The OSL Map and You

A Developers Guide

Introduction

3ds Max 2019 introduces the Open Shading Language (OSL). It is visible to the user in the form of the **OSL Map**. It is an execution environment for OSL shaders inside of 3ds max, and is exposed to the API as if it was a regular C++ 3ds max shader (i.e. it is of class **Texmap** and responds to calls to **EvalColor**, **EvalNormal** and **EvalNormalPerturb** methods.)

This means it out of the box works in **any** renderer supporting the classic 3ds max shader API (Scanline, vRay, Corona etc.). It also means it works **outside** of renderers, **anywhere** in 3ds max where a regular map is requested, such as in the Displacement modifier, etc.

It can also work with **renderers that support OSL natively**, such as Arnold. In that case, the execution environment inside the OSL Map is not used, instead, the OSL source code, the parameter values and shader bindings are sent to the renderer, which executes the OSL code itself.

More and more renderers supporting OSL natively are appearing every day.

OSL is a simple to understand shading language. Writing a shader in OSL is orders of magnitude less effort than developing the equivalent functionality as a 3ds max C++ shader. Instead of dealing with complex DLL plugins, building UI manually, and wrapping your head around “validity intervals” and other details of the 3ds Max API’s, you simply type some OSL code in a text file – and you are done!

OSL also benefits from JIT compilation and optimization of *entire shade trees at once,* as long as *all the shaders in the shade tree are OSL shaders*[1].

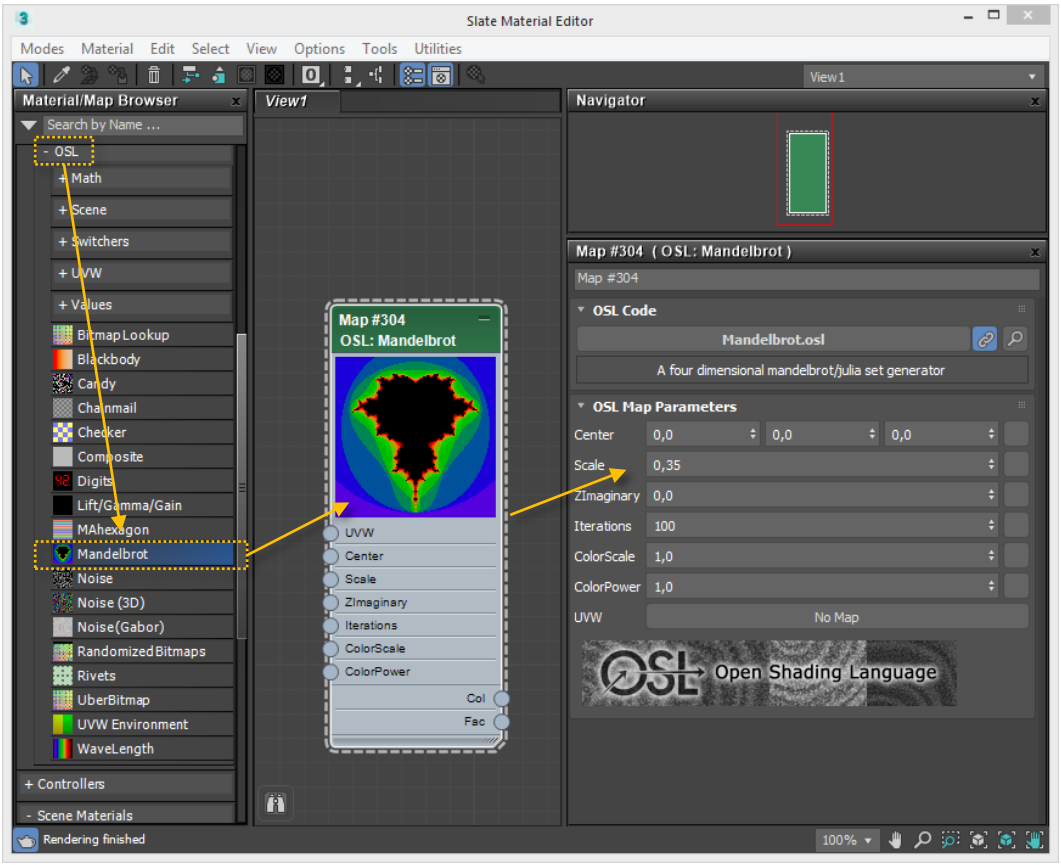
[1] You can mix OSL shaders and regular shaders, but the optimizations will suffer.

Use cases for the OSL Map

There are two fundamental “use cases” for the OSL Map. In a nutshell they look like this:

Use case #1: Normal “User” workflow

The user picks the **OSL** category in the Material/Map browser, picks some shader, drops it in the Slate material editor, and edits its parameters. Nothing is *effectively* any different from any other 3ds max map plugin; it shows up in the browser, you use it, you hit render or use ActiveShade – done.



To follow the metaphor of plugins, OSL shaders that populate the material browser come from subfolders named OSL inside the plugin directories. There is also a “system” OSL folder where basic shaders shipped directly by Autodesk, plus things like the OSL basic headers reside.

Shaders are loaded come from:

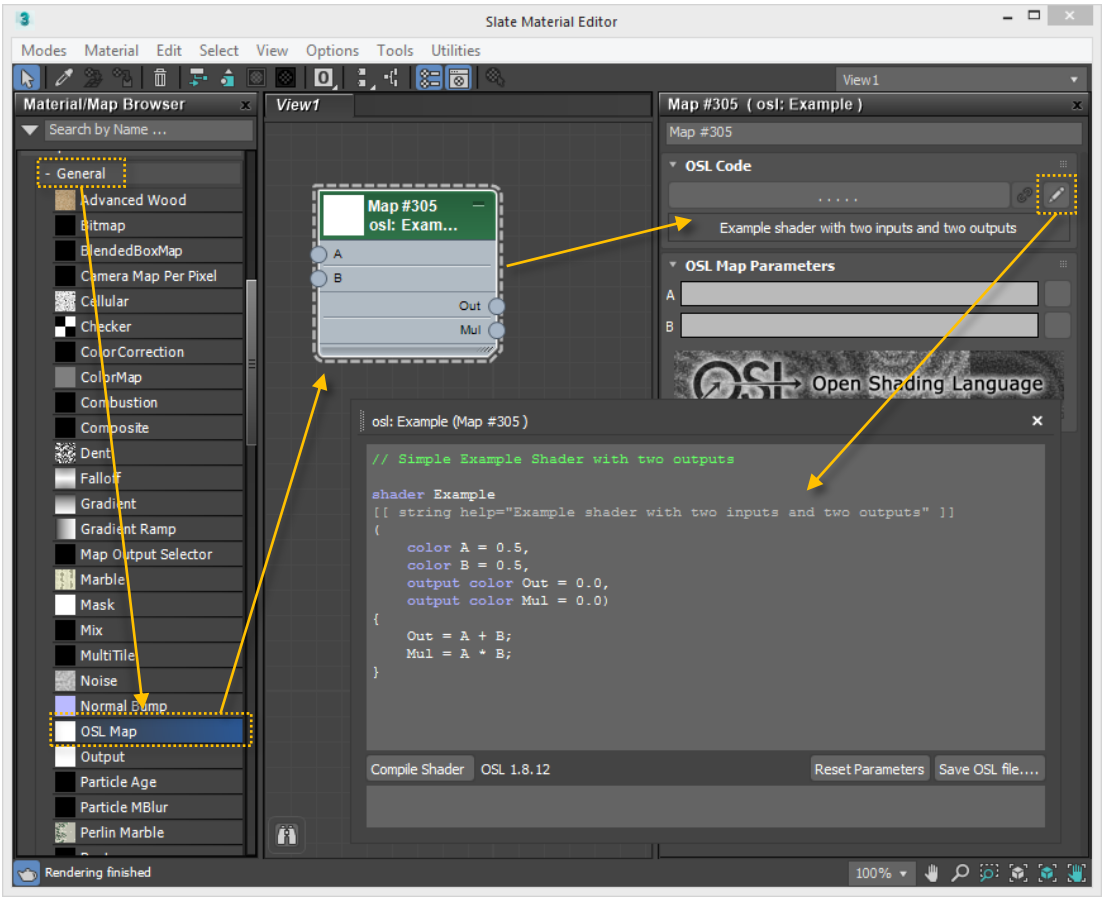
- <3dsmax>\OSL *(never touch files in this folder)*
- <3dsmax>\StdPlugs\OSL
- <3dsmax>\Plugins\OSL
- Etc.

TIP: If the OSL source is put in a *subfolder* to the OSL folder, it will show up in the Map browser in a *subcategory* named after the folder.

IMPORTANT: In this workflow, all shaders are hosted in *files in the plugin folders*, and the code in the file is what is being used at render time, again following the metaphor of plugins completely. Add a parameter to the code in the file, the shader will spawn a new parameter, and gain features. Change the algorithm, and the rendering result will change. Just like updating any other plugin.

Use case #2: The tinkerer and shader developer workflow

The **OSL Map** comes in another flavor, a bare, empty "unpopulated" map, with editing features. You find this among the General maps as the **OSL Map**. You drop it in slate, use it, but this is where it becomes exciting:



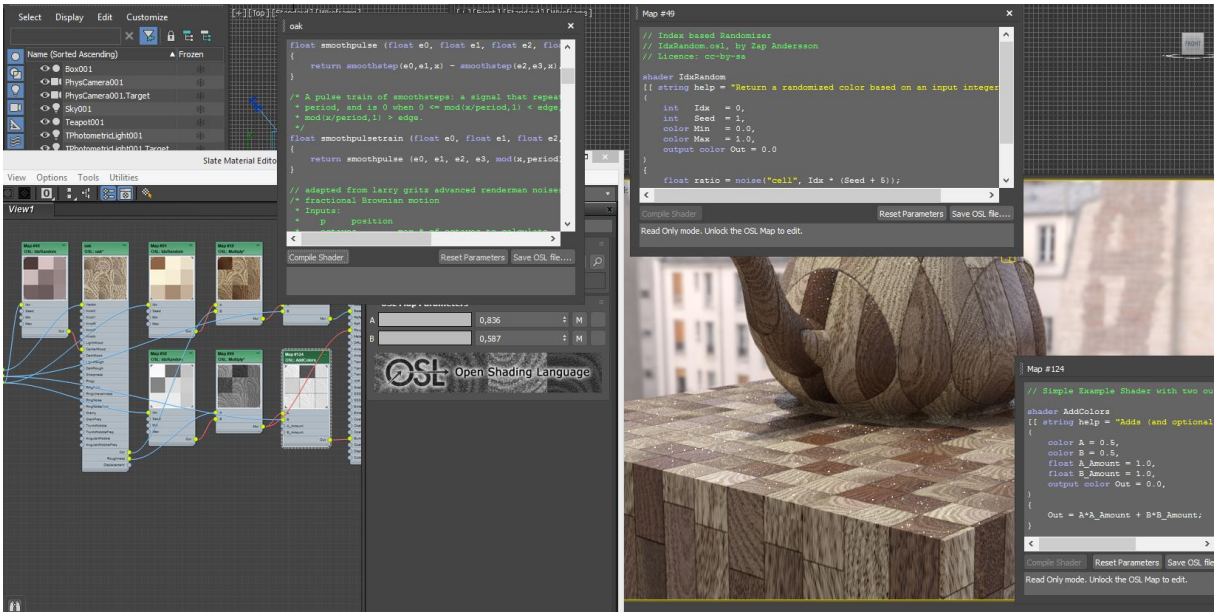
As before, user drags the **OSL Map** into slate, but it isn't really doing much (except a piece of demo code). The user can pick the ". . ." button up top to load in OSL shaders from disk that *doesn't* necessarily live in the **Plugins** folder hierarchy, but anywhere. The **OSL Map** will dynamically morph to the new parameters, spawn the additional inputs and outputs needed, and start rendering based on the loaded OSL file.

In this workflow, the OSL code is actually loaded *into* the **OSL Map**. It only uses the file when loading, from that point on, the code lives as a string parameter inside the **OSL Map**.

And now comes the fun part: The user can then click the **Edit** icon, and pop up the **OSL source editor**. It's a nice, simple, dockable syntax-coloring text editor, in which OSL code can be edited live. By hitting the **Compile Shader** button the shader will update to whatever the latest code says - even (with a renderer that supports it) *while rendering* in ActiveShade!

Edited files can be saved with the **Save OSL File** button, but since the code *lives inside the **OSL Map***, they can just as well be stored in max scenes, or even dropped in material libraries. No external dependencies on any files exist anymore, it is completely self-contained. A scene sent to a render farm across the world will never be missing a shader – they are *in the scene*.

This now turns the **OSL Map** into a complete shader development environment. You can work interactively with the code (multiple editors can be popped up at once).



When finished with your nice new shader, you can, if you want, save the OSL code to disk.

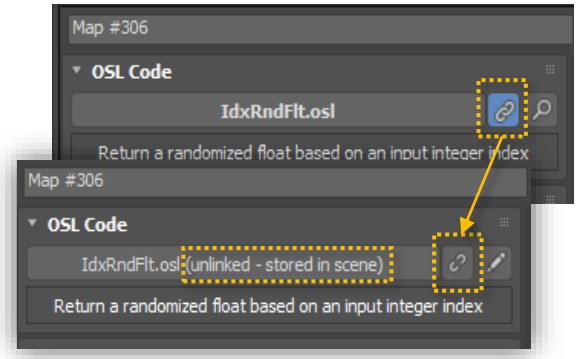
Maybe, when you are "done", even save it in … *one of the Plugins/OSL folders…* which means, next time 3ds max is launched, the shader will show up automatically in the Material/Map browser with all the others!

Yes, you can use the "bare" OSL Map as an editing and development environment for shaders you later deploy as "plugins", by putting them under the Plugins folder hierarchy.

Workflow #3 – there are no workflows

Sounds very Zen, but we lied. There's not really two workflows. There are only two mental states.

The **OSL Map** that runs the shaders preloaded from the map browser, and the **OSL Map** found in the General section are *one and the same*. As a matter of fact, dragging in something from the **OSL** section differs only in one aspect: the **OSL Map** comes in in the "linked" mode *by default*, and its file is preloaded to the appropriate OSL file. *That's it!*



The chain icons state indicates the "linked" mode.

When this mode is **on**, shaders are read from disk, from the file named on the file button.

When it is **off**, shader code is stored embedded *inside the **OSL Map** itself*.

And the best bit is – you can switch.

If you want to change an aspect of one of the preloaded shaders – you can! Just "unlink" it, and edit it to your hearts content. You will only be editing a *copy that is living in your scene*, never the original file on disk. All other shaders that are still in the "linked" state, will keep the plugin-like behavior. Those that are "unlinked" will allow local editing within that **OSL Map**.

If you attempt to “re-link” a modified map, it will revert back to the version in the file, and your edits will be discarded[2].

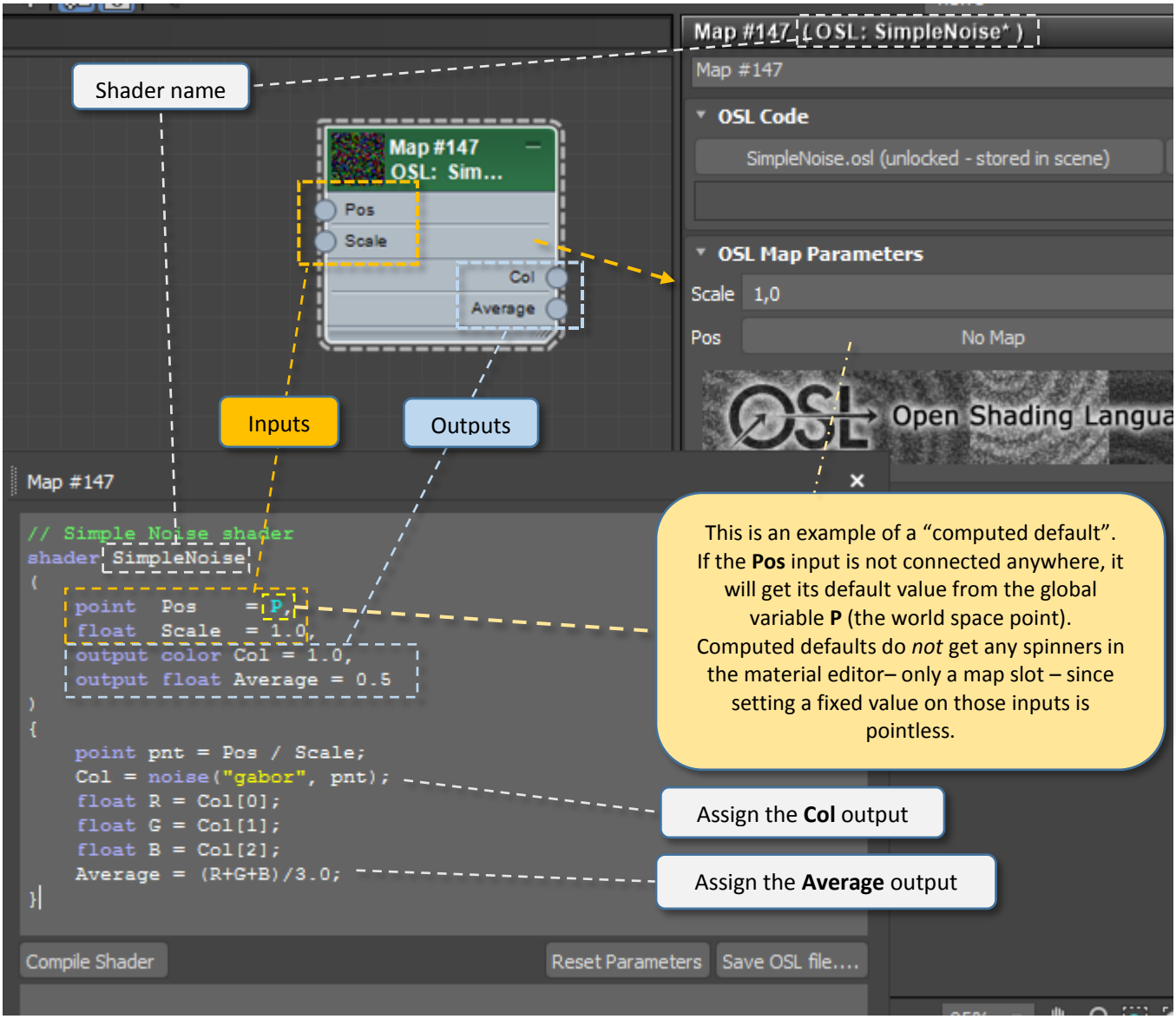
This makes simple one-off modifications, or even simple experimentation very easy. Tinker with something, if you break it, “link” it again, and it goes back to the way it was.

Writing an OSL shader for 3ds max

Describing the OSL shading language is way beyond the scope of this document, go to this link for more information on that topic:

https://github.com/imageworks/OpenShadingLanguage/blob/master/src/doc/osl-languagespec.pdf

We will however give a small example of what an OSL shader can look like, and some of the quirks of the language:



[2] Well, it is of course in principle possible, though strictly **not recommended**, to manually use the Save button and save the edit *over* the original file in the Plugins subfolder, but that is rather dangerous thing to try, and we discourage it strongly unless you truly know exactly what you are doing… and there are plenty of warnings discouraging you from doing such a thing.

The example above gives some of the basics of what writing an OSL shader is about. Here are some of the main points:

- A shader looks very similar to a C/C++ function definition
- Shaders have inputs and outputs. Both inputs and outputs *always* have to have a default value assigned (or an error is emitted when compiling).
- Inputs can sometimes have *computed defaults*. This is the value the input gets if it is *not connected to anything*. Most procedural texture shaders have an input for the coordinate for which it is texturing, but also uses a computed default so that the shader does *something reasonable* without being forced to connect e.g. a UV generating shader to its input.
- Since computed defaults are used for parameters that are inherently varying across the texture, having "fixed value" inputs for them is meaningless. Hence**,** *no value spinners are shown for these*, but *only* the slot for connecting them to some other shader.

Working with OSL code specifically for 3ds max

A few limitations

The OSL implementation in 3ds max 2019 has some specific limitations in this first version. Most importantly, it only shows up as a **Map**, for making procedural texturing. OSL "Closures" are *not supported* in this first version. In practice, this means – it is useful for texturing, not for making "materials".

There are several reasons for this, mostly technical, but one of them is that while the OSL language is well specified in its feature-set, the set of *closures* supported is renderer dependent, and can shift from renderer to renderer. We at Autodesk have a vision of making textures *and* materials interchangeable, and before a standard for closures is established, that is hard to accomplish.

Therefore, the recommended workflow at this time is to build your procedural texture maps in OSL, but connect the outputs to a standardized, well defined, renderer-independent material, such as the 3ds max **Physical Material**.

A second limitation is that only *base types* are supported for parameters. This effectively means integers, floats, strings, and point/vector/normal data types. Things like structs, or arrays are *not supported* in this version[3].

Note that this only applies to parameters; things like structs and arrays can of course be used inside the code itself. In practice, a good 99% of OSL shaders only use these types for input anyway.

[3] This will change in the future

Finally, due to the mode of embedding the OSL code into the **OSL Map** itself, *include files* are not supported. First, the standard OSL include file "stdosl.h" is implicitly included automatically by the compiler – you do *not* need to have a line…

```
#include <stdosl.h>
```

…in your shader code, and if you see one in a shader you downloaded online, you can delete it. Including other files will not work. For a shader that uses include files, you simply need to copy the relevant content from that file into the OSL source itself in this version[4].

How max fills in global variables and attributes

OSL by default works in a "common" coordinate space that can (in theory) be different from renderer to renderer. But in practice, *most* renderers use world space for this coordinate space, and this is true also for the **OSL Map**.

This means that for example the OSL global parameter **P** is in world space. Many shaders use the above mentioned feature of a *computed default* so that the shader works immediately, by having a line such as…

```
vector Input = P,
```

…in the shader code. This will make the **Input** parameter have the world space point as a default. Since texturing things in *world space* may not be what a user expects (since moving the object would drag it "through" the texture space). It is probably more useful to most users to instead use object space, which would look like this:

```
vector Input = transform("object", P),
```

Or, if it is a 2D texture, use the default UV space:

```
vector Input = vector(u, v, 0),
```

…since 3ds max populates the global OSL variables **u** and **v** with the default 3ds max UVW map channel 1.

Also, since in theory (according to the OSL spec) **P** is in "common" space, and you actually *want* it in world space, the *correct* way of doing that is:

```
vector Input = transform("world", P),
```

…although in the **OSL Map**'s execution environment, this is does nothing, since P is already in world space.

[4] This may change in the future

Scene Attributes

The OSL rendering state is filled in before 3ds max executes the shader. The standard variables such as **P**, **N**, **I** etc. are filled in, where the **u** and **v** variables gets pre-populated with map channel 1's UV coordinate.

OSL also allows the generic **getattribute** function to get a named attribute from the scene. It is up to the renderer executing the OSL code to populate these named attributes, which means, it is outside the control of the OSL Map itself to guarantee that these attributes are there or not. We are trying as much as possible to align between renderers, and this is the set of attributes the **OSL Map** itself uses (i.e. what can be guaranteed to exist when rendering inside of 3ds max and using the **OSL Map** as the execution environment)

3ds max 2019 OSL attributes when using the built-in OSL execution code

Attribute Name	**Data type**	**Meaning**
UVxx	point	The 3D UVW coordinate for map channel xx (**UV0** to **UV99**)
mtlID	int	The material ID of the face (or particle ID of a particle)
gBufID	int	The gBufID, which is called "Object ID" in the 3ds max object properties dialog
nodeName	string	The name of the instance in 3ds max. OSL has a built-in attribute **geom:name**, but that will be what the *renderer* calls the object, which isn't necessarily the same thing.
nodeID	int	What NodeID in the ShadeContext returns
nodeRenderID	int	What INode::GetRenderID() returns
nodeHandle	int	What INode::GetHandle() returns
wireColor	color	The instances wireframe color
paLife	float	Normalized particle age (0.0-1.0 over the age of the particle)
usr_*xxxxx*	int float string float,float,float	Any value from the 3ds max Object Properties "User Defined" page, as an integer, float, string, or three comma-separated floats. For example, typing "Hello=3.0" there, will allow you to get the float attribute **usr_Hello** from the OSL code to retrieve the value 3.0

But please remember this list is defined by the *renderer*, and is only guaranteed when run inside the **OSL Map**. For example, Arnold uses a slightly different naming to store UV coordinates.

Therefore it is ***discouraged*** to deal with UV channels directly in your OSL code: Add an input for the coordinates, defaulting it to `vector(u, v, 0)` which is guaranteed to yield a meaningful default value for 2D textures. Then let the user connect the Autodesk-supplied **GetUVW** shader, which has code in it to understand the different UV attribute formats for the supported renderers.

Metadata – making your shader Pretty

In OSL metadata can be assigned to whole shaders, or individual parameters. 3ds max supports a subset of those metadata settings, as well as some of its own.

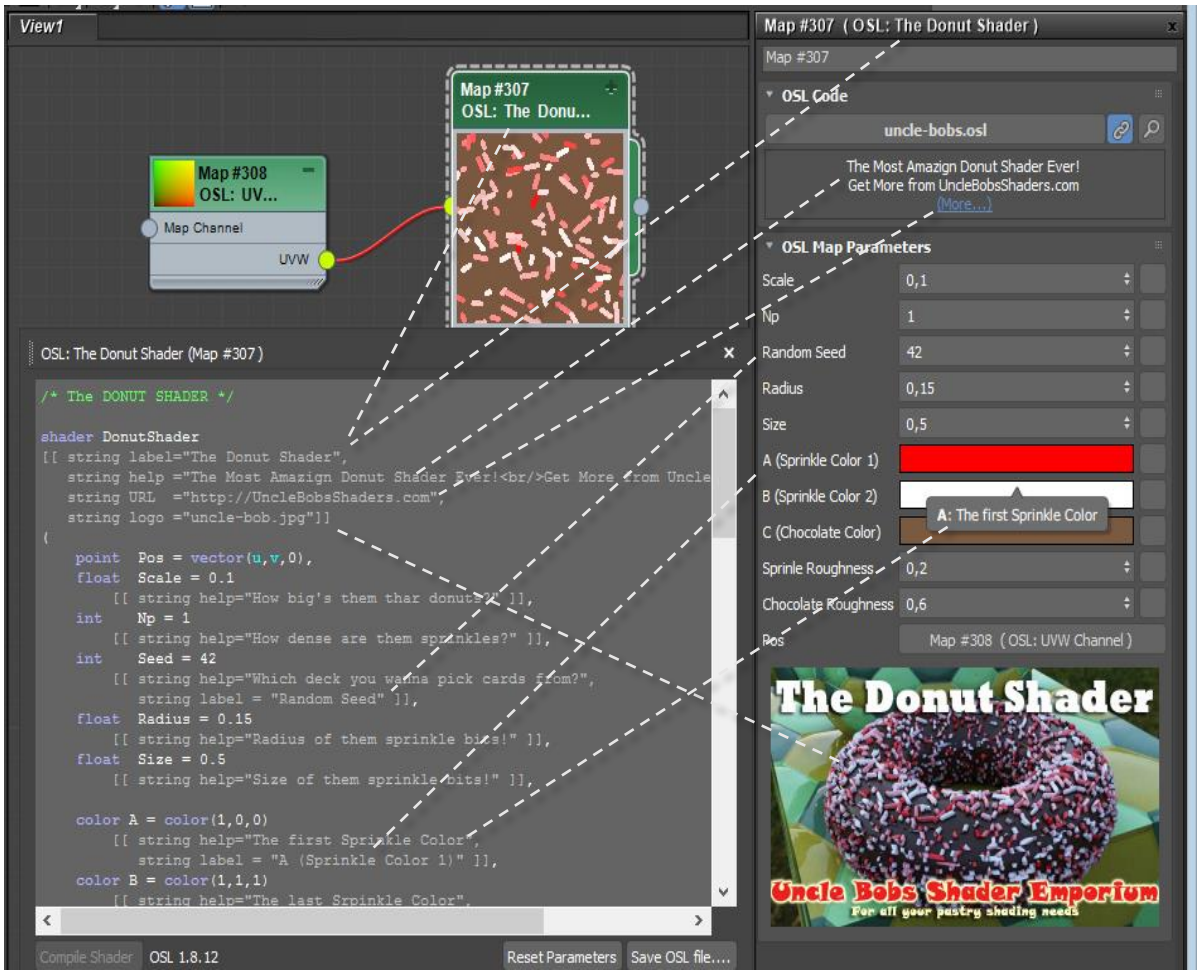
For the entire shader, the following metadata are supported

Name	Meaning
label	The "display name" of the shader. What it shows up as in the material browser, as well as its title in the material editor and on the node in SME.
help	A short help text showing up at the top of the material editor. This can contain a subset of HTML formatting.
URL	A link to a more extensive help page about the shader
logo	An image, loaded from the same folder as the OSL shader, displayed at the bottom of the material editor in place of the default logo. Allows you to personalize the look of your shaders a little, or even include a sample image of what it does directly in the UI. If the file cannot be found, the default logo is shown.
category	Sorts this shader in a particular category in the 3ds max map browser. By default, categories come from which subdirectory the shader is in, but this can be overridden by a specific category metadata value. It allows nested categories separated by a backslash, e.g. "Math\Color" puts the in the "Math" category, sub-category "Color"

For individual parameters, the following metadata are supported

Name	Meaning
label	The "display name" of the parameter. What it shows up as in the material editor
help	The tooltip of the control. Can contain a subset of HTML formatting.
min, max	The minimum and maximum value of the parameter
widget	The kind of UI control to display for the parameter. Supports the following OSL widget varieties: • "filename" (shows a file selector and makes the pointed-to file a 3ds max asset) • "checkBox" (shows an integer value as a checkbox) • "popup" (provides a popup list of choices) • "mapper" (provides a popup list of choices that maps to other values such as integers) See the OSL Specification for details
timeValue	If enabled (set to 1), any integer parameter this metadata is assigned to automatically gets populated with the current frame number, and any float parameter gets automatically populated with the current scene time in seconds.

Metadata example:



Concluding words

Developing shaders in OSL is a lot easier than writing 3ds Max C++ shaders. These shaders can much more easily be exchanged between users, and with the mode of embedding them in the scene file, exchange is actually automatic and transparent.

The added benefit is native support in more and more renders, such as Arnold and similar, and a potential for wider shader portability between render engines.

The built-in editing mode also makes simple one-off experiments much easier, and gives a never-before seen flexibility in 3ds max shading to developer and code-inclined end-users alike.

The suggestion is to think twice before developing a shader for max in anything other than OSL from here on out. You would have to have a very good and specific reason to do it in any other language….

There is a public GitHub for exchange of 3ds max shaders here:

https://github.com/ADN-DevTech/3dsMax-OSL-Shaders