera with a view direction towards the Z− axis. This camera has no specific attributes.

3.16.2 The perspectivecamera node

This node defines a perspective camera. The canonical camera is viewing in the direction of the Z− axis. The node is usually connected into a transform node for camera placement. It has the following attributes:

`fov` ..... `float`
The field of view angle, in degrees.

`depthoffield.enable` ..... `integer (0)`
Enables depth of field effect for this camera.

`depthoffield.fstop` ..... `double`
Relative aperture of the camera.

`depthoffield.focallength` ..... `double`
Focal length, in scene units, of the camera lens.

`depthoffield.focaldistance` ..... `double`
Distance, in scene units, in front of the camera at which objects will be in focus.

`depthoffield.aperture.enable` ..... `integer (0)`
By default, the renderer simulates a circular aperture for depth of field. Enable this feature to simulate aperture "blades" as on a real camera. This feature affects the look in out-of-focus regions of the image.

`depthoffield.aperture.sides` ..... `integer (5)`
Number of sides of the camera's aperture. The mininum number of sides is 3.

`depthoffield.aperture.angle` ..... `float (0)`
A rotation angle (in degrees) to be applied to the camera's aperture, in the image plane.

3.16.3 The fisheyecamera node

Fish eye cameras are useful for a multitude of applications (e.g. virtual reality). This node accepts these attributes:

`fov` ..... `float`
Specifies the field of view for this camera node, in degrees.

`mapping` ..... `string`
Defines one of the supported fisheye mapping functions:

`equidistant` → Maintains angular distances.

`equisolidangle` → Every pixel in the image covers the same solid angle.

`orthographic` → Maintains planar illuminance. This mapping is limited to a 180 field of view.

`stereographic` → Maintains angles throughout the image. Note that stereographic mapping fails to work with field of views close to 360 degrees.

3.16.4 The cylindricalcamera node

This node specifies a cylindrical projection camera and has the following attibutes:

`fov` ..... `float`
Specifies the *vertical* field of view, in degrees. The default value is 90.

`horizontalfov` ..... `float`
Specifies the horizontal field of view, in degrees. The default value is 360.

`eyeoffset` ..... `float`
This offset allows to render stereoscopic cylindrical images by specifying an eye offset

3.16.5 The sphericalcamera node

This node defines a spherical projection camera. This camera has no specific attributes.

3.16.6 Lens shaders

A lens shader is an OSL network connected to a camera through the `lensshader` connection. Such shaders receive the position and the direction of each tracer ray and can either change or completely discard the traced ray. This allows to implement distortion maps and cut maps. The following shader variables are provided:

`P` — Contains ray's origin.

`I` — Contains ray's direction. Setting this variable to zero instructs the renderer not to trace the corresponding ray sample.

`time` — The time at which the ray is sampled.

`(u, v)` — Coordinates, in screen space, of the ray being traced.

Chapter 4

Script Objects

It is a design goal to provide an easy to use and flexible scripting language for NSI. The Lua language has been selected for such a task because of its performance, lightness and features[1]. A flexible scripting interface greatly reduces the need to have API extensions. For example, what is known as "conditional evaluation" and "Ri filters" in the *RenderMan* API are superseded by the scripting features of NSI.

> NOTE — Although they go hand in hand, scripting objects are not to be confused with the Lua binding. The binding allows for calling NSI functions in Lua while scripting objects allow for scene inspection and decision making in Lua. Script objects can make Lua binding calls to make modifications to the scene.

To be continued ...

[1] Lua is also portable and streamable.

Chapter 5

Rendering Guidelines

5.1 Basic scene anatomy

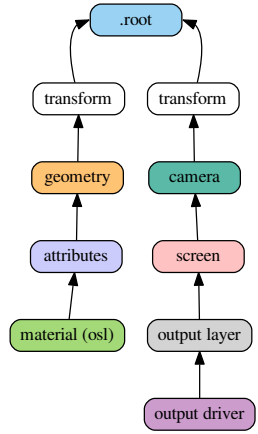

Figure 5.1: The fundamental building blocks of an NSI scene

A minimal (and useful) NSI scene graph contains the three following components:

1. Geometry linked to the `.root` node, usually through a transform chain
2. OSL materials linked to scene geometry through an attributes node [1]

[1] For the scene to be visible, at least one of the materials has to be emissive.

3. At least one *outputdriver* → *outputlayer* → *screen* → *camera* → `.root` chain to describe a view and an output device

The scene graph in Figure 5.1 shows a renderable scene with all the necessary elements. Note how the connections always lead to the `.root` node. In this view, a node with no output connections is not relevant by definition and will be ignored.

5.2 A word – or two – about attributes

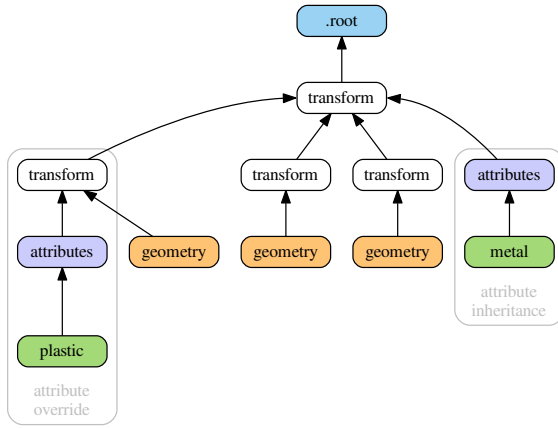

Figure 5.2: Attribute inheritance and override

Those familiar with the *RenderMan* standard will remember the various ways to attach information to elements of the scene (standard attributes, user attributes, primitive variables, construction parameters[2]). In NSI things are simpler and all attributes are set through the `SetAttribute()` mechanism. The only distinction is that some attributes are required (*intrinsic attributes*) and some are optional: a mesh node needs to have `P` and `nvertices` defined — otherwise the geometry is invalid[3]. In OSL shaders, attributes are accessed using the `getattribute()` function and *this is the only way to access attributes in* NSI. Having one way to set and to access attributes makes things simpler (a design goal) and allows for extra flexibility (another design goal). Figure 5.2

[2]Parameters passed to `Ri` calls to build certain objects. For example, knot vectors passed to `RiNuPatch`.

[3]In this documentation, all intrinsic attributes are usually documented at the beginning of each section describing a particular node.

shows two features of attribute assignment in NSI:

Attributes inheritance Attributes attached at some parent transform (in this case, a *metal* material) affect geometry downstream

Attributes override It is possible to override attributes for a specific geometry by attaching them to a transform directly upstream (the *plastic* material overrides *metal* upstream)

Note that any non-intrinsic attribute can be inherited and overridden, including vertex attributes such as texture coordinates.

5.3 Instancing

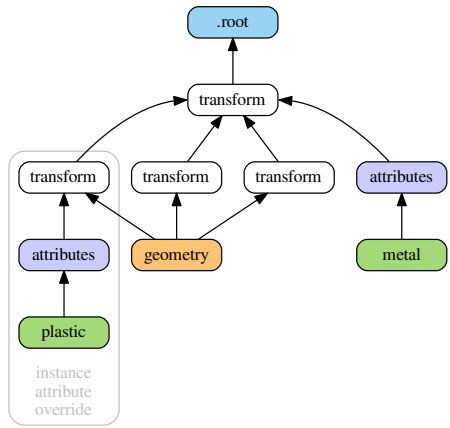

Figure 5.3: Instancing in NSI with attribute inheritance and per-instance attribute override

Instancing in NSI is naturally performed by connecting a geometry to more than one transform (connecting a geometry node into a `transform.objects` attribute). Figure 5.3 shows a simple scene with a geometry instanced three times. The scene also demonstrates how to override an attribute for one particular geometry instance, an operation very similar to what we have seen in section 5.2. Note that transforms can also be instanced and this allows for *instances of instances* using the same semantics.

5.4 Creating osl networks

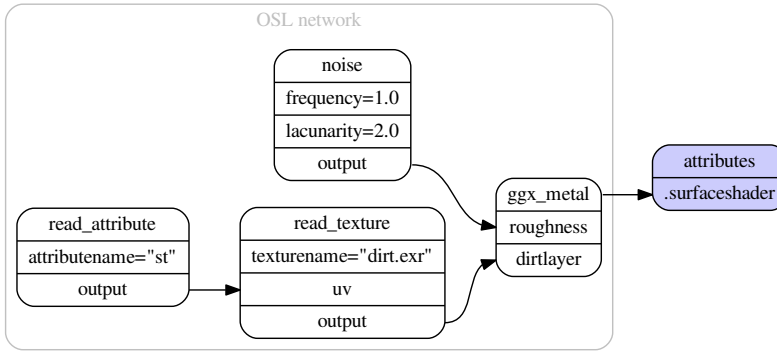

Figure 5.4: A simple OSL network connected to an attributes node

The semantics used to create OSL networks are the same as for scene creation. Each shader node in the network corresponds to a shader node which must be created using `NSICreate`. Each shader node has implicit attributes corresponding to shader's parameters and connection between said parameters is done using `NSIConnect`. Figure 5.4 depicts a simple OSL network connected to an attributes node. Some observations:

- Both the source and destination attributes (passed to `NSIConnect`) must be present and map to valid and compatible shader parameters (lines 21-23)
- There is no *symbolic linking* between shader parameters and geometry attributes (a.k.a. primvars). One has to explicitly use the `getattribute()` OSL function to read attributes attached to geometry. In Listing 5.1 this is done in the `read_attribute` node (lines 11-14). More about this subject in section 5.2.

```
Create "ggx_metal" "shader"
SetAttribute "ggx"
    "shaderfilename" "string" 1 ["ggx.oso"]

Create "noise" "shader"
SetAttribute "noise"
    "shaderfilename" "string" 1 ["simplenoise.oso"]
    "frequency" "float" 1 [1.0]
    "lacunarity" "float" 1 [2.0]

Create "read_attribute" "shader"
SetAttribute "read_attribute"
    "shaderfilename" "string" 1 ["read_attributes.oso"]
    "attributename" "string" 1 ["st"]

Create "read_texture" "shader"
SetAttribute "read_texture"
    "shaderfilename" "string" 1 ["read_texture.oso"]
    "texturename" "string" 1 ["dirt.exr"]

Connect "read_attribute" "output" "read_texture" "uv"
Connect "read_texture" "output" "ggx_metal" "dirtlayer"
Connect "noise" "output" "ggx_metal" "roughness"

# Connect the OSL network to an attribute node
Connect "ggx_metal" "Ci" "attr" "surfaceshader"
```

Listing 5.1: NSI stream to create the OSL network in Figure 5.4

5.5 Lighting in the nodal scene interface

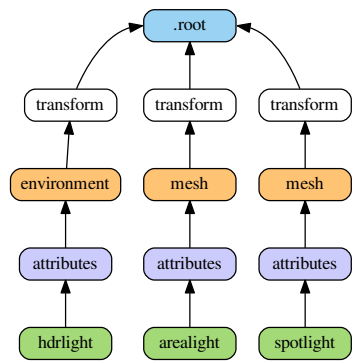

Figure 5.5: Various lights in NSI are specified using the same semantics

There are no special light source nodes in NSI (although the environment node, which defines a sphere of infinite radius, could be considered as a light in practice). Any scene geometry can become a light source if its surface shader produces an `emission()` closure. Some operations on light sources, such as *light linking*, are done using more general approaches (see section 5.8). Follows a quick summary on how to create different kinds of light in NSI.

5.5.1 Area lights

Area lights are created by attaching an emissive surface material to geometry. Listing 5.2 shows a simple OSL shader for such lights (standard OSL emitter).

```
// Copyright (c) 2009-2010 Sony Pictures Imageworks Inc., et al. All Rights Reserved.
surface emitter [[ string help = "Lambertian emitter material" ]]
(
    float power = 1 [[ string help = "Total power of the light" ]],
    color Cs = 1 [[ string help = "Base color" ]])
{
    // Because emission() expects a weight in radiance, we must convert by dividing
    // the power (in Watts) by the surface area and the factor of PI implied by
    // uniform emission over the hemisphere. N.B.: The total power is BEFORE Cs
    // filters the color!
    Ci = (power / (M_PI * surfacearea())) * Cs * emission();
}
```

Listing 5.2: Example emitter for area lights

5.5.2 Spot and point lights

Such lights are created using an epsilon sized geometry (a small disk, a particle, etc.) and optionally using extra parameters to the `emission()` closure.

```
surface spotLight(
    color i_color = color(1),
    float intenstity = 1,
    float coneAngle = 40,
    float dropoff = 0,
    float penumbraAngle = 0 )
{
    color result = i_color * intenstity * M_PI;

    /* Cone and penumbra */
    float cosangle = dot(-normalize(I), normalize(N));
    float coneangle = radians(coneAngle);
    float penumbraangle = radians(penumbraAngle);

    float coslimit = cos(coneangle / 2);
    float cospen = cos((coneangle / 2) + penumbraangle);
```

```
    float low = min(cospen, coslimit);
    float high = max(cospen, coslimit);

    result *= smoothstep(low, high, cosangle);

    if (dropoff > 0)
    {
        result *= clamp(pow(cosangle, 1 + dropoff),0,1);
    }
    Ci = result / surfacearea() * emission();
}
```

Listing 5.3: An example OSL spot light shader

5.5.3 Directional and HDR lights

Directional lights are created by using the environment node and setting the `angle` attribute to 0. HDR lights are also created using the environment node, albeit with a 2π cone angle, and reading a high dynamic range texture in the attached surface shader. Other directional constructs, such as *solar lights*, can also be obtained using the environment node.

Since the environment node defines a sphere of infinite radius any connected OSL shader must only rely on the `I` variable and disregard `P`, as is shown in Listing 5.4.

```
shader hdrlight( string texturename = "" )
{
    vector wi = transform("world", I);

    float longitude = atan2(wi[0], wi[2]);
    float latitude = asin(wi[1]);

    float s = (longitude + M_PI) / M_2PI;
    float t = (latitude + M_PI_2) / M_PI;

    Ci = emission() * texture (texturename, s, t);
}
```

Listing 5.4: An example OSL shader to do HDR lighting

> NOTE — Environment geometry is visible to camera rays by default so it will appear as a background in renders. To disable this simply switch off camera visibility on the associated attributes node.

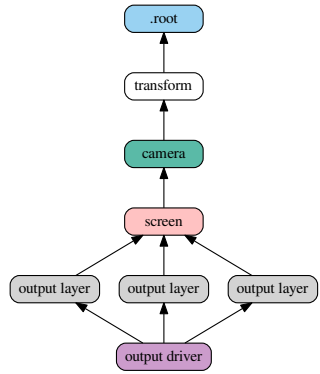

Figure 5.6: NSI graph showing the image output chain

5.6 Defining output drivers and layers

NSI allows for a very flexible image output model. All the following operations are possible:

- Defining many outputs in the same render (e.g. many EXR outputs)
- Defining many output layers per output (e.g. multi-layer EXRs)
- Rendering different scene views per output layer (e.g. one pass stereo render)
- Rendering images of different resolutions from the same camera (e.g. two viewports using the same camera, in an animation software)

Figure 5.6 depicts a NSI scene to create one file with three layers. In this case, all layers are saved to the same file and the render is using one view. A more complex example is shown in Figure 5.7: a left and right cameras are used to drive two file outputs, each having two layers (`Ci` and `Diffuse` colors).

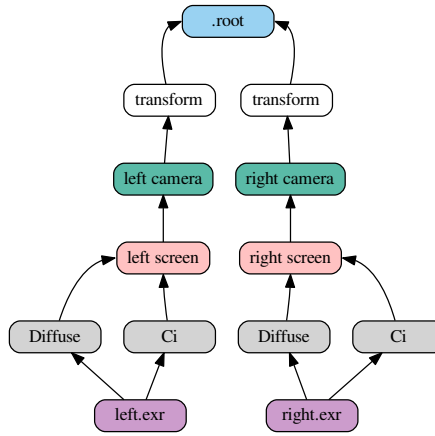

Figure 5.7: NSI graph for a stereo image output

5.7 Light layers

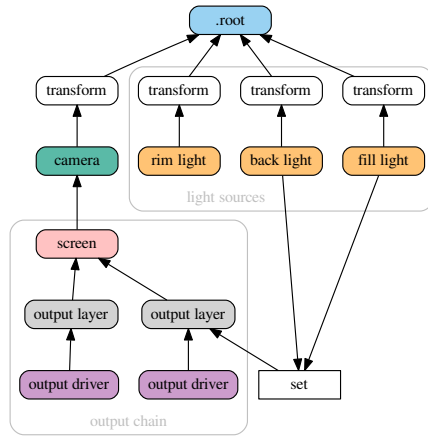

Figure 5.8: Gathering contribution of a subset of lights into one output layer

The ability to render a certain set of lights per output layer has a formal workflow in NSI. One can use three methods to define the lights used by a given output layer:

1. Connect the geometry defining lights directly to the `outputlayer.lightset` attribute
2. Create a set of lights using the `set` node and connect it into `outputlayer.lightset`
3. A combination of both 1 and 2

Figure 5.8 shows a scene using method 2 to create an output layer containing only illumination from two lights of the scene. Note that if there are no lights or light sets connected to the `lightset` attribute then all lights are rendered. The final output pixels contain the illumination from the considered lights on the specific surface variable specified in `outputlayer.variablename` (section 3.14).

5.8 Inter-object visibility

Some common rendering features are difficult to achieve using attributes and hierarchical tree structures. One such example is inter-object visibility in a 3D scene. A special case of this feature is *light linking* which allows the artist to select which objects a particular light illuminates, or not. Another classical example is a scene in which a ghost character is invisible to camera rays but visible in a mirror.

In NSI such visibility relationships are implemented using cross-hierarchy connection between one object and another. In the case of the mirror scene, one would first tag the character invisible using the `visibility` attribute and then connect the attribute node of the receiving object (mirror) to the visibility attribute of the source object (ghost) to *override* its visibility status. Essentially, this "injects" a new value for the ghost visibility for rays coming from the mirror.

Figure 5.9 depicts a scenario where both hierarchy attribute overrides and inter-object visibility are applied:

- The ghost transform has a visibility attribute set to 0 which makes the ghost invisible to all ray types
- The hat of the ghost has its own attribute with a visibility set to 1 which makes it visible to all ray types
- The mirror object has its own attributes node that is used to override the visibility of the ghost as seen from the mirror. The NSI stream code to achieve that would look like this:

  ```
  Connect "mirror_attribute" "" "ghost_attributes" "visibility"
      "value" "int" 1 [1]
      "priority" "int" 1 [2]
  ```

  Here, a priority of 2 has been set on the connection for documenting purposes, but it could have been omitted since connections always override regular attributes of equivalent priority.

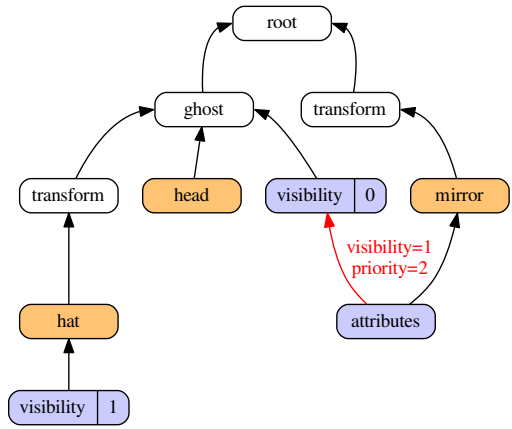

Figure 5.9: Visibility override, both hierarchically and inter-object

Acknowledgements

Many thanks to John Haddon, Daniel Dresser, David Minor and Gregory Ducatel for initiating the first discussions and encouraging us to design a new scene description API. Bo Zhou and Paolo Berto helped immensely with plug-in design which ultimately led to improvements in NSI (e.g. adoption of the `screen` node). Jordan Thistelwood opened the way for the first integration of NSI into a commercial plug-in. Stefan Habel did a thorough proof reading of the entire document and gave many suggestions.

The NSI logo was designed by Paolo Berto.

List of Figures

List of Tables

Listings

Index