NSI

Olivier Paquet, Aghiles Kheffache, François Colbert, Berj Banna

Dec 21, 2021

GENERAL

Authors
Olivier Paquet, Aghiles Kheffache, François Colbert, Berj Bannayan

CHAPTER

ONE

BACKGROUND

The Nodal Scene Interface (s) was developed to replace existing APIs in the 3Delight renderer which were showing their age. Particualry the RenderMan Interface and the RenderMan Shading Language.

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 s 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
: s integration is not superficial and affects scene definition. For example, there are no explicit light sources in s: light sources are created by connecting 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

TWO

HELP WANTED

The s API is used in the 3Delight renderer. More and more users of this renderer are switching their pipelines from using the *RenderMan Interface™* to s.

Aka: this *is* being used in production.

2.1 Naming

There are many things that lack coherence & stringency in naming of parts of the API.

The current documentation has new naming suggestions for some arguments, attributes and nodes that are marked with exclamation marks (**!**).

If you see a name written differently below the current name and marked with (**!**) this is a change suggestion.

Feedback on these is welcome. Please go to the GitHub repository for this documentation and open a ticket or comment on an existing one.

2.2 Spelling, Grammar & Content

If you find typos, grammar mistakes or think something should be changed or added to improve this documentation, do not hesitate to go ahead and open a pull request with your changes.

Each page has an *Edit on GitHub* button on the top right corner to make this process as painless as possible.

2.3 Language Bindings

The actual API is **C** which makes it easy to bind s to many different languages.

Currently the 3Delight renderer ships with free s bindings for **C++**, **Python** and **Lua**. There is also a Rust binding.

More bindings are always welcome!

CHAPTER

THREE

THE INTERFACE

3.1 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. 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 s has two major components:

- Methods to create nodes, attributes and their connections.
- *Node types* understood by the renderer.

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.

3.2 APIs

3.2.1 The C API

This section describes the C implementation of the s, 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 *removing connections*.

```
#define NSI_ALL_ATTRIBUTES ".all"
```

The NSI_ALL_ATTRIBUTES macro defines a special handle to refer to all attributes in some contexts, such as *removing connections*.

Context Handling

```
NSIContext_t NSIBegin(
   int n_params,
   const NSIParam_t *args
)
```

```
void NSIEnd(
   NSIContext_t ctx
)
```

These two functions control creation and destruction of a s 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:

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

Table 1: NSIBegin() optional arguments

Name	Type	Description/Values
`type`	string	Sets the type of context to create. The possibletypes are: `render` — Execute the calls directly in the renderer. This is the **default**. `apistream` — To write the interface calls to a stream, for later execution. The target for writing the stream must be specified in another argument.
`streamfilename` `stream.filename` (!)	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` `stream.format` (!)	string	The format of the command stream to write. Possible formats are: `nsi` — Produces an *nsi stream* `binarynsi` — Produces a binary encoded *nsi stream*
`stream.compression` `stream.compression` (!)	string	The type of compression to apply to the written command stream.
`streampathreplacement` `stream.path.replace`	int	Use `0` to disable replacement of path prefixes by references to environment variables which begin with `NSI_PATH_` in an s stream. This should generally be left enabled to ease creation of files which can be moved between systems.
`errorhandler`	pointer	A function which is to be called by the renderer to report errors. The default handler will print messages to the console.
`errorhandler.data`	pointer	The `userdata` argument of the *error reporting function*.
`executeprocedurals` `evaluate.replace` (!)	string	A list of procedural types that should be executed immediately when a call to *NSIEvaluate()* or a procedural node is encountered and `NSIBegin()`'s output `type` is `apistream`. This will replace any matching call to `NSIEvaluate()` with the results of the procedural's execution.

Arguments vs. Attributes

Arguments are what a user specifies when calling a function of the API. Each function takes extra, optional arguments.

Attributes are properties of nodes and are only set *through* the aforementioed optional arguments using the `NSISetAttribute()` and `NSISetAttributeAtTime()` functions.

Optional Arguments

Any API call can take extra arguments. These are always optional. What this means the call can do work without the user specifying these arguments.

Nodes are special as they have mandatory extra **attributes** that are set *after* the node is created inside the API but which must be set *before* the geometry or concept the node represents can actually be created in the scene.

These attributes are passed as extra arguments to the `NSISetAttribute()` and `NSISetAttributeAtTime()` functions.

Note: Nodes can also take extra **arguments** when they are created. These optional arguments are only meant to add information needed to create the node that a particular implementation may need.

As of this writing there is no implementation that has any such optional arguments on the `NSICreate()` **function.** The possibility to specify them is solely there to make the API future proof.

> **Caution:** Nodes do *not* have optional arguments for now. **An optional argument on a node is not the same as an attribute on a node.**

Attributes – Describe the Node's Specifics

Attributes are *only* for nodes. They must be set using the `NSISetAttribute()` or `NSISetAttributeAtTime()` functions.

They can **not** be set on the node when it is created with the `NSICreate()` function.

> **Caution:** **Only nodes have attributes.** They are sent to the API via optional arguments on the API's attribute functions.

Passing Optional Arguments

```
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 argument lists through the C interface. Most functions accept an array of the structure in a `args` argument along with its length in a `n_params` argument.

The meaning of these two arguments will not be documented for every function. Instead, each function will document the arguments which can be given in the array.

`name`

A C string which gives the argument's name.

`type`

Identifies the argument's type, using one of the following constants:

Table 2: types of optional arguments

`NSITypeFloat`	Single 32-bit floating point value.
`NSITypeDouble`	Single 64-bit floating point value.
`NSITypeInteger`	Single 32-bit integer value.
`NSITypeString`	String value, given as a pointer to a C string.
`NSITypeColor`	Color, given as three 32-bit floating point values.
`NSITypePoint`	Point, given as three 32-bit floating point values.
`NSITypeVector`	Vector, given as three 32-bit floating point values.
`NSITypeNormal`	Normal vector, given as three 32-bit floating point values.
`NSITypeMatrix`	Transformation matrix, in row-major order, given as 16 32-bit floating point values.
`NSITypeDoubleMatrix`	Transformation matrix, in row-major order, given as 16 64-bit floating point values.
`NSITypePointer`	C pointer.

Tuple types are specified by setting the bit defined by the `NSIArgIsArray` constant in the `flags` member and the length of the tuple in the `arraylength` member.

Tip: It helps to view `arraylength` as a part of the data type. The data type is a tuple with this length when `NSIArgIsArray` is set.

Note: If `NSIArgIsArray` is not set, `arraylength` is *ignored*.

The `NSIArgIsArray` flag is neccessary to distinguish between arguments that happen to be of *length* 1 (set in the `count` member) and tuples that have a *length* of 1 (set in the `arraylength` member) for the resp. argument.

Listing 1: A tuple argument of length 1 (and count 1) vs. a (non-tuple) argument of count 1

```
"foo" "int[1]" 1 [42]  # The answer to the ultimate question - in an a (single) tuple
"bar" "int" 1 13       # My favorite Friday
```

The `count` member gives the number of data items given as the value of the argument.

The `data` member is a pointer to the data for the argument. This is a pointer to a single value or a number values. Depending on `type`, `count` and `arraylength` settings.

Note: When data is an array, the actual number of elements in the array is $count \times arraylength \times n$. Where n is specified implictly through the `type` member in the table above.

For example, if the type is `NSITypeColor` (**3** values), `NSIArgIsArray` is set, `arraylength` is **2** and `count` is **4**, `data` is expected to contain **24** 32-bit floating point values ($3 \times 2 \times 4$).

The `flags` member is a bit field with a number of constants used to communicate more information about the argument:

Table 3: flags for optional arguments

`NSIArgIsArray`	to specify that the argument is an array type, as explained above.
`NSIArgPerFace`	to specify that the argument has different values for every face of a geometric primitive, where this might be ambiguous.
`NSIArgPerVertex`	Specify that the argument has different values for every vertex of a geometric primitive, where this might be ambiguous.
`NSIArgInterpolateLinear`	Specify that the argument is to be interpolated linearly instead of using some other, default method.

Note: `NSIArgPerFace` or `NSIArgPerVertex` are only strictly needed in rare circumstances when a geometric primitive's number of vertices matches the number of faces. The most simple case is a tetrahedral mesh which has exactly four vertices and also four faces.

Indirect lookup of arguments is achieved by giving an integer argument of the same name, with the `.indices` suffix added. This is read to know which values of the other argument to use.

Listing 2: A subdivision mesh using `P.indices` to reference the `P` argument

```
Create "subdiv" "mesh"
SetAttribute "subdiv"
  "nvertices" "int" 4 [ 4 4 4 4 ]
```

(continues on next page)

(continued from previous page)

```
  "P" "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"
```

Node Creation

```
void NSICreate(
    NSIContext_t context,
    NSIHandle_t handle,
    const char *type,
    int n_params,
    const NSIParam_t *args
)
```

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

`context`
: The context returned by `NSIBegin()`. See *context handling*.

`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 arguments 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.

`n_params`, `args` This pair describes a list of optional arguments. The `NSIParam_t` type is described in *this section*.

> **Caution:** There are *no* optional arguments defined as of now.

```
void NSIDelete(
    NSIContext_t ctx,
    NSIHandle_t handle,
    int n_params,
```

(continues on next page)

(continued from previous page)

```
    const NSIParam_t *args
)
```

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 arguments are:

`context`
    The context returned by `NSIBegin()`. See *context handling*.

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

It accepts the following optional arguments:

Table 4: NSIDelete() optional arguments

Name	Type	Description/Values
recursive	int	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 meet one of the following conditions: • They also have connections which do not eventually lead to the specified node. • Their connection to the deleted node was created with a `strength` greater than `0`. This allows, for example, deletion of an entire shader network in a single call.

Setting Attributes

```
void NSISetAttribute(
    NSIContext_t ctx,
    NSIHandle_t object,
    int n_params,
    const NSIParam_t *args
)
```

This functions sets attributes on a previously node. All *optional arguments* of the function become attributes of the node.

On a *shader node*, this function is used to set the implicitly defined shader arguments.

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,
    double time,
    int n_params,
    const NSIParam_t *args
)
```

This function sets time-varying attributes (i.e. motion blurred). The `time` argument 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` argument 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* node will default to whatever is declared inside the shader.

Making Connections

```
void NSIConnect(
    NSIContext_t ctx,
    NSIHandle_t from,
    const char *from_attr,
    NSIHandle_t to,
    const char *to_attr,
    int n_params,
    const NSIParam_t *args
)
```

```
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 arguments 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 arguments.

Table 5: NSIConnect() optional arguments

Name	Type	Description/Values
`value`		This can be used to change the value of a node's attribute in some contexts. Refer to *guidelines on inter-object visibility* for more information about the utility of this parameter.
`priority`		When connecting attribute nodes, indicates in which order the nodes should be considered when evaluating the value of an attribute.
`strength`	int (0)	A connection with a strength greater than `0` will *block* the progression of a recursive `NSIDelete`.

Severing Connections

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

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

Evaluating Procedurals

```
void NSIEvaluate(
    NSIContext_t ctx,
    int n_params,
    const NSIParam_t *args
)
```

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 s adds a third option which sits in-between — *Lua scripts*. 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 ability to load s commands straight from memory is also provided.

The optional arguments accepted by this function are:

Table 6: NSIEvaluate() optional arguments

<table>
<tr><th>Name</th><th>Type</th><th colspan="2">Description/Values</th></tr>
<tr><td rowspan="4">type</td><td rowspan="4">string</td><td colspan="2">The type of file which will generate the interface calls. This can be one of:</td></tr>
<tr><td>apistream</td><td>Read in an nsi stream. This requires either filename or buffer/size arguments to be specified too.</td></tr>
<tr><td>lua</td><td>Execute a Lua script, either from file or inline. See also how to evaluate a Lua script.</td></tr>
<tr><td>dynamiclibrary</td><td>Execute native compiled code in a loadable library. See dynamic library procedurals for an implementation example.</td></tr>
<tr><td>filename
stream.filename (!)</td><td>string</td><td colspan="2">The file from which to read the interface stream.</td></tr>
<tr><td>script</td><td>string</td><td colspan="2">A valid Lua script to execute when type is set to lua.</td></tr>
<tr><td>buffer
size</td><td>pointer
int</td><td colspan="2">These two arguments define a memory block that contains s commands to execute.</td></tr>
<tr><td>backgroundload</td><td>int</td><td colspan="2">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 the file will be loaded before rendering begins.</td></tr>
</table>

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` argument is one of the values defined by the `NSIErrorLevel` enum. The `code` argument is a numeric identifier for the error message, or 0 when irrelevant. The `message` argument 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:

Table 7: error levels

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.

Rendering

```
void NSIRenderControl(
    NSIContext_t ctx,
    int n_params,
    const NSIParam_t *args
)
```

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 :

Table 8: NSIRenderControl() intrinsic argument

<table>
<tr><th>Name</th><th>Type</th><th colspan="2">Description/Values</th></tr>
<tr><td rowspan="7">action</td><td rowspan="7">string</td><td colspan="2">Specifies the operation to be performed, which should be one of the following:</td></tr>
<tr><td>start</td><td>This starts rendering the scene in the provided context. The render starts in parallel and the control flow is not blocked.</td></tr>
<tr><td>wait</td><td>Wait for a render to finish.</td></tr>
<tr><td>synchronize</td><td>For an interactive render, apply all the buffered calls to scene's state.</td></tr>
<tr><td>suspend</td><td>Suspends render in the provided context.</td></tr>
<tr><td>resume</td><td>Resumes a previously suspended render.</td></tr>
<tr><td>stop</td><td>Stops rendering in the provided context without destroying the scene.</td></tr>
</table>

Table 9: NSIRenderControl() optional arguments

`progressive`	integer	If set to 1, render the image in a progressive fashion.
`interactive`	integer	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`		Specifies the frame number of this render.

`stoppedcallback`
`callback` (!)
	pointer	A pointer to a user function that should be called on rendering status changes. This function must have no return value and accept a pointer argument, a s context argument and an integer argument:
		`void StoppedCallback(`
		`    void* stoppedcallbackdata,`
		`    NSIContext_t ctx,`
		`    int status`
		`)`
		The third argument is an integer which can take the following values:
		• `NSIRenderCompleted` indicates that rendering has completed normally.
		• `NSIRenderAborted` indicates that rendering was interrupted before completion.
		• `NSIRenderSynchronized` indicates that an interactive render has produced an image which reflects all changes to the scene.
		• `NSIRenderRestarted` indicates that an interactive render has received new changes to the scene and no longer has an up to date image.

`stoppedcallbackdata` `callback.data` (!)	pointer	A pointer that will be passed back to the `stoppedcallback` function.

3.2.2 The C++ API

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.

Creating a Context

The core of these wrappers is the `NSI::Context` class. Its default construction will require linking with the renderer.

```
#include "nsi.hpp"

NSI::Context nsi;
```

The nsi_dynamic.hpp file provides an alternate api source which will load the renderer at runtime and thus requires no direct linking.

```
#include "nsi.hpp"
#include "nsi_dynamic.hpp"

NSI::DynamicAPI nsi_api;
NSI::Context nsi(nsi_api);
```

In both cases, a new nsi context can then be created with the `Begin()` method.

```
nsi.Begin();
```

This will be bound to the `NSI::Context` object and released when the object is deleted. It is also possible to bind the object to a handle from the C API, in which case it will not be released unless the `End()` method is explicitly called.

Argument Passing

The `NSI::Context` class has methods for all the other s calls. The optional arguments of those can be set by several accessory classes and given in many ways. The most basic is a single argument.

```
nsi.SetAttribute("handle", NSI::FloatArg("fov", 45.0f));
```

It is also possible to provide static lists:

```
nsi.SetAttribute(
    "handle",(
        NSI::FloatArg("fov", 45.0f),
        NSI::DoubleArg("depthoffield.fstop", 4.0)
    )
);
```

And finally a class supports dynamically building a list.

```
NSI::ArgumentList args;
args.Add(new NSI::FloatArg("fov", 45.0f));
args.Add(new NSI::DoubleArg("depthoffield.fstop", 4.0));
nsi.SetAttribute("handle", args);
```

The `NSI::ArgumentList` object will delete all the objects added to it when it is deleted.

Argument Classes

To be continued . . .

3.2.3 The Rust API

The *nsi* crate provides Rust wrappers for the s API. These are based on the low-level wrapper crate *nsi-sys* that contains autogenerated bindings on top of nsi.h.

Creating a Context

The core of these wrappers is the `Context` struct. Its construction triggers dynamic linking with the renderer.

```
let ctx = nsi::Context::new(None)?
```

3.2.4 The Lua API

The scripted interface is slightly different than its 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 s context to function calls since it's already embodied in the s Lua table (which is used as a class).
- The `type` argument can be omitted if the argument is an integer, real or string (as with the `Kd` and `filename` in the example below).
- s arguments can either be passed as a variable number of arguments or as a single argument representing an array of arguments (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.

Below is an example shader creation logic in Lua.

Listing 3: shader creation example in Lua

```
nsi.Create( "lambert", "shader" );
nsi.SetAttribute(
    "lambert", {
        { name = "filename", data = "lambert_material.oso" },
        { name = "Kd", data = 0.55 },
        { 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 }
    }
);
```

API calls

All (in a scripting context) useful s functions are provided and are listed below. There is also a `nsi.utilities` class which, for now, only contains a *method to print errors*.

Table 10: s functions

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

Optional function arguments format

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

- `name` – the name of the argument.
- `data` – the argument data. Either a value (integer, float or string) or an array.
- `type` – the type of the argument. Possible values are:

Table 11: Lua s argument types

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

- `arraylength` – length of the array for each element.

Here are some example of well formed arguments:

```
--[[ 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",
```

(continues on next page)

(continued from previous page)

```
        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" )
```

Evaluating a Lua script

Script evaluation is done through C, an s stream or even another Lua script. Here is an example using an s 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` argument. 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 s and Lua contexts.

Any error during script parsing or evaluation will be sent to s's error handler.

Some utilities, such as error reporting, are available through the `nsi.utilities` class.

Note: All Lua scripts are run in a sandbox in which all Lua system libraries are *disabled*.

Passing arguments to a Lua script

All arguments passed to `NSIEvaluate()` will appear in the `nsi.scriptarguments` 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.scriptarguments` table. So executing the following line in the `test.lua` script that the above snippete references:

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

Will print:

```
3.0
```

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 and are shown in .

Table 12: Lua s error codes

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

3.2.5 The Python API

The nsi.py file provides a python wrapper to the C interface. It is compatible with both Python 2.7 and Python 3.

An example of how to us it is provided in `python/examples/live_edit/live_edit.py`.

3.2.6 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].

3.2.7 Dynamic Library Procedurals

and nodes can execute code loaded from a dynamically loaded library that defines a procedural. Executing the procedural is expected to result in a series of s API calls that contribute to the description of the scene. For example, a procedural could read a part of the scene stored in a different file format and translate it directly into s calls.

This section describes how to use the definitions from the nsi_procedural.h header to write such a library in C or C++. However, the process of compiling and linking it is specific to each operating system and out of the scope of this manual.

[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.

Entry Point

The renderer expects a dynamic library procedural to contain a `NSIProceduralLoad()` symbol, which is an entry point for the library's main function:

```
struct NSIProcedural_t* NSIProceduralLoad(
    NSIContext_t ctx,
    NSIReport_t report,
    const char* nsi_library_path,
    const char* renderer_version);
```

It will be called only once per render and has the responsibility of initializing the library and returning a description of the functions implemented by the procedural. However, it is not meant to generate nsi calls.

It returns a pointer to an descriptor struct of type `NSIProcedural_t` (see *below*).

`NSIProceduralLoad()` receives the following arguments:

Table 13: NSIProceduralLoad() optional arguments

Name	Type	Description
ctx	NSIContext_t	The s context into which the procedural is being loaded.
report	NSIReport_t	A function that can be used to display informational, warning or error messages through the renderer.
nsi_library_path	const char*	The path to the s implementation that is loading the procedural. This allows the procedural to explicitly make its s API calls through the same implementation (for example, by using defined in nsi_dynamic.hpp). It's usually not required if only one implementation of s is installed on the system.
renderer_version	const char*	A character string describing the current version of the renderer.

Procedural Description

Listing 4: definition of `NSIProcedural_t`

```
typedef void (*NSIProceduralUnload_t)(
    NSIContext_t ctx,
    NSIReport_t report,
    struct NSIProcedural_t* proc);

typedef void (*NSIProceduralExecute_t)(
    NSIContext_t ctx,
    NSIReport_t report,
    struct NSIProcedural_t* proc,
    int n_args,
    const struct NSIParam_t* args);

struct NSIProcedural_t
{
    unsigned nsi_version;
    NSIProceduralUnload_t unload;
    NSIProceduralExecute_t execute;
};
```

The structure returned by `NSIProceduralLoad()` contains information needed by the renderer to use the procedural.

Note: The allocation of this structure is managed entirely from within the procedural and it will *never* be copied or modified by the renderer.

Tip: This means that it is possible for a procedural to extend the structure (by over-allocating memory or subclassing, for example) in order to store any **extra information** that it might need later.

The `nsi_version` member must be set to `NSI_VERSION` (defined in nsi.h), so the renderer is able to determine which version of s was used when compiling the procedural.

The function pointers types used in the definition are :

- `NSIProceduralUnload_t` is a function that cleans-up after the last execution of the procedural. This is the dual of `NSIProceduralLoad()`. In addition to arguments `ctx` and `report`, also received by `NSIProceduralLoad()`, it receives the description of the procedural returned by `NSIProceduralLoad()`.
- `NSIProceduralExecute_t` is a function that contributes to the description of the scene by generating s API calls. Since `NSIProceduralExecute_t` might be called multiple times in the same render, it's important that it uses the context `ctx` it receives as a argument to make its s calls, and not the context previously received by `NSIProceduralLoad()`. It also receives any extra arguments sent to , or any extra attributes set on a node. They are stored in the `args` array (of length `n_args`). `NSIParam_t` is described in .

Error Reporting

All functions of the procedural called by s receive a argument of type `NSIReport_t`. This is a pointer to a function which should be used by the procedural to report errors or display any informational message.

```
typedef void (*NSIReport_t)(
    NSIContext_t ctx, int level, const char* message);
```

It receives the current context, the error level (as described in) and the message to be displayed. This information will be forwarded to any error handler attached to the current context, along with other regular renderer messages. Using this, instead of a custom error reporting mechanism, will benefit the user by ensuring that all messages are displayed in a consistent manner.

Preprocessor Macros

Some convenient C preprocessor macros are also defined in `nsi_procedural.h` :

```
NSI_PROCEDURAL_UNLOAD(name)
```

and

```
NSI_PROCEDURAL_EXECUTE(name)
```

declare functions of the specified name that match `NSIProceduralUnload_t` and `NSIProceduralExecute_t`, respectively.

```
NSI_PROCEDURAL_LOAD
```

declares a `NSIProceduralLoad` function.

```
NSI_PROCEDURAL_INIT(proc, unload_fct, execute_fct)
```

initializes a `NSIProcedural_t` (passed as `proc`) using the addresses of the procedural's main functions. It also initializes `proc.nsi_version`.

So, a skeletal dynamic library procedural (that does nothing) could be implemented as in .

Please note, however, that the `proc` static variable in this example contains only constant values, which allows it to be allocated as a static variable. In a more complex implementation, it could have been over-allocated (or sub-classed, in C++) to hold additional, variable data[1]. In that case, it would have been better to allocate the descriptor dynamically – and release it in `NSI_PROCEDURAL_UNLOAD` – so the procedural could be loaded independently from multiple parallel renders, each using its own instance of the `NSIProcedural_t` descriptor.

```
#include "nsi_procedural.h"

NSI_PROCEDURAL_UNLOAD(min_unload)
{
}

NSI_PROCEDURAL_EXECUTE(min_execute)
{
}

NSI_PROCEDURAL_LOAD
{
    static struct NSIProcedural_t proc;
    NSI_PROCEDURAL_INIT(proc, min_unload, min_execute);
    return &proc;
}
```

[1] A good example of this is available in the *3Delight* installation, in file gear.cpp.

CHAPTER FOUR

NODES

The following sections describe available nodes in technical terms. Refer to *the rendering guidelines* for usage details.

Table 1: Overview of nsi nodes

Node	Function
root	The scene's root
global	Global settings node
set	Expresses relationships of groups of nodes
shader	s shader or layer in a shader group
attributes	Container for generic attributes (e.g. visibility)
transform	Transformation to place objects in the scene
instances	Specifies instances of other nodes
plane	An infinite plane
mesh	Polygonal mesh or subdivision surface
faceset	Assign attributes to part of a mesh, curves or paticles.
curves	Linear, b-spline and Catmull-Rom curves
particles	Collection of particles
procedural	Geometry to be loaded or generated in delayed fashion
volume	A volume loaded from an OpenVDB file
environment	Geometry type to define environment lighting
camera	Set of nodes to create viewing cameras
outputdriver	A target where to output rendered pixels
outputlayer	Describes one render layer to be connected to an `outputdriver` node
screen	Describes how the view from a camera node will be rasterized into an `outputlayer` node

4.1 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. 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 under the *transform node*.

4.2 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 does *not* need to be created using *NSICreate*. The following attributes are recognized by *3Delight*:

Table 2: global node optional attributes

Name	**Type**	**Description/Values**
`numberofthreads` `threads.count` (!)	integer	Specifies the total number of threads to use for a particular render: • A value of `0` lets the render engine choose an optimal thread value. This is 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 to decrease the total number of render threads.
`renderatlowpriority` `priority.low` (!)	integer	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.
`texturememory` `texture.memory` (!)	integer	Specifies the approximate maximum memory size, in megabytes, the renderer will allocate to accelerate texture access.
`bucketorder` `bucket.order` (!)	string	Specifies in what order the buckets are rendered. The available values are: `horizontal` — Row by row, left to right and top to bottom. This is the **default**. `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.
`frame`	integer	Provides a frame number to be used as a seed for the sampling pattern. See the *screen node*.
`lightcache`	integer (1)	Controls use of the renderer's light cache. Set this to *0* to switch the cache off. When this is switched on, each bucket is visited twice during rendering. **WARNING:** *display drivers that do not request scanline order need to make sure they handle this gracefully.*

Table 3: global node optional network cache attributes

`networkcache.size`	integer	Specifies the maximum network cache size, in gigabytes (*GB*, not *GiB*), 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.
`networkcache.write`	integer	Enables caching for image write operations. This alleviates pressure on networks by first rendering images to a local temporary location and copying them to their final destination at the end of the render. This replaces many small network writes by more efficient larger operations.

Table 4: global node optional attributes for licensing

`license.server`	string	Specifies the name or IP address of the license server to be used.
`license.wait`	integer	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 is set to `0`. The latter setting is useful when managing a renderfarm and other work could be scheduled instead.
`license.hold`	integer	By default, the renderer will get new licenses for every render and release them once it is done. This can be undesirable if several frames are rendered in sequence from the same process process. If this option is set to `1`, the licenses obtained for the first frame are held until the last frame is finished.

Table 5: global node optional attributes governing ray tracing quality

`maximumraydepth.diffuse` `diffuse.ray.depth.max` (!)	integer	Specifies the maximum bounce depth a ray emitted from a diffuse closure can reach. A depth of 1 specifies one additional bounce compared to purely local illumination.
`maximumraylength.diffuse` `diffuse.ray.length.max` (!)	double	Limits the distance a ray emitted from a diffuse closure 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 this to a negative value disables the limitation.
`maximumraydepth.reflection` `reflection.ray.depth.max` (!)	integer	Specifies the maximum bounce depth a reflection/glossy/specular ray can reach. Setting reflection depth to 0 will only compute local illumination resulting in only surfaces with an emission closure to appear in reflections.
`maximumraylength.reflection` `reflection.ray.length.max` (!)	double	Limits the distance a reflection/glossy/specular ray can travel. Setting this to a negative value disables the limitation.
`maximumraydepth.refraction` `refraction.ray.depth.max` (!)	integer	Specifies the maximum bounce depth a refraction ray can reach. The default value of 4 allows light to shine through a properly modeled object such as a glass.
`maximumraylength.refraction` `refraction.ray.length.max` (!)	double	Limits the distance a refraction ray can travel. Setting this to a negative value disables the limitation.
`maximumraydepth.hair` `hair.ray.depth.max` (!)	integer	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.
`maximumraylength.hair` `hair.ray.length.max` (!)	double	Limits the distance a hair ray can travel. Setting this to a negative value disables the limitation.
`maximumraydepth.volume` `volume.ray.depth.max` (!)	integer	Specifies the maximum bounce depth a volume ray can reach.
`maximumraylength.volume` `volume.ray.length.max` (!)	double	Limits the distance a volume ray can travel. Setting this to a negative value disables the limitation.

Table 6: global node optional attributes controlling overall image quality

`quality.shadingsamples` `shading.samples` (!)	integer	Controls the quality of BSDF sampling. Larger values give less visible noise.
`quality.volumesamples` `volume.samples` (!)	integer	Controls the quality of volume sampling. Larger values give less visible noise.
`show.displacement` `shading.displacement` (!)	integer	When set to 1, enables displacement shading. Otherwise, it must be set to *0* to ignore any displacement shader in the scene.
`show.atmosphere` `shading.atmosphere` (!)	integer	When set to 1, enables atmosphere shader(s). Otherwise, it must be set to *0* to ignore any atmosphere shader in the scene.
`show.multiplescattering` `shading.multiplescattering` (!)	double	This is a multiplier on the multiple scattering of VDB nodes. This parameter is useful to obtain faster draft renders by lowering the value below 1. The range is *0* to *1*.
`show.osl.subsurface` `shading.osl.subsurface` (!)	integer	When set to 1, enables the `subsurface()` s closure. Otherwise, it must be set to `0`, which will ignore this closure in s shaders.

For anti-aliasing quality see the *screen node*.

Table 7: global node optional attributes for statistics

Name | Type | Description/Values		
`statistics.progress`	integer	When set to 1, prints rendering progress as a percentage of completed pixels.
`statistics.filename`	string	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.

4.3 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 an *outputlayer*'s `lightset` attribute (see also *light layers*).

It has the following attributes:

Table 8: set node optional attributes

Name	Type	Description/Values
`members` `member` (!)	«connection(s)»	This connection accepts all nodes that are members of the set.

4.4 The Plane Node

This node represents an infinite plane, centered at the origin and pointing towards Z+. It has no required attributes. The UV coordinates are defined as the X and Y coordinates of the plane.

4.5 The Mesh Node

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

Table 9: mesh node required attributes

Name	Type	Description/Values
`P`	point	The positions of the object's vertices. Typically, this attribute will be indexed through a `P.indices` attribute.
`nvertices` `vertex.count` (!) `face.vertex.count` (!)	integer	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).

To render a mesh as a subdivision surface, at least the `subdivision.scheme` argument must be set. When rendering as a subdvision surface, the mesh node accepts these optionalattributes:

Table 10: mesh node as subdivision surface optional attributes

Name	Type	Description/Values
`subdivision.scheme`	string	A value of `"catmull-clark"` will cause the mesh to render as a Catmull-Clark subdivision surface.
`subdivision.cornervertices` `subdivision.corner.index` (!)	integer	A list of vertices which are sharp corners. The values are indices into the `P` attribute, like `P.indices`.
`subdivision.cornersharpness` `subdivision.corner.sharpness` (!)	float	The sharpness of each specified sharp corner. It must have a value for each value given in `subdivision.cornervertices`.
`subdivision.smoothcreasecorners` `subdivision.corner.automatic` (!)	integer	This tag requires a single integer argument with a value of 1 or `0` indicating whether or not the surface uses enhanced subdivision rules on vertices where *more than two* creased edges meet. With a value of 1 (**the default**) the vertex is subdivided using an extended crease vertex subdivision rule which yields a *smooth* crease. With a value of 0 the surface uses enhanced subdivision rules where a vertex *becomes a sharp corner* when it has more than two incoming creased edges. Note that sharp corners can still be explicitly requested using the `subdivision.corner.index` & `subdivision.corner.sharpness` tags.
`subdivision.creasevertices` `subdivision.crease.index` (!)	integer	A list of crease edges. Each edge is specified as a pair of indices into the `P` attribute, like `P.indices`.
`subdivision.creasesharpness` `subdivision.crease.sharpness` (!)	float	The sharpness of each specified crease. It must have a value for each pair of values given in `subdivision.creasevertices`.

The mesh node also has these optional attributes:

Table 11: mesh node optional attributes

Name	Type	Description/Values
`nholes` `hole.count` (!)	integer	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. The example below shows the definition of a polygon mesh consisting of three square faces, with one triangular hole in the first one and square holes in the second one.
`clockwisewinding` `clockwise` (!)	integer	A value of 1 specifies that polygons with clockwise winding order are front facing. **The default** is `0`, making counterclockwise polygons front facing.

Below is a sample s stream snippet showing the definition of a mesh with holes.

```
Create "holey" "mesh"
SetAttribute "holey"
```

(continues on next page)

(continued from previous page)

```
  "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 no 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 ]
```

4.6 The Faceset Node

This node is used to provide a way to attach attributes to parts of another geometric primitive, such as faces of a *mesh*, curves or particles. It has the following attributes:

Table 12: faceset node attributes

Name	Type	Description/Values
`faces` `face.index` (!)	integer	A list of indices of faces. It identifies which faces of the original geometry will be part of this face set.

```
Create "subdiv" "mesh"
SetAttribute "subdiv"
  "nvertices" "int" 4 [ 4 4 4 4 ]
  "P" "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"
```

4.7 The Curves Node

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

Table 13: curves node required attributes

Name	Type	Description/Values
`nverts` `vertex.count` (!)	integer	The number of vertices for each curve. This must be at least 4 for cubic curves and 2 for linear curves. 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.

It also has these optional attributes:

Table 14: curves node optional attributes

<table>
<tr><th>Name</th><th>Type</th><th colspan="2">Description/Values</th></tr>
<tr><td rowspan="5"><code>basis</code></td><td rowspan="5">string</td><td colspan="2">The basis functions used for curve interpolation. Possible choices are:</td></tr>
<tr><td><code>b-spline</code></td><td>B-spline interpolation.</td></tr>
<tr><td><code>catmull-rom</code></td><td>Catmull-Rom interpolation. This is the default value.</td></tr>
<tr><td><code>linear</code></td><td>Linear interpolation.</td></tr>
<tr><td><code>hobby</code> (!)</td><td>Hobby interpolation.</td></tr>
<tr><td><code>N</code></td><td>normal</td><td colspan="2">The presence of a normal indicates that each curve is to be rendered as an oriented ribbon. The orientation of each ribbon is defined by the provided normal which can be constant, a per-curve or a per-vertex attribute. Each ribbon is assumed to always face the camera if a normal is not provided.</td></tr>
<tr><td><code>extrapolate</code></td><td>integer</td><td colspan="2">By default, when this is set to <code>0</code>, cubic curves will not be drawn to their end vertices as the basis functions require an extra vertex to define the curve. If this attribute is set to <code>1</code>, an extra vertex is automatically extrapolated so the curves reach their end vertices, as with linear interpolation.</td></tr>
</table>

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: A single curve is considered a face as far as use of *NSIParamPerFace* is concerned. See also the *faceset node*.

4.8 The Particles Node

This geometry node represents a collection of *tiny* particles. Particles are represented 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).

Table 15: particles node required attributes

Name	Type	Description/Values
P	point	The center of each particle.
width	float	The width of each particle. It can be specified for the entire particles node (only one value provided), per-particle or indirectly through a width.indices attribute.

It also has these optional attributes:

Table 16: particles node optional attributes

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	integer	This attribute has to be the same length as P. It 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 is mandatory by definition.

4.9 The Procedural Node

This node acts as a proxy for geometry that could be defined at a later time than the node's definition, using a procedural supported by . Since the procedural is evaluated in complete isolation from the rest of the scene, it can be done either lazily (depending on its boundingbox attribute) or in parallel with other procedural nodes.

The procedural node supports, as its attributes, all the arguments of the *NSIEvaluate* API call, meaning that procedural types accepted by that api call (s archives, dynamic libraries, Lua scripts) are also supported by this node. Those attributes are used to call a procedural that is expected to define a sub-scene, which has to be independent from the other nodes in the scene. The procedural node will act as the sub-scene's local root and, as such, also supports all the attributes of a regular node. In order to connect the nodes it creates to the sub-scene's root, the procedural simply has to connect them to the regular .root.

In the context of an *interactive render*, the procedural will be executed again after the node's attributes have been edited. All nodes previously connected by the procedural to the sub-scene's root will be deleted automatically before the procedural's re-execution.

Additionally, this node has the following optional attribute :

Table 17: procedural node optional attribute

Name	Type	Description/Values
boundingbox	point[2]	Specifies a bounding box for the geometry where boundingbox[0] and boundingbox[1] correspond, respectively, to the 'minimum' and the 'maximum' corners of the box.

4.10 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 for more practical details). The following optional node attribute is recognized:

Table 18: environment node optional attribute

Name	Type	Description/Values
`angle`	double	Specifies the cone angle representing the region of the sphere to be sampled. The angle is measured around the Z+ axis. If the angle is set to 0, the environment describes a directional light. See *the guidelines* for more information on about how to specify light sources.

Tip: To position the environment dome one must connect the node to a *transform node* and apply the desired rotation(s).

4.11 The Shader Node

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

Table 19: shader node attributes

Name	Type	Description/Values
`shaderfilename`	string	This is the name of the file which contains the shader's compiled code.
`shaderobject`	string	This contains the complete compiled shader code. It allows specifying shaders without going through files.

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

4.12 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 nodes (to build). Attribute values are gathered along the path starting from the geometric primitive, through all the transform nodes it is connected to, until the 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:

Table 20: attributes node attributes

Name	Type	Description/Values
`surfaceshader` `shader.surface` (!)	«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` `shader.displacement` (!)	«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` `shader.volume` (!)	«connection»	The *shader node* which will be used to shade the volume inside the primitive is connected to this attribute.
`ATTR.priority`	integer	Sets the priority of attribute `ATTR` when gathering attributes in the scene hierarchy.

`visibility.camera`
`visibility.diffuse`
`visibility.hair`
`visibility.reflection`
`visibility.refraction`
`visibility.shadow`
`visibility.specular`
`visibility.volume`
	integer	These attributes set visibility for each ray type specified in s. 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`	integer	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`	integer	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.
`regularemission` `emission.regular` (!)	integer	If this is set to 1, closures not used with `quantize()` will use emission from the objects affected by the attribute. If set to 0, they will not.
`quantizedemission` `emission.quantized` (!)	integer	If this is set to 1, quantized closures will use emission from the objects affected by the attribute. If set to `0`, they will not.
`bounds` `boundary`	«connection»	When a geometry node (usually a *mesh node*) is connected to this attribute, it will be used to restrict the effect of the attributes node, which will apply only inside the volume defined by the connected geometry object.

4.13 The Transform Node

This node represents a geometric transformation. Transform nodes can be chained together to express transform concatenation, hierarchies and instances.

A transform node also accepts attributes to implement *hierarchical attribute assignment and overrides*.

It has the following attributes:

Table 21: transform node attributes

Name	Type	Description/Values
`tranformationmatrix` `matrix` (!)	dou- blema- trix	This is a 4×4 matrix which describes the node's transformation. Matrices in s *post-multiply* so column vectors are of the form:
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		

`objects`

`object` (!) «connection(s)» This is where the transformed objects are connected to. This includes geometry nodes, other transform nodes and camera nodes.

`geometryattributes`

`attribute` (!) «connection(s)» This is where *attributes nodes* may be connected to affect any geometry transformed by this node.

See the guidelines on *attributes* and *instancing* for explanations on how this connection is used.

4.14 The Instances Node

This node is an efficient way to specify a large number of instances. It has the following attributes:

Table 22: instances node attributes

Name	Type	Description/Values
`sourcemodels` `object` (!)	«con- nec- tion(s)»	The instanced models should connect to this attribute. Connections must have an integer `index` attribute if there are several, so the models effectively form an ordered list.
`transformationmatrices` `matrix` (!)	dou- blema- trix	A transformation matrix for each instance.

Table 23: instances node optional attributes

`modelindices` `object.index` (!)	integer	An optional model selector for each instance.
`disabledinstances` `disable.index` (!)	[inte- ger; …]	An optional list of indices of instances which are not to be rendered.

4.15 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 node. It has the following attributes:

Table 24: outputdriver node attributes

Name	Type	Description/Values
`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`	integer	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.

4.16 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:

Table 25: outputlayer node attributes

Name	Type	Description/Values
`variablename`	string	This is the name of a variable to output.
`variablesource`	string	Indicates where the variable to be output is read from. Possible values are: `shader` – computed by a shader and output through an s closure s (such a `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.camera`, `P.world`).
`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	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. `half` – IEEE 754 half-precision binary floating point (binary16). `float` – IEEE 754 single-precision binary floating point (binary32).
`layertype`	string	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.

continues on next page

Table 25 – continued from previous page

<table>
<tr><th>Name</th><th>Type</th><th>Description/Values</th></tr>
<tr><td><code>dithering</code></td><td>integer</td><td>If set to 1, dithering is applied to integer scalars. Otherwise, it must be set to 0.
It is sometimes desirable to turn off dithering, for example, when outputting object IDs.</td></tr>
<tr><td><code>withalpha</code></td><td>integer</td><td>If set to 1, an alpha channel is included in the output layer. Otherwise, it must be set to <code>0</code>.</td></tr>
<tr><td><code>sortkey</code></td><td>integer</td><td>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 <code>sortkey</code> attribute appear first.</td></tr>
<tr><td><code>lightset</code></td><td>«connection(s)»</td><td>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.</td></tr>
<tr><td><code>outputdrivers</code>
<code>outputdriver</code>
(!)</td><td>«connection(s)»</td><td>This connection accepts nodes to which the layer's image will be sent.</td></tr>
<tr><td><code>filter</code></td><td>string
(<code>blackman-harris</code>)</td><td>The type of filter to use when reconstructing the final image from sub-pixel samples. Possible values are:
<code>box</code>
<code>triangle</code>
<code>catmull-rom</code>
<code>bessel</code>
<code>gaussian</code>
<code>sinc</code>
<code>mitchell</code>
<code>blackman-harris</code> (default)
<code>zmin</code>
<code>zmax</code>
<code>cryptomattelayer%u</code> Take two values from those present in each pixel's samples.</td></tr>
<tr><td><code>filterwidth</code></td><td>double</td><td>Diameter in pixels of the reconstruction filter. It is ignored when filter is <code>box</code> or <code>zmin</code>.<table><tr><td>Filter</td><td>Suggested Width</td></tr><tr><td><code>box</code></td><td><code>1.0</code></td></tr><tr><td><code>triangle</code></td><td><code>2.0</code></td></tr><tr><td><code>catmull-rom</code></td><td><code>4.0</code></td></tr><tr><td><code>bessel</code></td><td><code>6.49</code></td></tr><tr><td><code>gaussian</code></td><td><code>2.0–2.5</code></td></tr><tr><td><code>sinc</code></td><td><code>4.0–8.0</code></td></tr><tr><td><code>mitchell</code></td><td><code>4.0–5.0</code></td></tr><tr><td><code>blackman-harris</code></td><td><code>3.0–4.0</code></td></tr></table></td></tr>
<tr><td><code>backgroundvalue</code></td><td>float</td><td>The value given to pixels where nothing is rendered.</td></tr>
</table>

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

4.17 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*.

For an exmplanation of coordinate systems/spaces mentioned below, e.g. `NDC`, `screen`, etc., please refer to the Open Shading Language specification

Table 26: screen node attributes

Name	Type	Description/Values
`outputlayers` `outputlayer` (!)	«connection(s)»	This connection accepts 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][2]	The region of the image to be rendered. It is defined by a two 2D coordinates. Each represents a point in *NDC* space: • `Top-left` corner of the crop region. • `Bottom-right` corner of the crop region.
`prioritywindow`	integer[2][2]	For progressive renders, this is the region of the image to be rendered first. It is defined by two coordinates. Each represents a pixel position in `raster` space: • `Top-left` corner of the high priority region. • `Bottom-right` corner of the high priority region.
`screenwindow`	double[2][2]	Specifies the screen space region to be rendered. It is defined by two coordinates. Each represents a point in `screen` space: • `Top-left` corner of the region. • `Bottom-right` corner of the region. Note that the default screen window is set implicitely by the frame aspect ratio:
-1		

$$\begin{bmatrix} f & 1 \end{bmatrix} \text{ for } f = \frac{xres}{yres}$$

`pixelaspectratio` float (1) Ratio of the physical width to the height of a single pixel. A value of 1 corresponds to square pixels.

`staticsamplingpattern` integer (`0`) This controls whether or not the sampling pattern used to produce the image changes for every frame.

A nonzero value will cause the same pattern to be used for all frames. A value of `0` will cause the pattern to change with the frame attribute of the global node.

4.18 The Volume Node

This node represents a volumetric object defined by OpenVDB data. It has the following attributes:

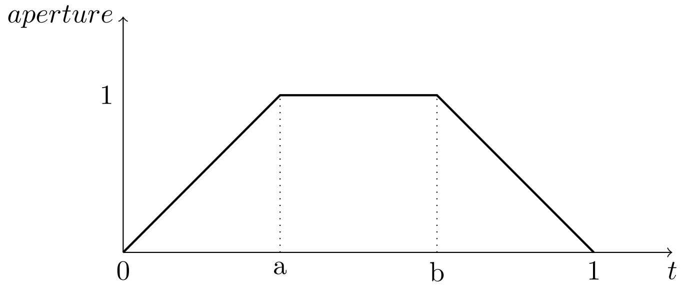
Table 27: volume node attributes

Name	Type	Description/Values
`vdbfilename` `filename` (!)	string	The path to an OpenVDB file with the volumetric data.
`colorgrid`	string	The name of the OpenVDB grid to use as a scattering color multiplier for the volume shader.
`densitygrid`	string	The name of the OpenVDB grid to use as volume density for the volume shader.
`emissionintensitygrid`	string	The name of the OpenVDB grid to use as emission intensity for the volume shader.
`temperaturegrid`	string	The name of the OpenVDB grid to use as temperature for the volume shader.
`velocitygrid`	double	The name of the OpenVDB grid to use as motion vectors. This can also name the first of three scalar grids (i.e. "velocityX").
`velocityscale`	double (1)	A scaling factor applied to the motion vectors.

4.19 Camera Nodes

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

Table 28: camera nodes shared attributes

Name	Type	Description/Values
`screens` `screen` (!)	«connection(s)»	This connection accepts nodes which will rasterize an image of the scene as seen by the camera. Refer to for more information.
`shutterrange`	double[2]	Time interval during which the camera shutter is at least partially open. It is defined by a list of exactly two values: • Time at which the shutter starts **opening**. • Time at which the shutter finishes **closing**.
`shutteropening`	double[2]	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. This feature simulates a mechanical shutter on which open and close movements are not instantaneous. Below is an image showing the geometry of such a trapezoid filter.  Fig. 1: An example shutter opening configuration with $a = \frac{1}{3}$ and $b = \frac{2}{3}$.
`clippingrange`	double[2]	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.
`lensshader`	«connection»	An s shader through which camera rays get sent. See *lens shaders*.

4.19.1 The Orthographiccamera Node

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

4.19.2 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 node for camera placement. It has the following attributes:

Table 29: perspective node attributes

Name	Type	Description/Values
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	Vertical focal length, in scene units, of the camera lens.
depthoffield.focallengthratio	double (1.0)	Ratio of vertical focal length to horizontal focal length. This is the squeeze ratio of an anamorphic 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	double (0)	A rotation angle (in degrees) to be applied to the camera's aperture, in the image plane.

Table 30: perspective node extra attributes

depthoffield.aperture.roundness	double (0)	This shapes the sides of the polygon When set to 0, the aperture is polygon with flat sides. When set to 1, the aperture is a perfect circle. When set to -1, the aperture sides curve inwards.
depthoffield.aperture.density	double (0)	The slope of the aperture's density. A value of 0 gives uniform density. Negative values, up to -1, make the aperture brighter near the center. Positive values, up to 1, make it brighter near the edge.
depthoffield.aperture.aspectratio	double (1)	Circularity of the aperture. This can be used to simulate anamorphic lenses.

4.19.3 The Fisheyecamera Node

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

Table 31: fisheye camera node attributes

<table>
<tr><th>Name</th><th>Type</th><th colspan="2">Description/Values</th></tr>
<tr><td>fov</td><td>float</td><td colspan="2">The field of view angle, in degrees.</td></tr>
<tr><td rowspan="5">mapping</td><td rowspan="5">string (equidistant)</td><td colspan="2">Defines one of the supported fisheye mapping functions. Possible values are:</td></tr>
<tr><td>equidistant</td><td>Maintains angular distances.</td></tr>
<tr><td>equisolidangle</td><td>Every pixel in the image covers the same solid angle.</td></tr>
<tr><td>orthographic</td><td>Maintains planar illuminance. This mapping is limited to a 180 field of view.</td></tr>
<tr><td>stereographic</td><td>Maintains angles throughout the image. Note that stereographic mapping fails to work with field of views close to 360 degrees.</td></tr>
</table>

4.19.4 The Cylindricalcamera Node

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

Table 32: cylindrical camera nodes shared attributes

Name	Type	Description/Values
`fov`	float (`90`)	Specifies the *vertical* field of view, in degrees. The default value is 90.
`horizontalfov` `fov.horizontal` (!)	float (`360`)	Specifies the horizontal field of view, in degrees. The default value is 360.
`eyeoffset`	float	This allows to render stereoscopic cylindrical images by specifying an eye offset

4.19.5 The Sphericalcamera Node

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

4.19.6 Lens Shaders

A lens shader is an s 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

FIVE

SCRIPT OBJECTS

It is a design goal to provide an easy to use and flexible scripting language for s.

The Lua language has been selected for such a task because of its performance, lightness and features[#]_. 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 s.

Note: Although they go hand in hand, scripting objects are not to be confused with the Lua binding.

The binding allows for calling s 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 ...

CHAPTER

SIX

RENDERING GUIDELINES

6.1 Basic Scene Anatomy

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

1. Geometry linked to the `.root` node, usually through a transform chain.
2. s materials linked to scene geometry through an *attributes* node.
3. At least one *output-driver→outputlayer→screen→camera→*`.root` chain to describe a view and an output device.

The scene graph in 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.

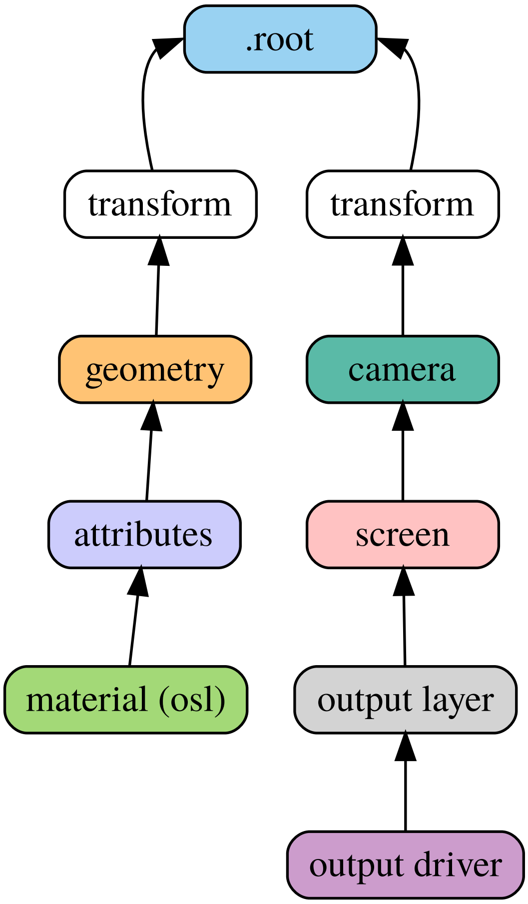


Fig. 1: The fundamental building blocks of an s scene

> **Caution:** For the scene to be visible, at least one of the materials has to be *emissive*.

6.2 A Word – or Two – About Attributes

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). E.g parameters passed to RenderMan Interface calls to build certain objects. For example, knot vectors passed to `RiNuPatch()`.

In s things are simpler and all attributes are set through the `NSISetAttribute()` 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.

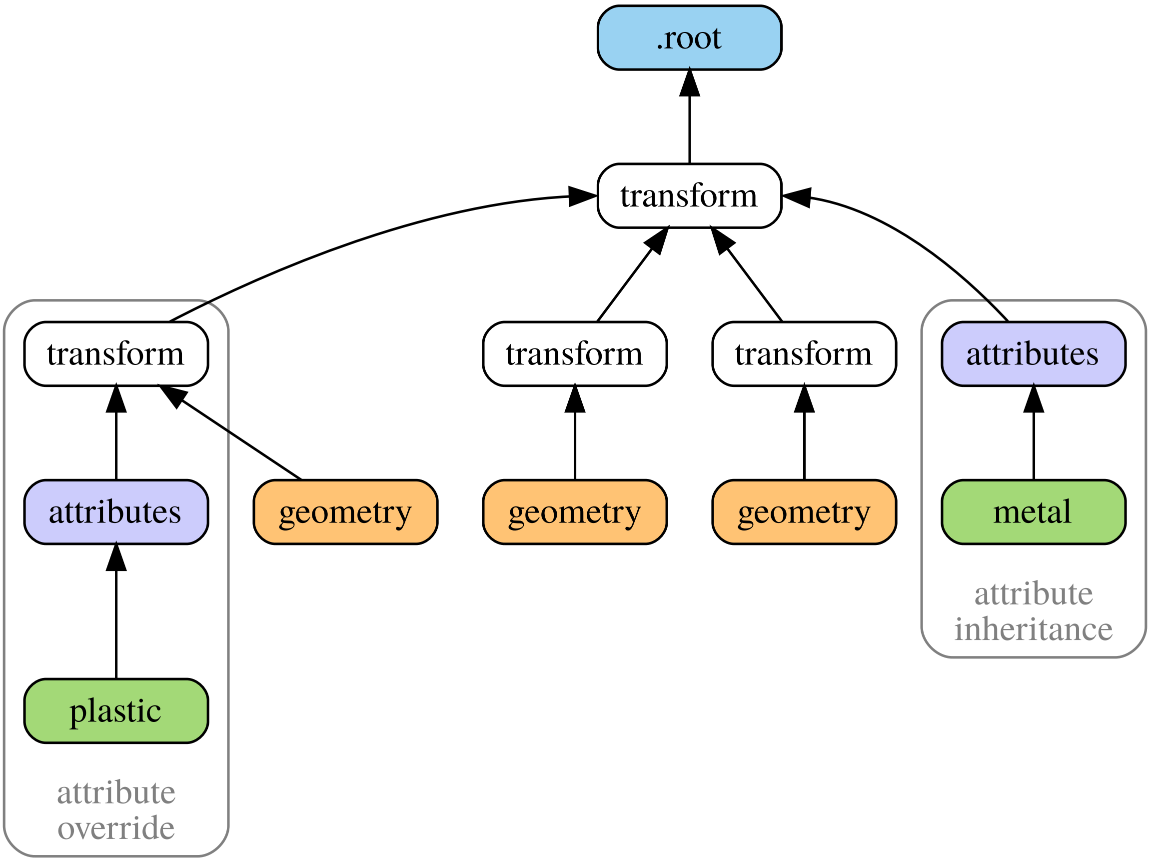


Fig. 2: Attribute inheritance and override

Note: In this documentation, all intrinsic attributes are documented at the beginning of each section describing a particular node.

In s 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). shows two features of attribute assignment in s:

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

Attribute override It is possible to override attributes for a specific geometry by attaching them to a *transform* node 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.

6.3 Instancing

Instancing in s is naturally performed by connecting a geometry to more than one transform (connecting a geometry node into a `transform.objects` attribute).

The above figure 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 *the attributes section*. Note that transforms can also be instanced and this allows for *instances of instances* using the same semantics.

6.4 Creating s Networks

The semantics used to create s 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 arguments is done using `NSIConnect`. Above diagran depicts a simple s 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*).

  Note: There is an exception to this: any non-shader node can be connected to a string attribute of a shader node. This will result in the non-shader node's handle being used as the string's value.

  This behavior is useful when the shader needs to refer to another node, in a s call to `transform()` or `getattribute()`, for example.

- There is no *symbolic linking* between shader arguments and geometry attributes (a.k.a. primvars). One has to explicitly use the `getattribute()` s function to read attributes attached to geometry. In this is done in the `read_attribute` node (*Lines 11–14*). Also see the section on *attributes*.

```
1 Create "ggx_metal" "shader"
2 SetAttribute "ggx"
3     "shaderfilename" "string" 1  ["ggx.oso"]
4
5 Create "noise" "shader"
6 SetAttribute "noise"
```

(continues on next page)

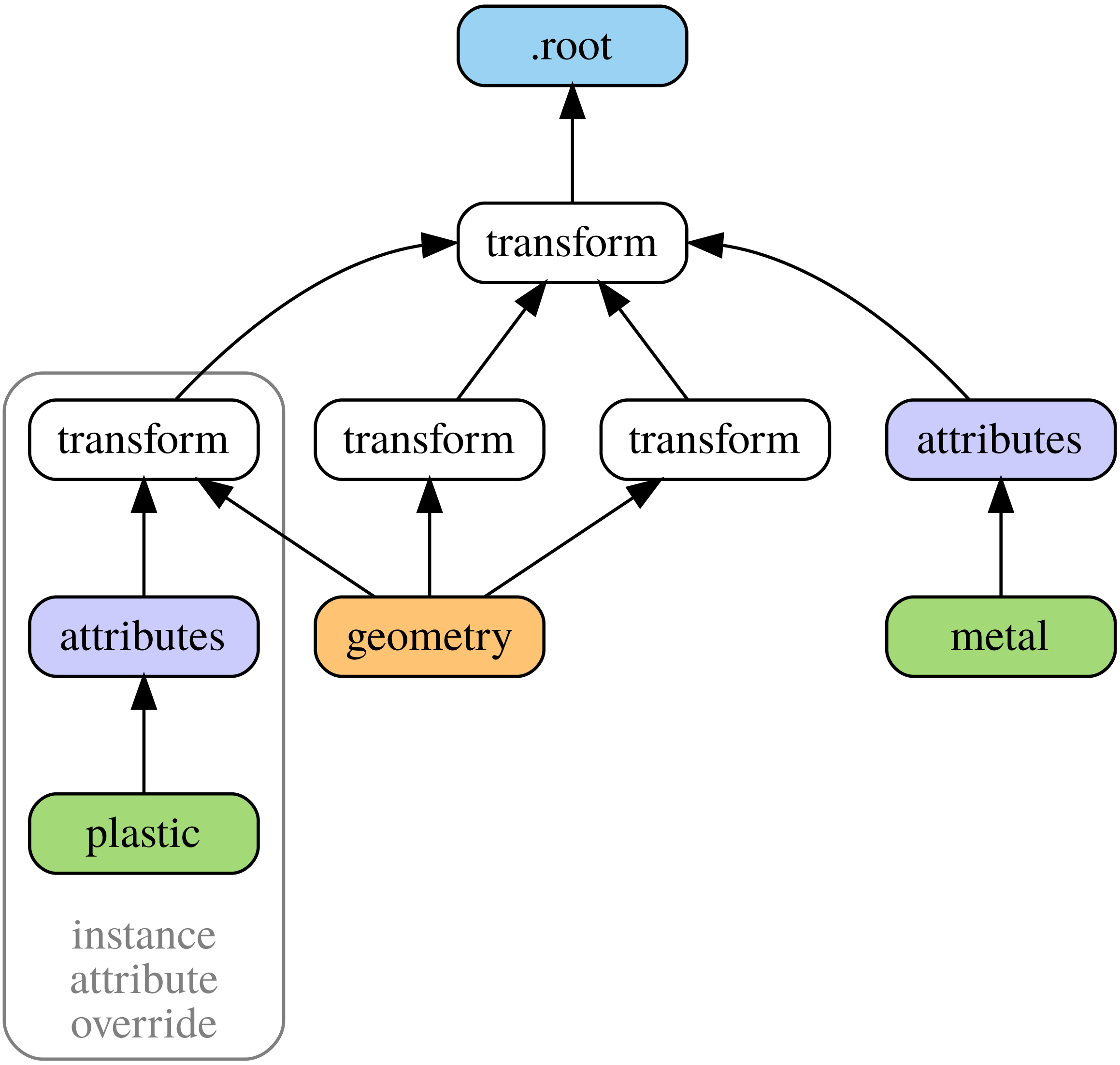


Fig. 3: Instancing in s with attribute inheritance and per-instance attribute override

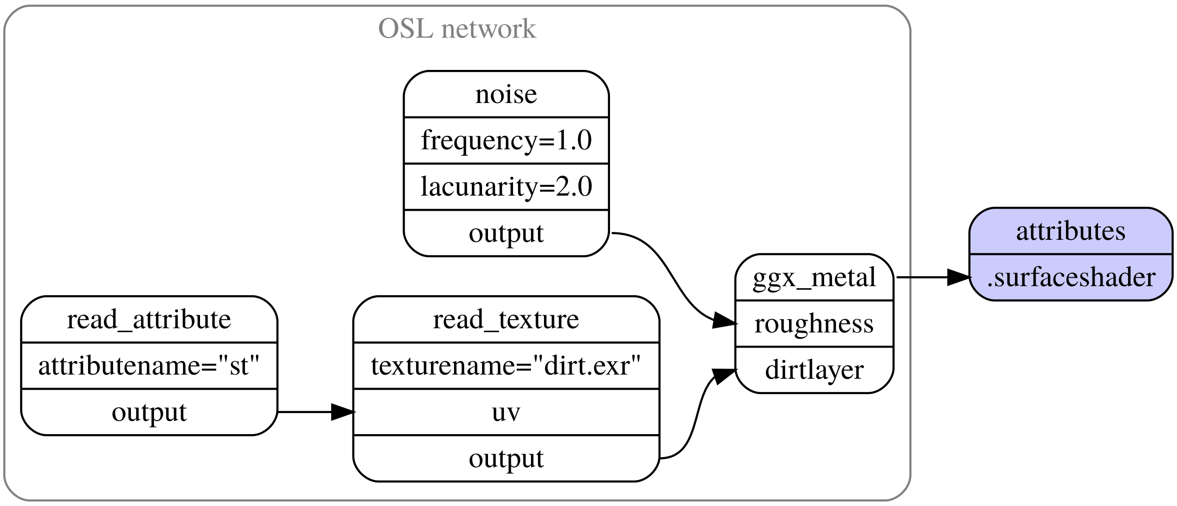


Fig. 4: A simple s network connected to an attributes node

(continued from previous page)

```
    "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"
```

6.5 Lighting in the Nodal Scene Interface

There are no special light source nodes in s (although the node, which defines a sphere of infinite radius, could be considered 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*.

Following is a quick summary on how to create different kinds of light in s.

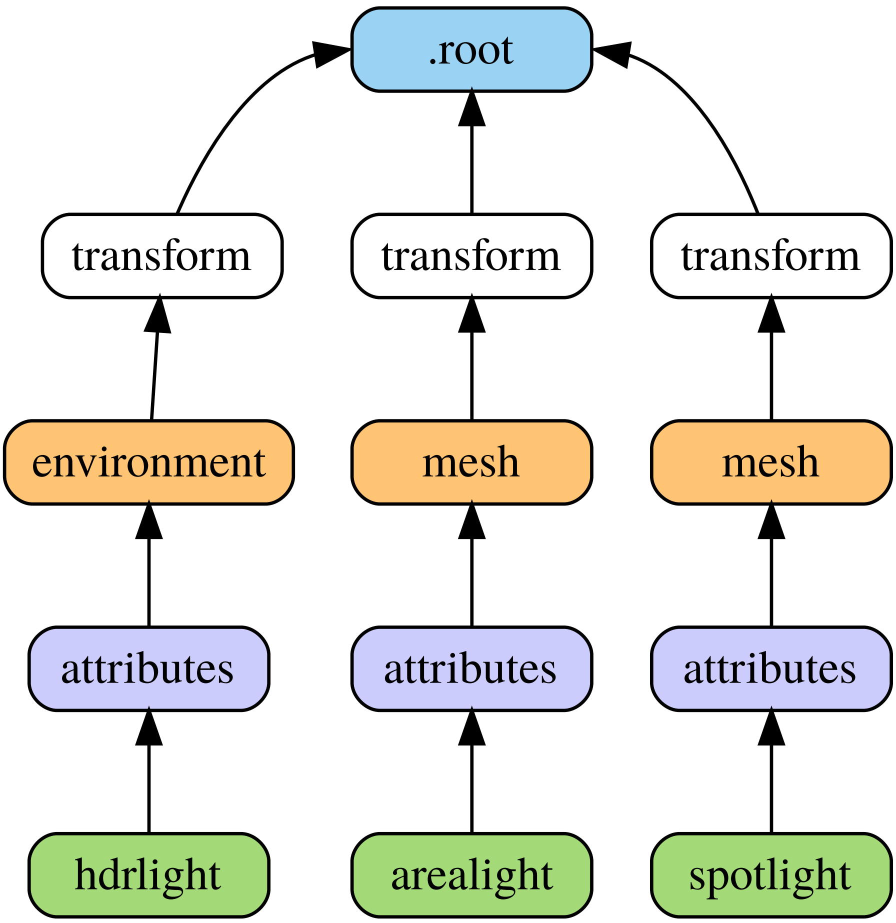


Fig. 5: Creating lights in nsi

6.5.1 Area Lights

Area lights are created by attaching an emissive surface material to geometry. Below is a simple s shader for such lights (standard s emitter).

Listing 1: Example emitter for area lights

```
// 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();
```

(continues on next page)

(continued from previous page)

```
}
```

6.5.2 Spot and Point Lights

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

Listing 2: An example OSL spot light shader

```
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();
}
```

6.5.3 Directional and HDR Lights

Directional lights are created by using the node and setting the `angle` attribute to 0. HDR lights are also created using the environment node, albeit with a 2 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 node defines a sphere of infinite radius any connected s shader must only rely on the `I` variable and disregard `P`, as is shown below.

Listing 3: An example OSL shader to do HDR lighting

```
shader hdrlight(
    string texturename = ""
) {
```

(continues on next page)

(continued from previous page)

```
    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);
}
```

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 node.

6.6 Defining Output Drivers and Layers

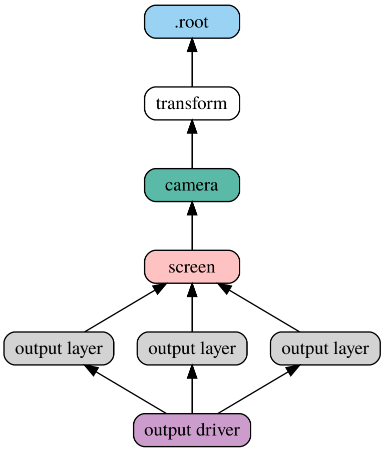


Fig. 6: s graph showing the image output chain

s 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)

depicts a s 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 : a left and right cameras are used to drive two file outputs, each having two layers (`Ci` and `Diffuse` colors).

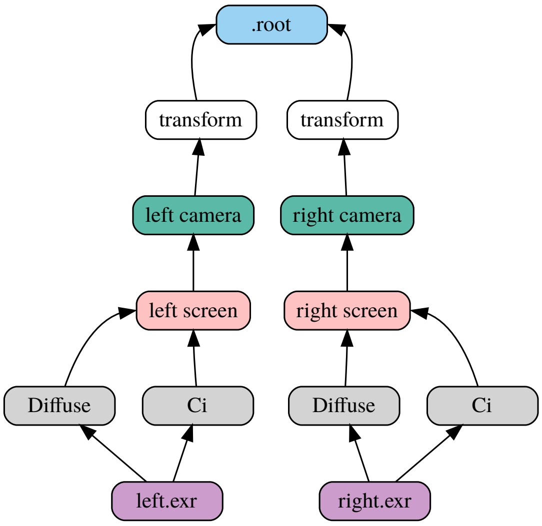


Fig. 7: s graph for a stereo image output

6.7 Light Layers

The ability to render a certain set of lights per output layer has a formal workflow in s. 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

Above diagram a scene using method 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` ().

6.8 Inter-Object Visibility

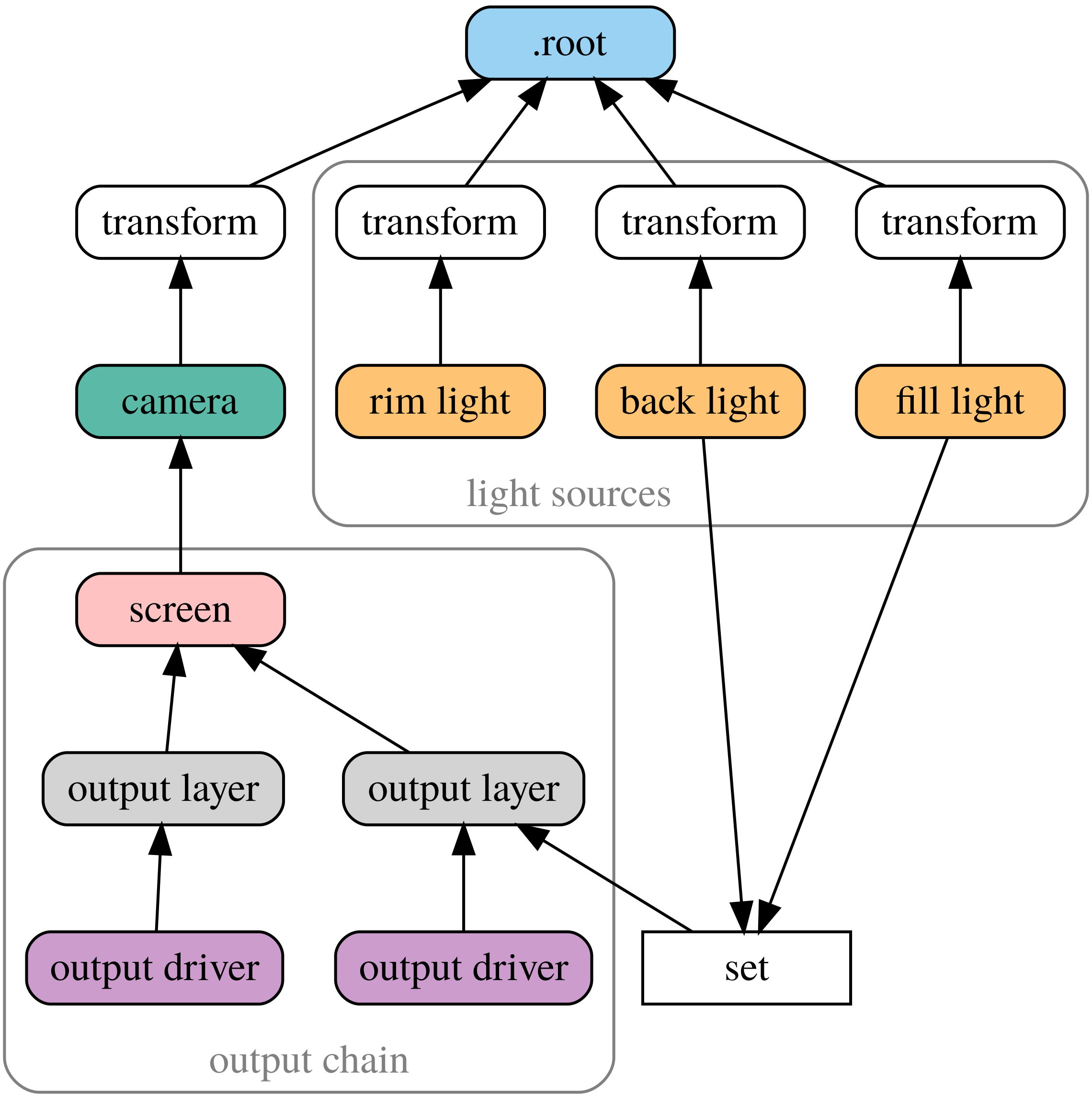
Some common rendering features are difficult to achieve using attributes and hierarchical tree structures. One such example is inter-object visibility in a 3D scene. A special case of this feature is *light linking* which allows the artist to select which objects a particular light illuminates, or not.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

In s 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 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.

Above figure shows 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]
```



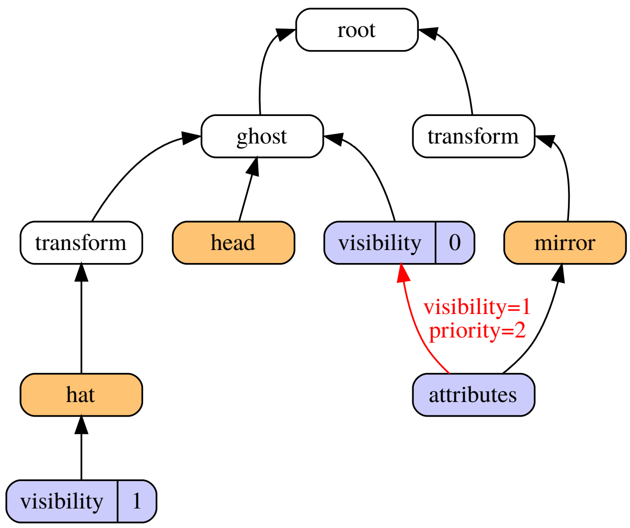


Fig. 8: Visibility override, both hierarchically and inter-object

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.

CHAPTER

SEVEN

COOKBOOK

The Nodal Scene Interface (NSI) is a simple yet expressive API to describe a scene to a renderer. From geometry declaration, to instancing, to attribute inheritance and shader assignments, everything fits in 12 API calls. The following recipes demonstrate how to achieve most common manipulations.

7.1 Geometry Creation

Creating geometry nodes is simple. The content of each node is filled using the *NSISetAttribute* call.

Listing 1: **Geometry Creation**

```
## Polygonal meshes can be created minimally by specifying "P".
## NSI's C++ API provides an easy interface to pass parameters to all NSI
## API calls through the Args class.

Create "simple polygon" "mesh"
SetAttribute "simple polygon"
    "P" "point" 1 [ -1  1  0   1  1  0   1 -1  0   -1 -1  0 ]
```

Geometry Creation in C++

```
/*
    Polygonal meshes can be created minimally by specifying "P".
    NSI's C++ API provides an easy interface to pass parameters
    to all NSI API calls through the Args class.
*/
const char *k_poly_handle = "simple polygon"; /* avoids typos */

nsi.Create( k_poly_handle, "mesh" );

NSI::ArgumentList mesh_args;
float points[3*4] = { -1, 1, 0,  1, 1, 0, 1, -1, 0, -1, -1, 0 };
mesh_args.Add(
    NSI::Argument::New( "P" )
        ->SetType( NSITypePoint )
        ->SetCount( 4 )
        ->SetValuePointer( points ) );
nsi.SetAttribute( k_poly_handle, mesh_args );
```

Specifying normals and other texture coordinates follows the same logic. Constant attributes can be declared in a concise form too:

Listing 2: **Adding constant attributes**

```
SetAttribute "simple polygon"
    "subdivision.scheme" "string" 1 ["catmull-clark"]
```

Adding constant attributes in C++

```
/** Turn our mesh into a subdivision surface */
nsi.SetAttribute( k_poly_handle,
    NSI::CStringPArg("subdivision.scheme", "catmull-clark") );
```

7.2 Transforming Geometry

In NSI, a geometry is rendered only if connected to the scene's root (which has the special handle ".root"). It is possible to directly connect a geometry node (such as the simple polygon above) to scene's root but it wouldn't be very useful. To place/instance a geometry anywhere in the 3D world a transform node is used as in the code snippet below.

Listing 3: **Adding constant attributes**

```
Create "my translation" "transform"
Connect "translation"  "" ".root" "objects"
Connect "simple polygon" "" "translation" "objects" );

# Transalte 1 unit in Y
SetAttribute "my translation"
    "transformationmatrix" "matrix" 1 [
    1 0 0 0
    0 1 0 0
    0 0 1 0
    0 1 0 1]
```

Adding constant attributes in C++

```
const char *k_instance1 = "my translation";

nsi.Create( k_instance1, "transform" );
nsi.Connect( k_instance1, "", NSI_SCENE_ROOT, "objects" );
nsi.Connect( k_poly_handle, "", k_instance1, "objects" );

/*
    Matrices in NSI are in double format to allow for greater
    range and precision.
*/
double trs[16] =
{
    1., 0., 0., 0.,
    0., 1., 0., 0.,
    0., 0., 1., 0.,
    0., 1., 0., 1. /* transalte 1 unit in Y */
};

nsi.SetAttribute( k_instance1,
    NSI::DoubleMatrixArg("transformationmatrix", trs) );
```

Instancing is as simple as connecting a geometry to different attributes. Instances of instances do work as expected too.

```
const char *k_instance2 = "another translation";
trs[13] += 1.0; /* translate in Y+ */

nsi.Create( k_instance2, "transform" );
nsi.Connect( k_poly_handle, "", k_instance2, "objects" );
nsi.Connect( k_instance2, "", NSI_SCENE_ROOT, "objects" );

/* We know have two instances of the same polygon in the scene */
```

CHAPTER EIGHT

ACKNOWLEDGEMENTS

Many thanks to John Haddon, Daniel Dresser, David Minor, Moritz Mœller 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 s (e.g. adoption of the `screen` node). Jordan Thistlewood opened the way for the first integration of s into a commercial plug-in. Stefan Habel did a thorough proofreading of the entire document and gave many suggestions.

The s logo was designed by Paolo Berto.

CHAPTER

NINE

INDEX

INDEX

O

P

Q

R

S

T

V

W