Quick Tour

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 subsections demonstrate how to achieve most common manipulations.

Geometry Creation

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

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 );
```

NSI Stream

```
## 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 ]
```

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

C++

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

NSI Stream

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

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 *transf orm* node is used as in the code snippet below.

```
const char *k_instance1 = "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) );
```

NSI Stream

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

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

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

Instancing is as simple as connecting a geometry to different attributes (yes, instances of instances are possible).

```
const char *k_instance2 = "more 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 */
```

Assigning Shaders

Shaders are created as any other nodes using the `NSICreate` API call. They are not assigned directly on geometries but through an intermediate attributes nodes. Having an extra indirection allows for more flexible export as we will see in the following chapters.

```
/**
        Create a simple shader node using the standard OSL "emitter" shader.
        Set it's parameter to something different than defaults.
*/
nsi.Create( "simpleshader", "shader" );
float red[3] = {1,0,0};
nsi.SetAttribute( "simpleshader",
        (
                NSI::CStringPArg("shaderfilename", "emitter"),
                NSI::ColorArg( "Cs", red),
                NSI::FloatArg( "power", 4.f )
        ) );

/** Create an attributes nodes and connect our shader to it */
nsi.Create( "attr", "attributes" );
nsi.Connect( "simpleshader", "", "attr", "surfaceshader" );

/* Connecting the attributes node to the mesh assign completes the assignment */
nsi.Connect( "attr", "", "simple mesh", "geometryattributes" );
```

Creating shading networks uses the same `NSIConnect` calls as for scene description.

```
/** We can inline OSL source code directly */
const char *sourcecode = "shader uv() { Ci = emission() * color(u, v, 0); } ";
nsi.Create( "uv", "shader" );
nsi.SetAttribute( "uvshader", NSI::CStringPArg("shadersource", sourcecode) );

/** We can now connect our new shader node into our simple emitter */
nsi.Connect( "uvshader", "Ci", "simplesshader", "Cs" );
```

Assignment Priorities

Now that we have one mesh instanced twice with the same shader, how do we override the shader of one of the two instances? It is usual in scene graph APIs to have a hierarchical assignment strategy where the leaf nodes inherit parent attributes. In such scenarios, overriding of attributes is possible only on the child nodes. In our case, the child node has a shader assigned to it. Thankfully, NSI allows you to perform *top-down* overrides by specifying priories on connections. We can connect an attribute node, along with its shader, on the instance transform node (FIXME: graph needed here).

To be continued ...

```
/**
        Create another simple shader node using the standard OSL "emitter" shader.
        User default shader prameters.
*/
nsi.Create( "simpleshader2", "shader" );
nsi.SetAttribute( "simpleshader",
        NSI::CStringPArg("shaderfilename", "emitter") );

const char *k_attributes_override = "attribute override";
nsi.Create( k_attributes_override, "attribute" );

/* Default priiority is 0, so 1 will override */
nsi.Connect( k_simpler_shader2, "", k_attributes_override, "surfaceshaders",
    NSI::IntegerArg( "priority", 1) );

/*
  Connecting the attributes to the second instance transform will
  override the shader below because of higher priority.
*/
nsi.Connect(
        k_attributes_override, "", k_instance2, "geometryattributes" );
```

Multi-Camera

NSI has a powerful camera description features that allow rendering of multiple points of view at once (as in stereo renders). Moreover, a single camera can be used to render multiple different *screens.* This is achieved by separating the camera's *view description* from *screen's characteristics* su ch as resolution, crop and shading samples. It means it is possible to render two different images using the same camera with each image having its own resolution and quality settings.

```
/**
        Create a perspective camera and make it point down the Z axis.
        In NSI, Z- goes straight away from the viewer (same reference
        system as in most 3D applications, e.g. Maya).
*/
const char *k_camera_handle = "camera";
nsi.Create( k_camera_handle, "camera" );
```