OSL in 3ds Max 2019

Zap Andersson 2018-03-16

Introduction

3ds Max 2019 introduces the Open Shading Language (OSL).

OSL is an open source shading language, and is fairly simple to understand. Writing a shader in OSL is orders of magnitude less effort than developing the equivalent functionality as a 3ds max C++ map. Simply put, type some OSL code in an editor, and you are done. More details about this in the developer documentation.

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 implemented so it can work like any regular built-in 3ds max map.

This means it out of the box works in **any** renderer supporting the regular 3ds max shading 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.

But 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 uses “Just-In-Time” compilation and optimization of *entire shade trees at once* - as long as *all the shaders in the shade tree are OSL shaders.* You *can* mix OSL shaders and regular shaders, but the optimizations will suffer

A few Limitations of OSL in 3ds max

The OSL Map has a few limitations in this first version.

Only supports Maps, not Materials

A shader written in OSL can have multiple outputs returning different data types like colors, floating point values, integers, strings, and so on. This is all supported.

However, OSL also supports a special type known as a “closure”. One can think of closures as being “materials”. This is not supported by the current version. In practice, this means, the OSL Map can only be used to build procedural textures (Maps), and *not* to build Materials.

Therefore, the recommended workflow 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**.

If an OSL shader you find online doesn’t appear to do anything, or doesn’t appear to have any outputs, it might be an OSL “material”.

Only supports plain data types

The OSL Map in max only support the plain data types: Colors, vectors, floats, integers and strings. It does *not* support arrays or structs as inputs.

This is rarely a problem in practice, because very few OSL shaders actually use arrays or structs as inputs. Note that the limitation only applies to parameters –arrays and structs still work fine inside the OSL code. More information in the developer documentation.

No #include statements

OSL code can in 3ds Max not contain the **#include** statement. If your OSL code contains…

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

…is not needed and can be removed, the file **stdosl.h** is implicitly included by the compiler.

A note on terminology

Sadly, computer graphics terminology is not always precise, and words like “shader”, “map”, “texture”, “procedural” and “material” mean different things when used in different context, in different applications, or even for different renderers.

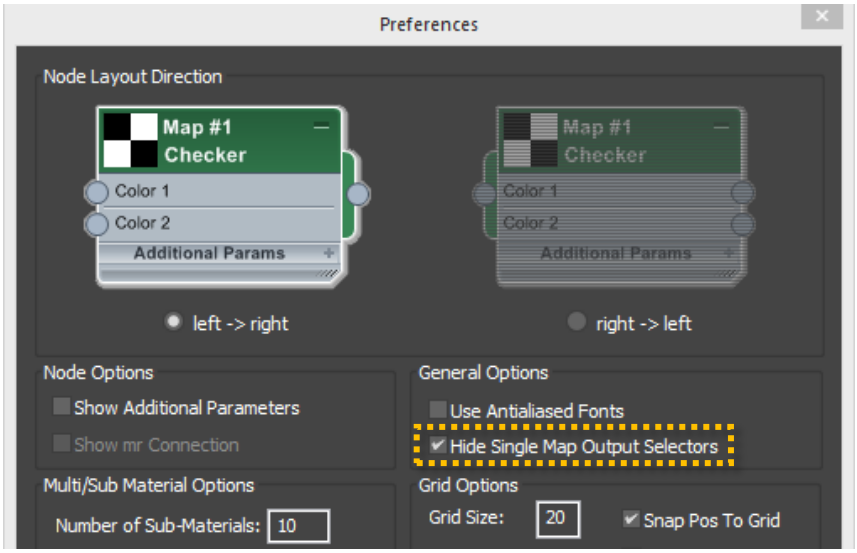
In OSL, everything is called a “shader”. It is, after all, a “shading” language. What the shader does, depends on its return type; things that returns a *closure* would act as what 3ds max calls a “material”, and things that do not, would act as what 3ds max calls a “map”.

Since we do not support closures yet (see *Only supports Maps, not Materials*), it will, in max, always show up as a “map”, and hence only exists under the “Maps” category. But since this help text largely speaks about OSL, the term “shader” will be used when referring to things in the OSL context, even though it does behave like a “map” to max.

A note on maps with multiple outputs

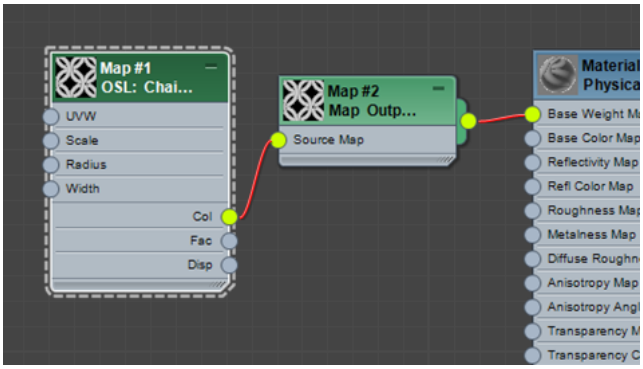
The original 3ds max rendering API did not support multiple outputs. This is an extension that was introduced later. For this reason, most 3ds max materials and maps do not understand connecting to a map with multiple outputs. To solve this, the **Output Selector** map was introduced, which is a go-between that is automatically inserted between something that has multiple outputs, and something that does not understand multiple outputs.

In 3ds max 2019 this “in between map” is automatically hidden, so in Slate Material Editor it looks like a normal connection (except the wire is blue instead of red). The option can be turned on or off by choosing **Preferences** from the **Options** menu in Slate, and toggling the **Hide Single Map Output Selectors** option:

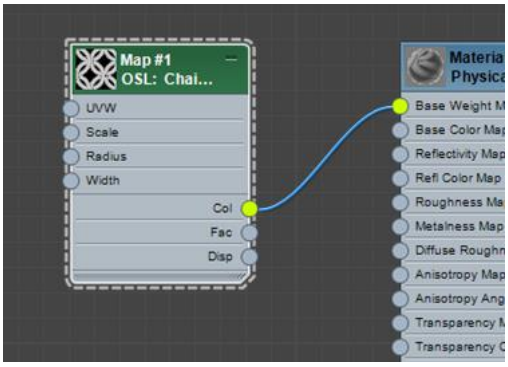


It is important to note that this hiding is purely cosmetic and only applies to Slate. The Compact material will still display the **Output Selector**, it will be seen by MaxScript, and so on.

Output Selector Visible



Output Selector Hidden

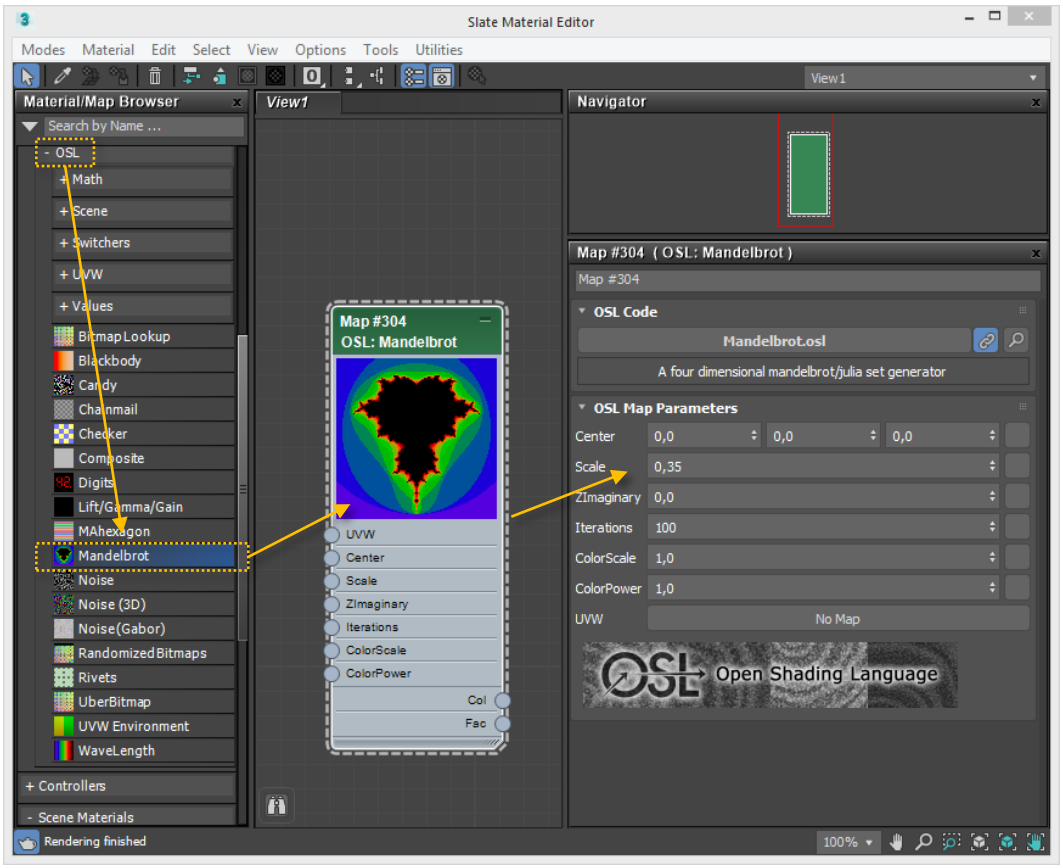


Use cases for the OSL Map

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

Use case #1: Normal “User” workflow

You pick the **OSL** category in the Material/Map browser, pick some map, drop it in the Slate material editor, and edit 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 in a supported renderer – done.



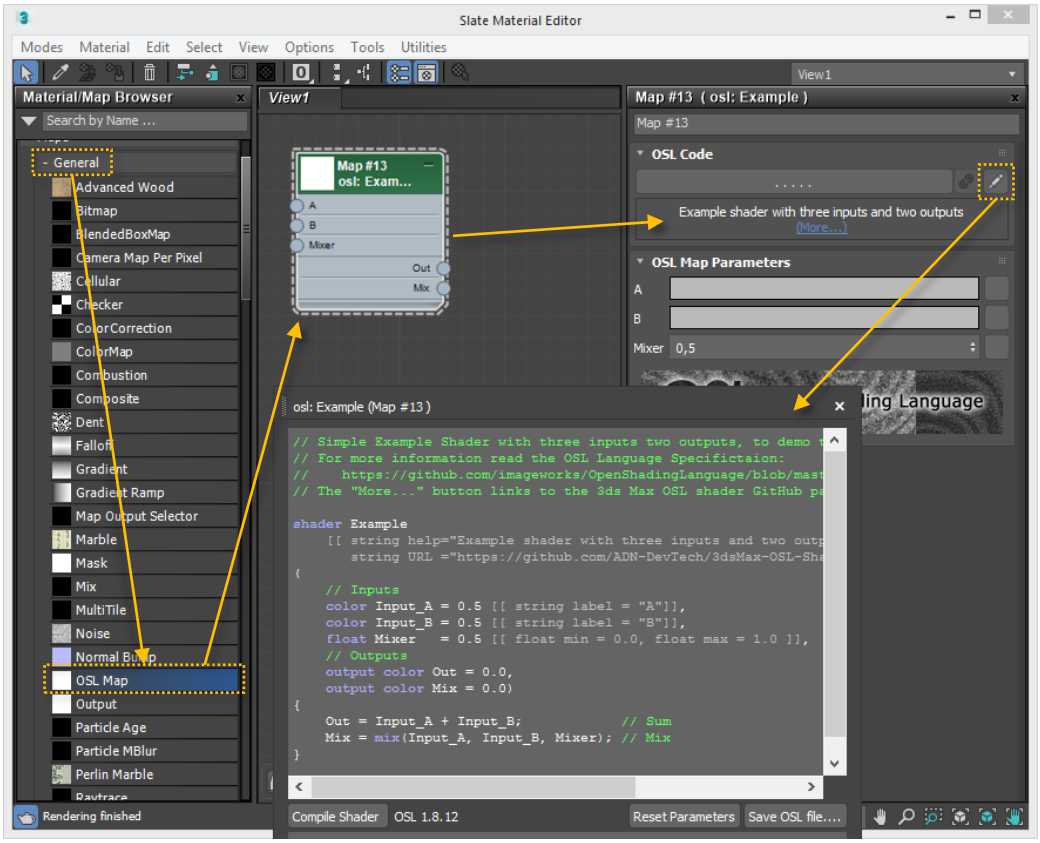
All the OSL maps that populate the material browser in this way come from 3ds max plugin folders. More details on exactly how OSL files are automatically loaded, displayed or categorized can be found in ***Adding more OSL Shaders*** and in the developer documentation.

It is important to note, that in this workflow, all shader code is hosted in *files under the plugin folders*, and in general behave like any other plugin. If an updated version is dropped in the appropriate folder, the new version will be used when rendering, just like updating any other plugin.

Max ships with roughly 100 OSL shaders, which are documented in the section ***Quick tour of the preinstalled OSL shaders***.

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:



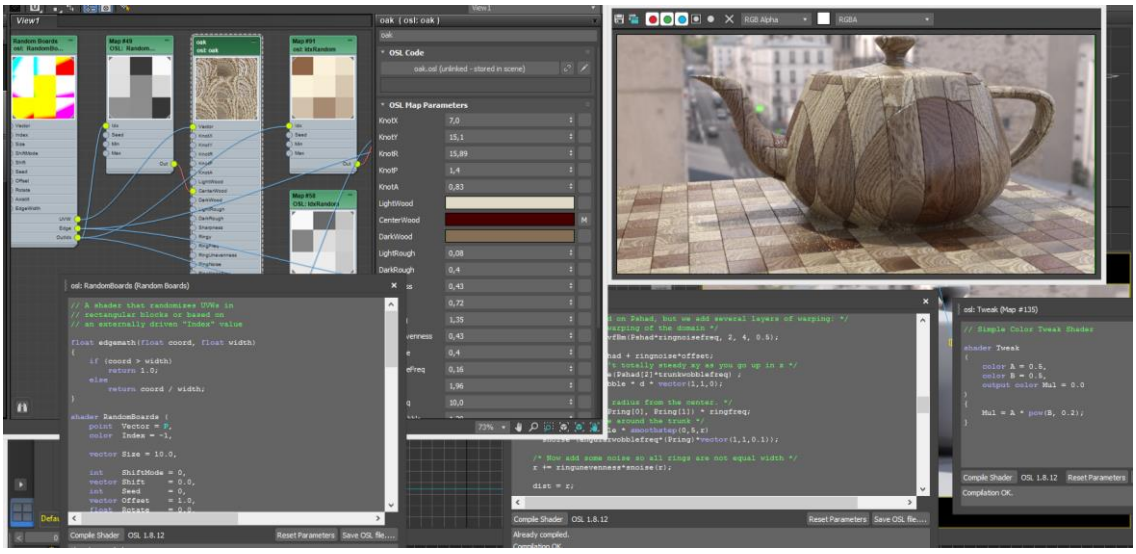
As before, you drag the **OSL Map** into slate, but it isn’t really doing much (except a piece of demo code). You can pick the “. . .” button up top to load in OSL shaders from disk that *doesn’t* necessarily live in the **Plugins** folder hierarchy, but anywhere. The **OSL Map** will dynamically morph 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: Clicking the **Edit** icon pops 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 (with a Ctrl-S shortcut while the cursor is in the OSL editor text box) 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.

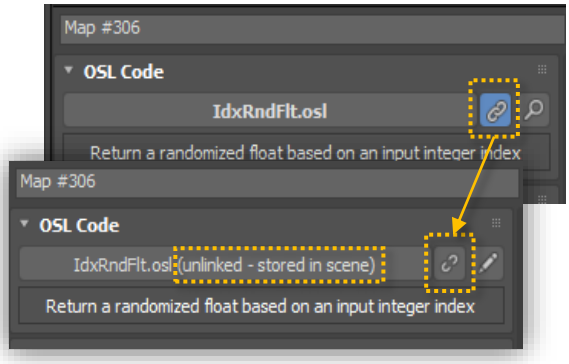
Maybe, when you are "done", even save it in … *one of the Plugins/OSL folders…* which means, next time 3ds max is launched, the shader will show up automatically in the Material/Map browser with all the others!

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.

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.

Adding more OSL Shaders

OSL shaders that automatically show up in the Map Browser are loaded from two kinds of places:

- The main “system” **OSL** folder located under the 3ds Max root folder. This contains all OSL files shipped with 3ds Max – **do not change these files or put any new files in this folder!**
- Subdirectories named **OSL** to each of the paths listed under **Customize** -> **Configure System Paths** -> **3rd Party Plug-Ins**

So to add your own OSL shaders, there are two options:

- Simply put them in the *<3dsmax>***\Plugins\OSL** folder
- Create a new folder on your computer, and add that folder as a **3rd Party Plug-In** path. Create a subdirectory under it named **OSL** and put the files in there.

In both cases, if you put files in further subdirectories under the **OSL** directory, this will show up as categories in the Map Browser.

ADN (Autodesk Developers Network) runs an OSL GitHub page at...

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

…where you can find more OSL shaders to download.

Quick tour of the preinstalled OSL shaders

About the shaders

The OSL shaders shipping with 3ds max 2019 come in a main category with several subcategories. The subcategories contain mostly smaller helper shaders who will be only briefly documented, whereas the main category contains the slightly more complex shaders that are documented in slightly more detail.

When exploring the OSL maps, remember that the top of the UI contains a small explanatory help text, and that each parameter can have a tooltip, and that this document only explains the maps shipping with 3ds max 2019, but you can easily use third party ones, or even write your own….

A special note on "computed defaults" and UVW inputs

OSL has a *very special feature* called a "computed default". That means that a map can have an input which *if it is not connected anywhere*, can contain some specially computed value as a useful default. You recognize a computed default by the fact that the input has a map input, but there are no value spinners in the UI for the value (because, if it is not connected, the computed value is used).

Many maps use this, for example, all of the included OSL textures come with an **UVW** input. In contrast to the classic max maps, this allows connecting the output of *any* computation[1] to drive the lookup position of the texture.

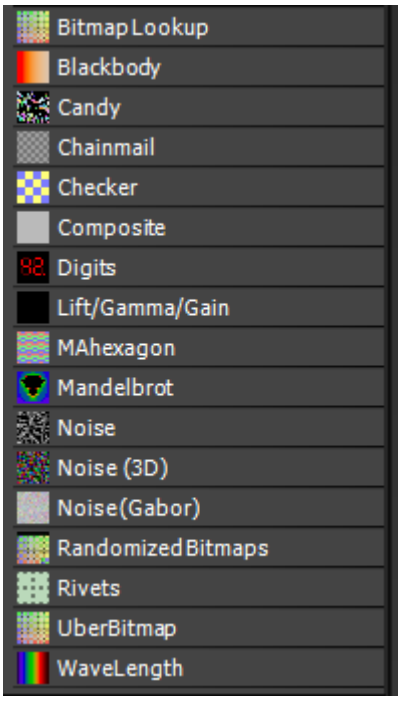
But for the map to do *something meaningful* when the UVW input is *not* connected, *computed defaults* are used. Most 2D textures use a computed default mapping to 3ds max **map channel 1**, and most 3D textures has a computed default of the shaded points **object space** coordinate. That way, maps give a reasonable default texturing when used directly. If one wants to use a different UVW map channel, or transform the UVW coordinates somehow, one can use maps from the **UVW** category (explained in more detail below) to accomplish that.

Note however that exactly *what* computed default is used is completely up to the OSL code itself. A few maps does "special" things purely for demonstration purposes, for example the **Blackbody** shader uses the U coordinate of **map channel 1** to make an example gradient across the surface, as does the **Wavelength** shader. This also means that OSL maps downloaded from the net may literally do "anything" for a computed default.

Just keep an eye out for things that only has a Map input and no spinners, and know that it's probably intended for you to supply a meaningful input there.

[1] Anything that returns a three-component value such as a vector, point, or even a color.

General Category

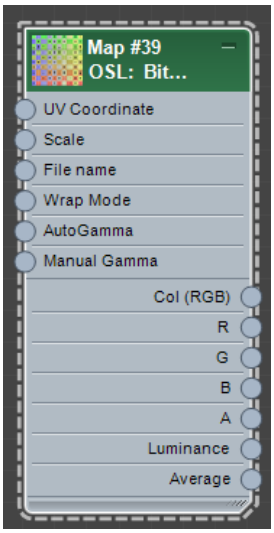


The maps in the general category serve a few different purposes. First, some of the larger, more complete and immediately useful ones, such as **Bitmap Lookup, Composite, Rivets** or **RandomizedBitmap** are here.

But it also contains a few maps that can be used as basis and examples for home grown OSL shader experiments (such as **Checker** or **MAHexagon**).

BitmapLookup

This shader is the generic way to look up a pixel in a bitmap in OSL.



Unlike the classic max **Bitmap**, here you can supply any computation to feed the UV coordinate for what pixel to look up, giving ultimate freedom.

Note: Texture lookups in OSL are done via OpenImageIO and hence supports only the image types OpenImageIO supports.

Note: Avoid using non OSL shaders in an OSL shade tree, it is much less efficient to use the regular max **Bitmap** shader than this one when working with OSL.

These are the parameters of the shader:

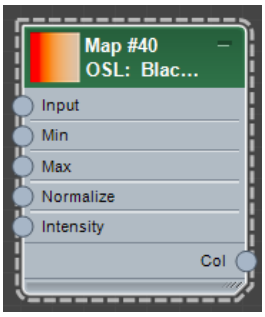
Parameter	Description
UV Coordinate	The input coordinate for which to look up a pixel. Defaults to UV values from 3ds max map channel 1. To use another map channel, connect the UVW Channel map. Or connect any map or combination of maps computing a vector.
Scale	A scale factor, for convenience. Larger values makes the texture larger.
File Name	The name of the file to display
Wrap Mode	How to treat lookups outside the 0-1 UV range. Available values are: • "default" – whatever is the renderers default • "black" – returns black for outside points • "clamp" – repeats the edge pixels for outside points • "periodic" – repeats the texture in a cyclic fashion • "mirror" – repeats the texture in a mirrored fashion
AutoGamma	Attempts to automatically compute the gamma setting for the texture
Manual Gamma	When **AutoGamma** is off, the manual gamma to use

The shader has following outputs:

Output	Description
Col (RGB)	The output as a color
R / G / B / A	The individual R, G or B and Alpha outputs of the color.
Luminance	The weighted luminance of the color. Useful as the "black and white" output that is visually similar to the original image.
Average	The average value of the RGB color. Also "black and white" but does not discriminate between colors.

BlackBody

Computes the color emitted by a radiative black body for a given range of Kelvin temperatures.



These are the parameters of the shader:

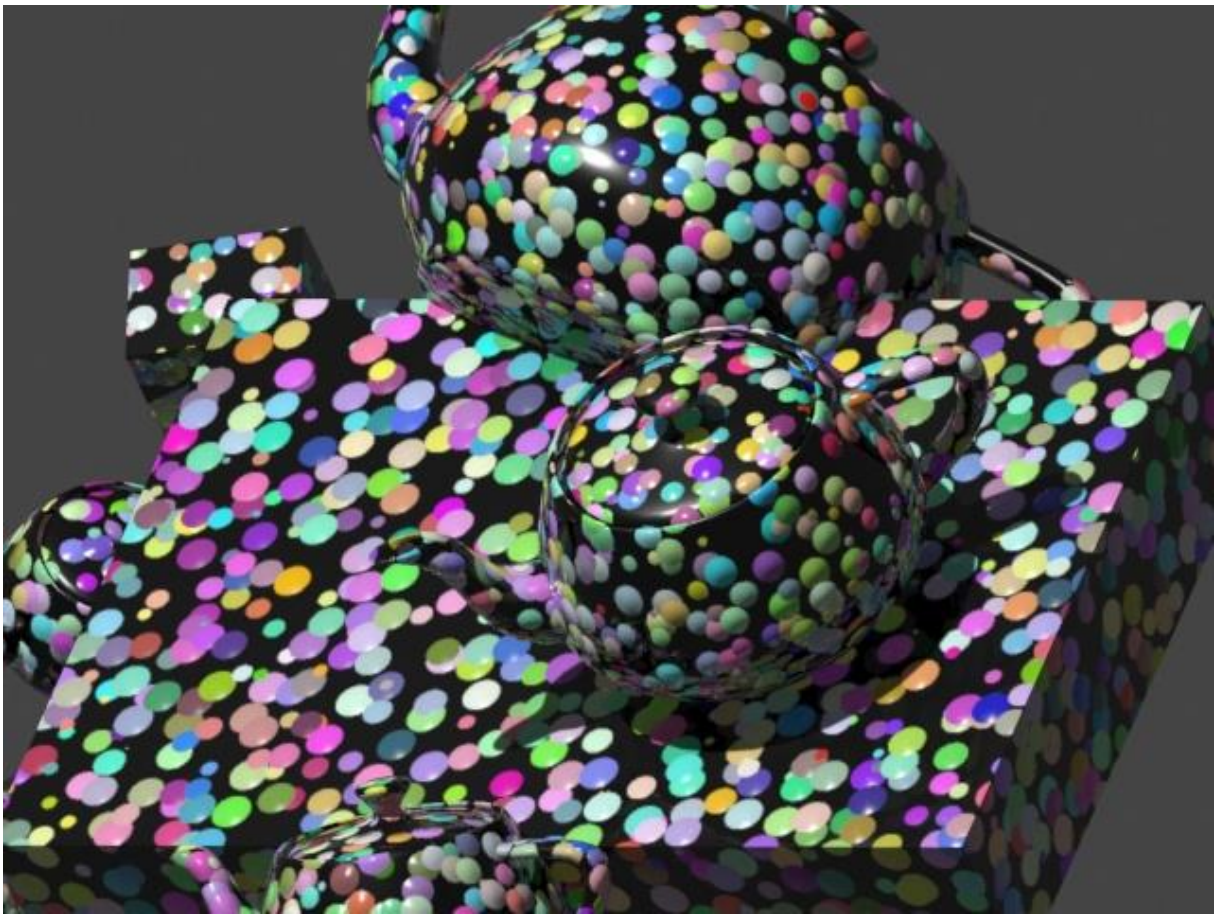
Parameter	Description
Input	The input value. Connect the parameter driving the temperature here. The default, when not connected, is a demo gradient along the U axis of map channel 1
Min	The Kelvin temperature for when Input = 0.0
Max	The Kelvin temperature for when Input = 1.0

Normalize	When off, the shader computes both a color and an intensity (which varies quite strongly with temperature). When on, it keeps the intensity constant and only computes the color.
Intensity	Intensity multiplier of the output

It has a single color output named **Col**.

Candy

Computes random spheres at random positions in space (which on a 2d surface looks like random circles). Technically, space is divided in a grid with a randomly sized and positioned sphere in each.



These are the parameters of the shader:

Parameter	**Description**
UVW	The UVW coordinate. The default, when not connected, is object space.
Scale	A simple scale factor of the effect. Sphere (circle) centers are (on average) this far apart. Technically, the size of the grid boxes.
Radius	The size of each sphere (circle) in relation to its grid box. Beware that values larger than 1.0 may reveal the grid box edge.
RandomOverlap	When off, a slightly faster algorithm is used, but that always causes the spheres to appear to have the same “order”, which can, if significant overlap is used, look like the spheres are sorted in a given order. When on, the sphere order is randomized.
BumpAmount	Scales the strength of the Bump output

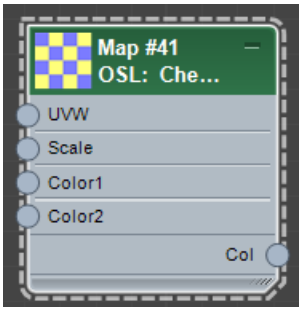
BumpShape	Tweaks the “curve shape” of the Bump output. When above 1.0 makes the bumps to appear more “pointy”, when below 1.0 they appear more “round”.

The shader has five outputs:

Output	Description
Col	A random color per sphere, or black when outside the spheres
Fac	Circle presence factor. Is 0.0 when not on a sphere, 1.0 when on a sphere.
Bump	A bump output, suitable to connect to the Bump or Displacement input of e.g. a Physical Material.
Dist	The normalized distance from the center of the sphere (0.0) to the edge (1.0)
Rnd	A random number per sphere (from 0.0 to 1.0)

Checker

A checkerboard pattern, as a programming example of the simplest possible useful shader, and a good start point for your own experiments in shader writing.



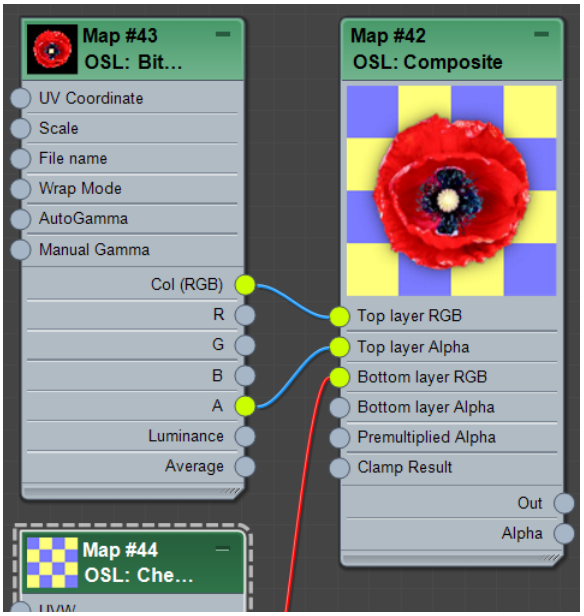
Note that this shader is technically 3D, and doesn’t actually do 2d squares but 3d *cubes*. But when fed only 2d coordinates (which is the default) it will appear as a regular checker shader.

These are the parameters of the shader:

Parameter	Description
UVW	The UVW coordinate. When not connected, defaults to map channel 1
Scale	A scale factor for the squares / cubes
Color 1	The first color
Color 2	The second color

Composite

A simple compositing shader that does an “over” compositing operation between two layers, i.e. map a top layer over a bottom layer based on an alpha (opacity) channel.



To composite multiple layers, these can simply be cascaded, one feeding into the layers of the other.

These are the parameters of the shader:

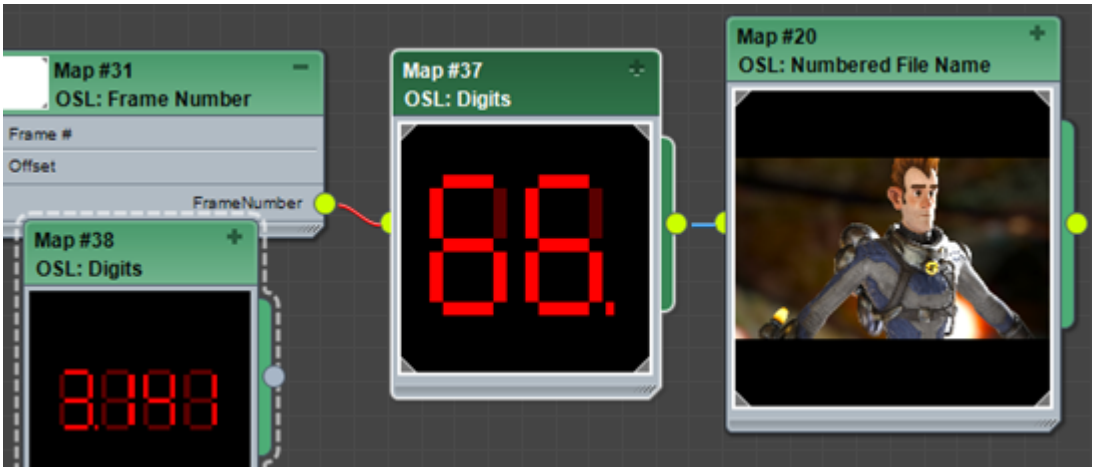
Parameter	Description
Top layer RGB	The color channel of the top layer
Top layer Alpha	The alpha channel of the top layer
Bottom layer RGB	The color channel of the bottom layer
Bottom layer Alpha	The alpha channel of the bottom layer
Premultiplied Alpha	If the top layer should be interpreted as having premultiplied (aka as “associated”) alpha channel or not.
Clamp Result	Clamps the result to the 0.0 to 1.0 range

The shader has two outputs:

Output	Description
Out	The output color
Alpha	The output alpha

Digits

A shader simulating a 7-segment display. Can be used for actual rendering, but may actually be more useful for debugging things in Slate.



These are the parameters of the shader:

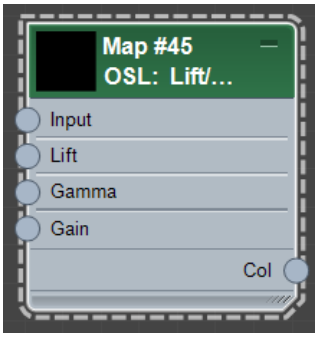
Parameter	Description
UVW	The UVW coordinate. When not connected, defaults to map channel 1. The shader keeps the digits within the unit square in UV space.
Number	An input for the number to be displayed
Digits	A number (from 2 to 8) of for the minimum number of digits to display. The shader will automatically add digits when necessary to display the number or support the requested number of decimals.
Decimals	The number of decimals
Scale	A simple scale factor of the digits.
BGColor	The background color
OnColor	The color for when a segment is on (lit)
OffColor	The color for when a segment is off (dark)
UseOffColor	If this is off, the OffColor will not be used and the BGColor is used instead

The shader has three outputs:

Output	Description
Col	The color output
Fac	A segment opacity output. Returns 0.0 for the background and 1.0 for a digit segment. If “UseOffColor” is off, only the lit segments return 1.0
Number	A pass-through output for the number. Useful for debugging or making the data flow clearer in a shading graph.

Lift/Gamma/Gain

A simple color shader to change brightness/contrast/gamma of a color. Note that a more complex color tweaking map can be found in the Colors category

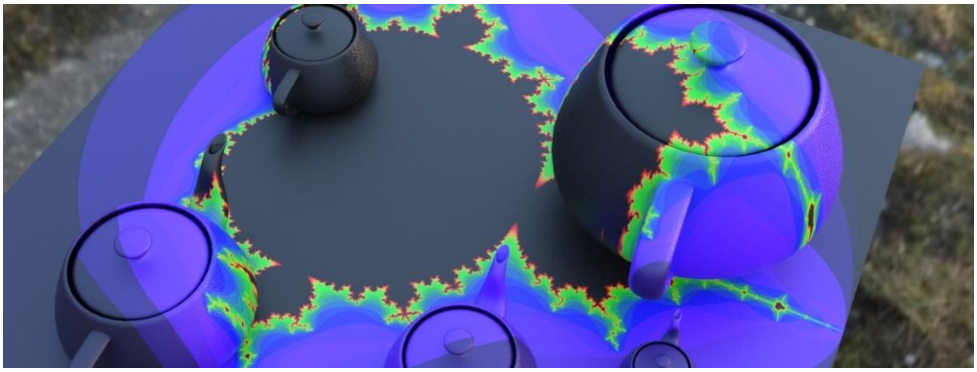


These are the parameters of the shader:

Parameter	Description
Input	The color to modify
Lift	Adds a “lift” by adding this value to the color
Gamma	Tweaks the color curve by applying a gamma (power) to the color
Gain	An intensity multiplier to change the brightness of the color

Mandelbrot

Computes a Mandelbrot/Julia fractal in up to 4 dimensions.



The fractal is computed by iterating the function…

$$Z = Z^2 + C$$

…until it reaches infinity.

Most people think of the Mandelbrot equation as having a two-dimensional input by feeding different initial C values in as a complex number for UV point, and initializing Z to zero.

Conversely, the Julia set is created by picking a fixed C value, and initializing the Z as a complex number for each UV point.

This shader combines the two, allowing you to both initialize C and Z at the same time, allowing you to make things like three- or four-dimensional Mandelbrot sets, or other fancy things.

These are the parameters of the shader:

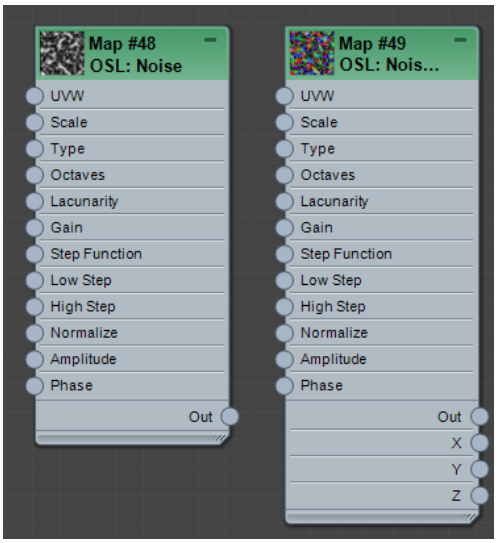
Parameter	**Description**
UVW	The U and V coordinate initializes the real and imaginary components of the C constant, and the W coordinate initializes the real component of the Z constant. This means, for a regular 2D input coordinate (which is the default if nothing is connected), a regular Mandelbrot set is rendered, but for a 3D coordinate, the Mandelbrot set is extended into three dimensions.
Center	The center point, to align the Mandelbrot set better with the unit UV square.
Scale	A simple scale factor
Zimaginary	Since the W coordinate of the UVW input only set the real component of Z, here you can feed in the imaginary component.
Iterations	How many iterations to try before “giving up” to find infinity
ColorScale	Tweaks the output color range
ColorPower	Tweaks the output color range curve shape

The shader has two outputs:

Output	**Description**
Col	The color output, using a built-in color gradient based on a rainbow
Fac	A float output proportional to the number of iterations run. Can be used to drive some other type of effect rather than the default “rainbow”

Noise

This shader works identically to Noise (3D) with the exception that it only has a single output, and is therefore slightly more efficient for just applying noise effects to a single value. See **Noise (3D)** for parameter documentation.



Noise (3D)

A noise shader that works identically to **Noise** except it can return three noise values at the same time, either independently or as a 3D vector (or color).

The classic max “Noise” can add complexity by adding up multiple noises over different “octaves”, where the effect of each “octave” is doubled in frequency and halved in intensity, and then applies a min/max cutoff to the result.

This noise shader works a bit differently. It still sums up multiple levels of noise over what is called “octaves”, but in actuality, the amount of frequency change per iteration is defined by the Lacunarity, and the intensity change by the Gain. The defaults (2.0 and 0.5 respectively) yields a behavior similar to classic noise, but you have full control over these settings and can set them to any other values, to sculpt the behavior with more detail.

Also, the min/max from the classic noise is mirrored here by the “Step Function”, but the big difference is that it is applied per iteration of noise, rather on the final result.

OSL noise has six different types

- **perlin** – classic Perlin noise, returning values in the -1 to 1 range.
- **uperlin** – same as Perlin but unsigned, returning values in the 0 to 1 range.
- **cell** – A noise that returns a fixed value for each unit cube of space
- **hash** – A noise that returns a fixed random output for a fixed input, but with no continuity, i.e. if the input changes even slightly, a *completely different* output value is returned.
- **simplex** – Simplex noise
- **gabor** – Gabor noise. Not that Gabor noise has some additional parameters and for that reason there is a separate Noise (Gabor) shader.

For more technical details of the various noise types, refer to the OSL language specification.

These are the parameters of the shader:

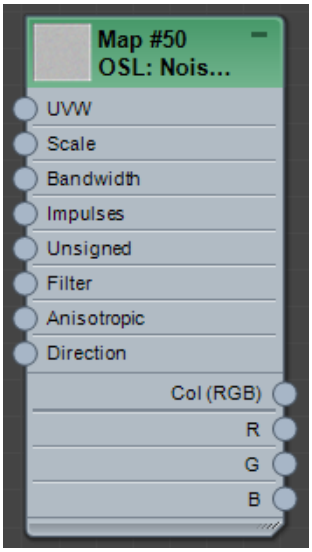
Parameter	Description
UVW	The UVW coordinate. The default, when not connected, is object space.
Scale	A simple scale factor of the noise.
Type	A choice of the type of noise.
Octaves	How many “octaves” of noise. Effectively how many times noise is iterated to add complexity.
Lacunarity	How the frequency changes for each iteration. The default 2.0 will create noise that changes spatially twice as fast for each iteration.
Gain	How the intensity changes for each iteration. The default 0.5 will halve the intensity for each iteration.
Step Function	When on, uses the step function. Note that it is applied per iteration, unlike the min/max of the classic max Noise shader.
Low Step	The low point of the step function.
High Step	The high point of the step function.
Normalize	If on, normalizes the shader output to the sum of all the iterations. Basically, keeps the output constant regardless of you adding/removing an iteration.
Amplitude	Just a scale factor for the output
Phase	OSL noise functions are actually four dimensional internally. This is simply the fourth coordinate (the other three coming from the UVW input), which allows to for example evolve the noise over time, or similar.

The shader has four outputs:

Output	Description
Out	The output as a 3D vector (which can just as well be used as a color)
X / Y / Z	The individual X, Y or Z outputs of the vector, i.e. the three noise values independently.

Noise (Gabor)

A specific map for Gabor noise, exposing some of its specific parameters.



Gabor noise is slower to compute than regular noise, but filters better, and have some additional parameters. If none of the additional things are necessary, one can just as well use the “gabor” mode of the above **Noise** or **Noise (3D)** maps.

It is a band-limited, filtered, sparse convolution noise based on the Gabor impulse function (see Lagae et al., SIGGRAPH 2012). The OSL Gabor noise is designed to have somewhat similar frequency content and range as Perlin noise. It is significantly more expensive than Perlin noise, but its advantage is that it correctly filters automatically based on the input derivatives.

These are the parameters of the shader:

Parameter	**Description**
UVW	The UVW coordinate. The default, when not connected, is object space.
Scale	A simple scale factor of the noise.
Bandwidth	The bandwidth of the noise
Impulses	The number of impulses per cell
Unsigned	When on, returns values in the range of 0 to 1. When off, returns values in the range -1 to 1
Filter	When on, uses the filtering features. Should probably always be on.
Anisotropic	• **Isotropic** (the default): Gabor noise will be isotropic. The direction vector is not used in this mode. • **Anisotropic along**: the Gabor noise will be anisotropic with the 3D frequency given by the direction vector (which defaults to (1,0,0)). • **Anisotropic Radial**: a hybrid mode will be used which is anisotropic along the direction vector, but radially isotropic perpendicular to that vector.
Direction	The direction vector for anisotropy

The shader has four outputs:

Output	Description
Col (RGB)	The output as a color
R / G / B	The individual R, G or B outputs of the color, i.e. the three noise values independently.

RandomizeBitmaps

Randomizes up to ten bitmaps with random position, scale, rotation, color shift, opacity, density, etc.

Conceptually, the maps starts by putting one copy of the bitmap in each unit square of texture space, after which it applies a set of random operations to it, making the outcome seem random. In the default values, *only* the position is randomized. If even that is turned to zero, the result will actually be a completely regular grid. But the map allows full control of how to randomize every aspect of each instance of the bitmap(s), making the fact the "starting point" is a regular grid impossible to detect.

However, for this to work correctly, a shader will have to look at the bitmap to be put in the currently shaded unit square of UV space, and also each of the neighboring unit squares around it, in case a bitmap from that square had been pushed into this square due to position or scale transformations. For this reason the **Overlap** parameter exists. The default of 1 causes each lookup to actually to check each of the nearest unit squares in each direction, checking in total a 3 x 3 grid (9 squares) with the "current" square in the middle. Sometimes, if very large position offsets, or very large scales are used, this may not be enough, causing the appearance of cut-off edges of bitmaps that are moved far or scaled up a lot. This can be solved by increasing the **Overlap** parameter to 2, but please keep in mind this entails looking up neighbors 2 steps in every direction, causing a 5x5 grid (25 squares) to be looked up. For a value of 3, a 7x7 grid (49 squares) are looked up, etc.

Obviously, this decreases performance a lot for each step up in the **Overlap** parameters value. Be cautious when increasing this!

A special note on the **Probability** parameter, which has three values. The first is simply an overall probability which will cause a instance of a bitmap to either appear or not. However, randomness in nature tends to have a "clumping" behavior, so the 2nd value is a "clumped probability" which applies a large-scale noise function on the probability. The 3rd value is the scale (in UVW space) of said noise.

These are the parameters of the shader:

Parameter	**Description**
UVW	The UVW coordinate for the map. Defaults to UV values from 3ds max map channel 1. To use another map channel, connect the UVW Channel map. Or connect any map or combination of maps computing a vector.
Input	The background onto which the randomized bitmap(s) is composited. This allows cascading several **Randomized Bitmap** together for extreme complexity.
InputAlpha	The background alpha value
Number of Files	How many of the below file names are used
File name N	Up to 10 files (**File name 0** to **File name 9**) with different images
Seed	The random seed
Probability	The vector defining the probability. See notes above • X is the overall "uniform" probability • Y is the more naturally looking "clumped" probability • Z is the scale of the "clumping"
PosRandom	A vector defining the position randomness. Only the X and Y values are used. For large offset, one may need to increase **Overlap** – see above.
ScaleMIn / ScaleMax	A vector defining the minimum and maximum scale. Only X and Y values are used. See **UniformScale** for details. For large scales, where images bleed strongly into neighboring grid cells, one may need to increase **Overlap** – see above.
UniformScale	• When **on**, the scale of each instance of a bitmap is a random value reaching from **ScaleMin** to **ScaleMax**, but the X and Y scale components always change in sync with each other, to maintain the aspect ratio of the image. • When **off**, the X and Y scale are randomized independently, which will then potentially alter the aspect ratio of the bitmap.
PixelScale	• When this is **zero**, the entire width of the image will (before any random scale) be considered the 1.0 scale and match one unit grid cell in UVW space, regardless of the pixel resolution of the image. • When this is **nonzero**, that many *pixels* of the image is considered the 1.0 scale. This allows randomization of many differently sized images. Note that this can easily cause large images to be scaled so big they easily bleed into adjacent grid cells, needing a larger **Overlap** – see above.
RotMin / RotMax	The minimum and maximum random rotation (in degrees). Each bitmap will get a random rotation between these two values.
HSVMin / HSVMax	The minimum and maximum HSV (Hue, Saturation and Value) tweaks as a vector, where • X is the amount of hue *shift* (default 0.0) • Y is the *multiplier* for saturation (default 1.0) • Z is the *multiplier* for the value (default 1.0).

	Each bitmap will get a random HSV shift between these two values, each component randomized independently.
AlphaMin / Max	The minimum and maximum alpha (opacity). Each bitmap will get random alpha between these two values.

The shader has following outputs:

Output	**Description**
Out	The output as a color
Alpha	The Alpha output.

Rivets

Also known as “the Steam-Punk shader”. Adds a set of rivets and an edge around each unit square in UV space (after applying the **Scale** parameter), with a set of randomizations applied to them for a more “distressed” look. This shader might be comically domain specific, but think about it as an example of just *one* of the infinite number of things that are *possible* using OSL shaders…

Note: Above image also uses the **UVW Row Offset** map to give the offset appearance.

These are the parameters of the shader:

Parameter	**Description**
UVW	The UVW coordinate for the map. Defaults to UV values from 3ds max map channel 1. To use another map channel, connect the UVW Channel map. Or connect any map or combination of maps computing a vector.
Scale	A simple scale factor to the UVWs. The default 0.5 will show 4 plate “tiles” on a 1x1 square of UV space on the object.
U_Rivets / V_Rivets	How many rivets are along the U and V axis of a unit square of UV space (after scaling). Technically, the shader divides each “plate” (unit square of UV space) into this many sub-tiles, and adds a rivet in the center of each of the tiles that touch the edge.

RivetRadius	The radius of a rivet (in relation to the size of each “rivet tile” as defined by the **U_Rivets** and **V_Rivets** parameters)
Dirt	The *additional* radius distance to add dirt *around* the rivet. If **RivetRadius** is 0.5 and **Dirt** is 0.1, the actual radius of the dirty area is 0.6
Seed	The random seed
PlateColor	The color of the plates
RivetColor	The color of the rivets
DirtColor	The color of the dirt around the rivets. Fades into the plate color.
SpaceColor	The color of the space between the plates
HoleColor	The color of the holes. Only seen if **RivetProbability** is less than 1.0
RivetCenter	A vector defining where, in each “rivet tile” each rivet is located. The Z coordinate is not used.
RivetRandom	A vector defining random position offset for each rivet within each “rivet tile”. The Z coordinate is not used.
RivetProbability	Amount of rivets that are still attached. Missing rivets are shown as a circle of the **DirtColor** with a smaller circle of **HoleColor** inside it.
HoleRadius	The radius of the hole
HoleEdge	The width of the hole “edge”. This is only affects the **Bump** output to give a beveled edge appearance on the hole.
BumpAmount	The strength of the bump effect on the rivets themselves. Only affects the **Bump** output.
BumpShape	The shape of the bump effect. A value larger than 1.0 makes a more “pointy” looking bump, a value lower than 1.0 a “rounder” looking bump. Only affects the **Bump** output.
PlateGap	The width of the gap between the plates, in UV units. The gap shows the **SpaceColor**
PlateEdge	The width of the plate edge, which only affects the **Bump** output to give a beveled edge appearance.
PlateWobble	A vector defining how much the plates “wobble” and bend. The X and Y coordinates warps the actual UV of each plate, and the Z affects the **Bump** output making the plate appear to buckle and bend. Note that while this m y seem like a regular UV distortion, it is a *different* distortion for each plate, making then appear warped in individual distinct ways.
WobbleScale	**PlateWobble** is driven by a noise function. This vector scales that noise function with relation to UV space.
Rivets follow plate wobble	• When **off**, the position of rivets are computed completely independently from any distortion of the plate geometry. • When **on**, any computation for rivet position is done within the distorted coordinate space of the plate, making rows of rivet to appear to follow bends in the plate.

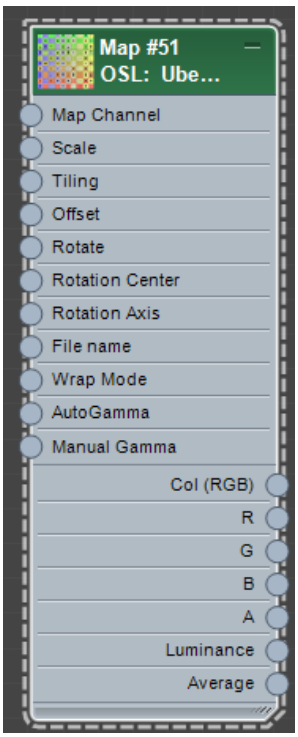
The shader has following outputs:

Output	**Description**
Col	The output color
RivetFac	1.0 when on a rivet, 0.0 when not.
HoleFac	1.0 when on a hole, 0.0 when not.
PlateFac	1.0 when on a plate, 0.0 when in a gap.
Dist	Radial distance on a rivet

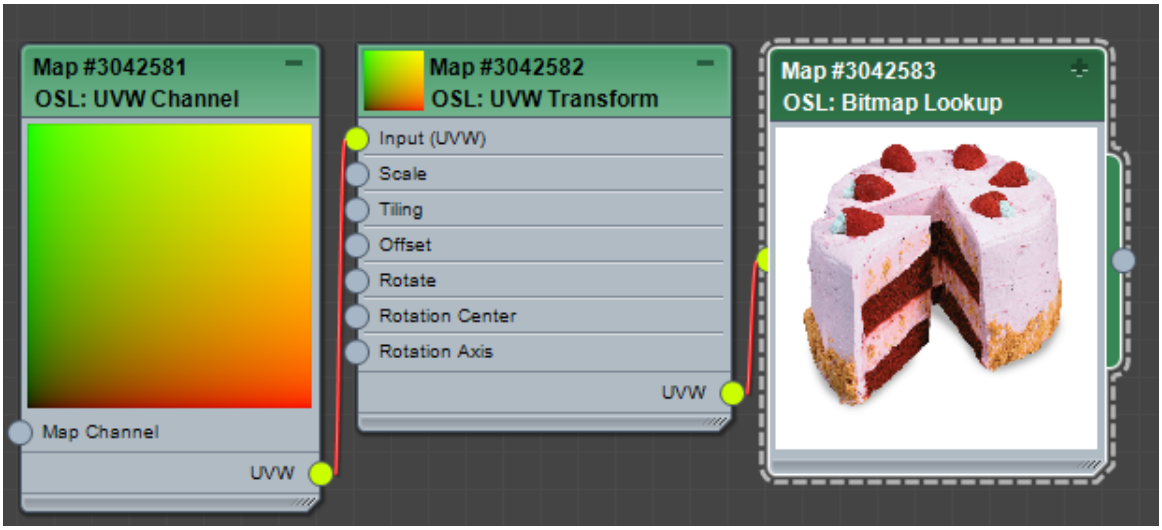
Bump	For connecting to the **Bump** input on a material. All bump effects appear in this output.
TileIdx	Index of a “rivet tile”. Use this to drive an indexed based randomizer to make variations from rivet to rivet.
PlateIdx	Index of each “plate”. Use this to drive an indexed based randomizer to make variations from plate to plate.

UberBitmap

UberBitmap is an all-in-one bitmap shader.



The generic “Bitmap Lookup” defaults to map channel 1. To use another map one must connect the **UVW Channel** map to its input. If one wants to tweak the coordinates in any complex ways, one may *also* need to use an **UVW Transform** map. Which makes you end up with this almost every time:



Since this use case of looking up a bitmap based on a map channel with some transformation applied is so *common*, the **UberBitmap** exists to combine the chain of **UVW Channel, UVW Transform** and **Bitmap Lookup** into a single shader. It is what is most similar to the classic max **Bitmap** shader.

These are the parameters of the shader, which is a superset of the three maps combined:

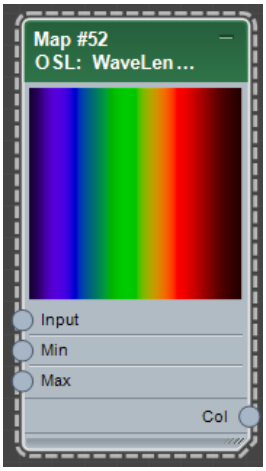
Parameter	**Description**
Map Channel	Which 3ds max map channel to use.
Scale	A uniform size multiplier (bigger values = bigger texture)
Tile	A per axis size divider (bigger values = smaller texture for that axis)
Offset	The offset of the coordinate
Rotate	How many degrees to rotate around the rotation axis
Rotation Center	The coordinate (in UVW space) of the center of rotation
Rotation Axis	A direction (in UVW space) around which rotation is performed
File Name	The name of the file to display
Wrap Mode	How to treat lookups outside the 0-1 UV range. Available values are: • "default" – whatever is the renderers default • "black" – returns black for outside points • "clamp" – repeats the edge pixels for outside points • "periodic" – repeats the texture in a cyclic fashion • "mirror" – repeats the texture in a mirrored fashion
AutoGamma	Attempts to automatically compute the gamma setting for the texture
Manual Gamma	When **AutoGamma** is off, the manual gamma to use

The shader has following outputs:

Output	**Description**
Col (RGB)	The output as a color
R / G / B / A	The individual R, G or B and Alpha outputs of the color.
Luminance	The weighted luminance of the color. Useful as the "black and white" output that is visually similar to the original image.
Average	The average value of the RGB color. Also "black and white" but does not discriminate between colors.

WaveLength

Computes an RGB color for a given wavelength



These are the parameters of the shader:

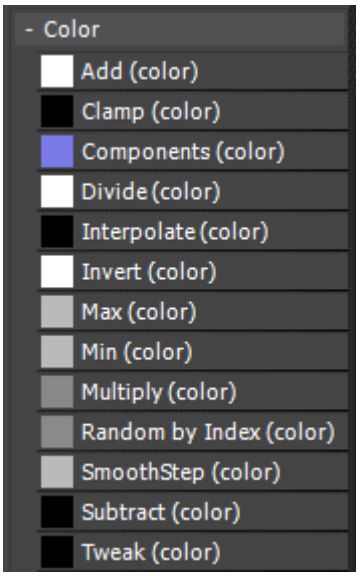
Parameter	**Description**
Input	The input value that drives the color
Min	The wavelength (nm) for when **Input** is 0.0
Max	The wavelength (nm) for when **Input** is 1.0

Subcategories

Shaders in the subcategories are simpler shaders that will not be documented on a per-parameter level. They should in theory be either self-explanatory or have enough tooltips to make their use self-evident.

Math/Color

Contains mathematical operations and helper functions relating to colors and color processing.

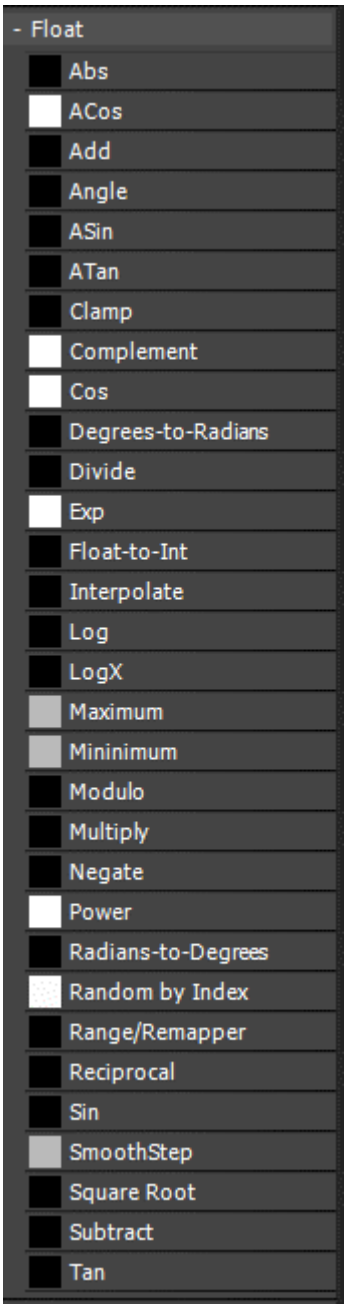


Map	**Function**
Add (color)	Add two colors, **A** and **B**, with an optional scale for each color.
Clamp (color)	Restrains the **Input** color between the **Min** and **Max** values
Components (color)	Break a color into individual **R**, **G** and **B** components or assemble a color from **R**, **G** and **B** values.
Divide (color)	Divide color **A** with color **B**, with optional 0-to-1 clamping.
Interpolate (color)	Interpolates between the **Min** and **Max** colors, depending on the value of the **Input** color. Note that interpolation is done per R, G and B component.
Invert (color)	Returns the photographic negative of the color (i.e. 1.0-**Input**)
Max (color)	Returns the highest R, G or B component from the input **A** and **B** colors.
Min (color)	Returns the lowest R, G or B component from the input **A** and **B** colors.
Multiply (color)	Multiply the R, G and B components of the **A** and **B** colors.
Random by Index (color)	Interpolates randomly between the **Min** and **Max** colors based on an integer **Index** and a random **Seed**. Used to generate e.g. random color variations per object or similar, by feeding in different per-object integer value.
SmoothStep (color)	For each R, G and B component, if **Input** is below **Min**, the result is 0.0, when above **Max** the result is 1.0, and in between the two, a smooth curve function causing a visually pleasing ramp between 0.0 and 1.0
Subtract (color)	Subtract the color **B** from the color **A**, with an optional scale value for each.
Tweak (color)	Tweak the **Input** color, by mapping the input range ranging from **InputMin** to **InputMax** to the output range ranging from **OutputMin**

	to **OutputMax**. Between these operations the colors **HSV** value can be modified by *adding* the H value to the colors hue, and *multiplying* the S and V values with the saturation and value of the color respectively. The result can be optionally clamped to the 0.0-1.0 range.

Math/Float

Contains mathematical operations and helper functions relating to floating point numbers. Note: All trigonometric functions work in units of *radians*.

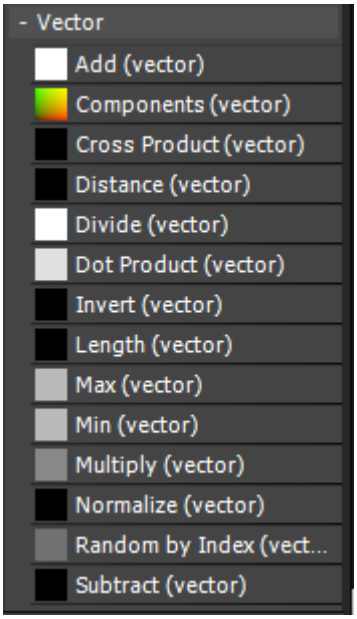


Map	**Function**
Abs	The absolute value (i.e. the value with any negative sign removed)
ACos	The Arccosine (inverse Cosine) trigonometric function.
Add	Adds **A** and **B**
ASin	The Arcsine (inverse Sine) trigonometric function

ATan	The Arctangent (inverse Tangent) trigonometric function
Clamp	Clamps the **Input** value between the **Min** and **Max** values
Complement	Returns the complement number (1-**Input**)
Cos	Returns the Cosine of **Input**
Degrees-to-Radians	Converts an **Angle** in degrees to radians.
Divide	Divides **A** with **B**. Division by zero simply returns zero.
Exp	The natural logarithm *e* raised to the power of **X**
Float-to-Int	Convert a floating point value to an integer. The rounding can be done by *floor* (nearest lower integer in the direction of negative infinity), *ceil* (next higher integer in the direction of positive infinity), *round* (standard rounding) or *trunc* (simply throwing away decimals) operations.
Interpolate	Interpolate between the **Min** and **Max** values based on the **Input** value.
Log	The natural logarithm of **Input**
LogX	The logarithm of **Input** in an arbitrary **Base**
Maximum	The highest number of **A** or **B**
Minimum	The lowest number of **A** or **B**
Modulo	The remainder (modulo) of dividing **A** with **B**
Multiply	Multiplies **A** with **B**
Negate	Returns the **Input** with the sign reversed (positive to negative or vice versa)
Power	Raise **A** to the power of **B**
Radians-to-Degrees	Converts an **Angle** in radians to degrees.
Random by Index	Returns a random value between **Max** and **Min** for each integer **Index** and a random **Seed.** Useful for making per-object randomizations.
Range/Remapper	Takes the **Input** value and remaps the range from **Input Range Start** to **Input Range End** to the output range going from **Output Range Start** to **Output Range End**, applying a curve (power function) in between, where a **Remapping Curve** of 1.0 means a linear transformation. The output value can be optionally **Clamped to Output Range**, otherwise if the **Input** value strayed outside the input range, this could cause the Out value to stray outside the output range.
Reciprocal	Return 1.0 divided by the **Input**
Sin	Returns the Sine of **Input**
SmoothStep	If the **Input** value is below **Min,** returns 0.0. If it is above **Max**, returns 1.0. In between these values, returns a smoothly interpolated visually pleasing curve.
Square Root	Returns the square root of **Input**
Subtract	Subtracts **B** from **A**
Tan	Returns the Tangent of **Input**

Math/Vector

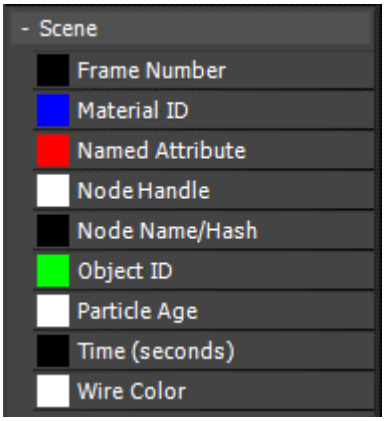
Contains mathematical operations and helper functions relating to vectors, points or normals (all three treated as a “vector” type)



Map	**Function**
Add (vector)	Add vectors **A** and **B** (with optional scaling)
Components (vector)	Allow getting or setting individual X, Y or Z coordinates of a vector.
Cross Product (vector)	Take the cross product of **A** and **B**. A cross product of two vectors is a third vector perpendicular to both. If the incoming vectors are unit length, so will the resulting vector be.
Distance (vector)	Take the distance between **A** and **B** seen as points.
Dot Product (vector)	Take the dot product of **A** and **B**. The dot product is the cosine of the angle between the two vectors times the product of the length of both, which means that if both vectors are unit length, it’s simply the cosine of the angle between them.
Invert (vector)	Inverts a vector by negating each of its components.
Length (vector)	Computes the length of a vector
Max (vector)	Returns the maximum of each of the X, Y and Z components of the vectors **A** and **B**
Min (vector)	Returns the minimum of each of the X, Y and Z components of the vectors **A** and **B**
Multiply (vector)	Multiplies **A** with **B**
Normalize (vector)	Returns the normalized version of **Input**, i.e. the vector that has the same direction, but re-scaled so it has the length 1.0, also known as “unit length”.
Random by Index (vector)	Returns a random vector between **Max** and **Min** for each integer **Index** and a random **Seed**
Subtract (vector)	Subtract vector **B** from **A** (with optional scaling)

Scene

Maps for getting data about the scene



Map	**Function**
Frame Number	Returns the current frame number with an optional **Offset**
Material ID	Returns the Material ID of the current face (or the particle ID for particles), either as a random color, or an integer **Index**.
Named Attribute	Get any OSL named attribute. The available named attributes are described below. For each of the output types there is a matching "Default" input, which will be the output value if that attribute is missing for some reason (or exists, but is the wrong type). The **User Defined Property** checkbox simply prepends "usr_" to the attribute being requested from OSL, so filling in "*Foo*" in **Attribute** with this checked will actually ask OSL for the "*usr_Foo*" attribute.
Node Handle	Returns the Node Handle of an Instance, which is a value suitable for driving indexed randomizers with. Note However that node handles might not be persistent across e.g. merge-ing or XRef-ing of scenes, or even across certain complex scene edits.
Node Name/Hash	Returns the name of the node, or perhaps more interesting, the **Hash** of the name. This can also be used to drive indexed randomizer, but this has the guarantee, that a given *name* will *always* return a given Hash, so as long as the name is persistent, so is the hash.
Object ID	Returns the number known as the "Object ID" in the object properties dialog (technically, the **GBufID**).
Particle Age	Returns the normalized particle age, 0.0 at birth and 1.0 at death of the particle. For convenience a **Scale** can be applied to the value.
Time (seconds)	Returns the scene time in seconds, with an optional **Offset**
Wire Color	Returns the wireframe color of the object.

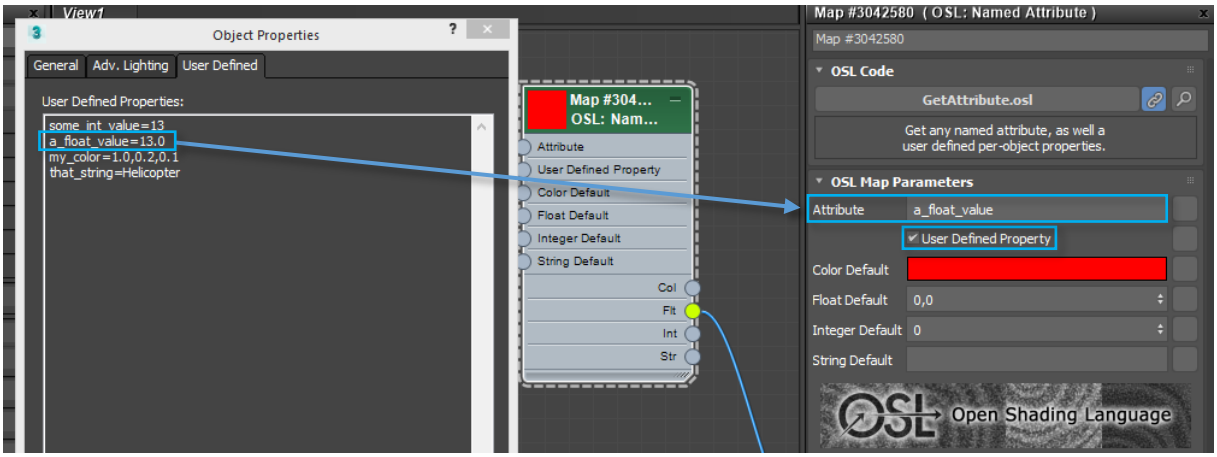
About Named Attributes

The Named Attribute shader can return any named attribute that the current OSL execution environment exposes. However, it is important to understand *which execution environment that is*. For example, when rendering inside of 3ds max's Scanline renderer, or any other renderer supporting the classic shader interface, it is 3ds max own *built in* execution environment being used, whereas when rendering in e.g. the Arnold renderer, it is the Arnold execution environment being used.

This makes the list of attributes available renderer-dependent. There is a recommended list, but it cannot be absolutely guaranteed that every renderer implementation in max follows the recommended list.

For this reason, it is better to use the special maps for getting scene values, such as the “Node Handle” map. In reality, it is simply getting a specially named attribute called “nodeHandle”, but the advantage of using the specific Map rather than the general “Named Attribute” map, is that the “Node Handle” map can in principle be updated in case a certain new renderer behaves differently and names its attributes in a different manner, whereas using the “Named Attribute” map and typing in “nodeHandle” in the **Attribute** parameter, will probably work… but might not.

User Defined Attributes



In a renderer with a compliant OSL execution environment, the Named Attribute can also return User Defined attributes. These are values typed into the Object Properties dialog on the “User Defined” tab. Simply typing a name/value pair like for example “Foo=13.0” will expose to OSL an attribute on that object named “usr_Foo” with the floating point value 13.0.

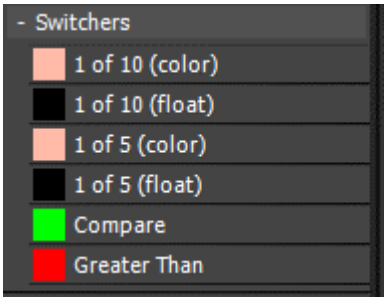
To make this easier to use, one can simply type “Foo” in the **Attribute** parameter and check the **User Defined Property** checkbox, which prepends the “usr_” prefix under the hood. Values of type integer, string, float, or three comma separated floating point numbers (for colors or vectors) are supported.

The type of the data is defined by the format of the value, i.e.

Type	**How it is written**
Integer	13
Float	13.0
Color or Vector	1.0,0.5,0.1
String	Anything else

Switchers

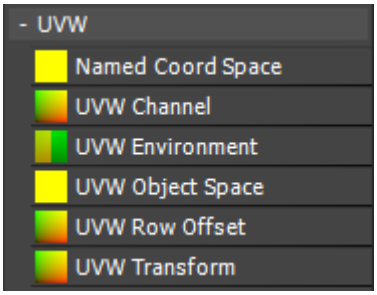
Maps that switch between / choose other maps due to various criteria like index numbers, conditions, and similar.



Map	Function
1-of-10 (color / float)	A map each for float or color values. Has ten inputs, with an **Input index to use** value that selects which of the 10 inputs are used, so that input index 0 will return the value at **Input 0** and so on. If **Start index** is nonzero, that is the actual index that will return **Input 0**, e.g. for a Start Index of 10, it is the input index value of 10 that will return the **Input 0** value. This is how several switchers can be cascaded together to return more than 10 values – plug one switcher into the Outside of the other, utilizing the **Start Index** to change its range. When the **Range: Number of inputs** is zero, any input index outside the 0-9 range (taking the **Start Index** offset into account) will return the **Outside** value. However, when it is nonzero, only that many of the inputs will be used, and an index higher than that will wrap around to the first input again. So if it is set to 3, an input index that would have normally mapped to **Input 4**, will instead wrap around to **Input 0**
1-of-5 (color / float)	Works identically to the 1-or-10, except it has only 5 inputs, for efficiency.
Compare	Compares two numbers **A** and **B** and returns three different (color) values depending on if A is greater than B, the numbers are “equal” (within the **Equality threshold**) or B is greater than A.
Greather Than	A simpler shader that simply returns two different (color) values depending if **A** is greather than **B** or not.

UVW

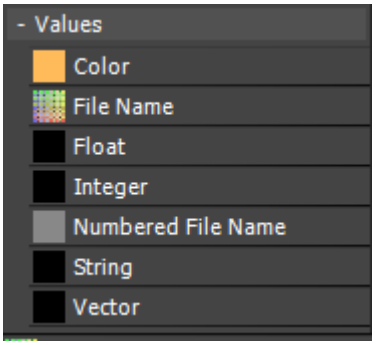
Maps for processing texture-mapping coordinates



Map	**Function**
Named Coord Space	Converts the input (by default the shading point) from “common” coordinates to the given named coordinate space. Available choices are “world”, “object”, “screen”, “raster” and “NDC”.
UVW Channel	Returns the UVW values from a 3ds max map channel. Note that channel 0 is the vertex color channel.
UVW Environment	Returns UVW coordinates suitable for environment mapping. Supports standard “spherical” mapping, but also “mirrorball” mappings, where the environment map is a picture of a mirror ball. Only makes sense when used in the scene environment slot.
UVW Object Space	Converts the input value (by default the shading point) to Object Space. Effectively a shorthand for using the **Named Coord Space** shader with “object” as the coordinate space.
UVW Row Offset	Offsets rows of the UV space. Useful to make bricks and offset tile patterns. Takes the input UVW coordinates (by default map channel 0), applies **Scale** and **Tiling** to it and divides the resulting UV values in **RowCount** number of rows (each the hight of 1 in the V direction), offsetting each row in the U direction by **Offset** plus **RandomOffset**
UVW Transform	Takes the input UVW coordinate (by default map channel 0), and applies a **Scale** and **Tiling** to it, where **Scale** is a uniform size multiplier (bigger values = bigger texture) and **Tile** is a per-axis size divider (bigger values = smaller texture for that axis), adds an **Offset** and then rotates the UVW coordinate **Rotate** degrees around an axis centered at **Rotation Center** and pointing in the direction of **Rotation Axis** in UVW space.

Values

Maps that pass-through or expose plain values, for connecting shared values to other maps, so that a given value only have to input in one place, but can be connected to multiple places.



Map	Function
Color	Returns a color value
File Name	Returns a file name value, which entails there is a file selection button to pick the file, and the picked file will be consider an asset of the scene.
Float	Returns a floating point value
Integer	Returns an integer value
Numbered File Name	Takes an input file name, and converts it to one with a given four-digit frame number added before the dot of the extension. If **Replace** is on, the 4 characters before the dot is *replaced* by the number (useful if you picked file0000.exr already and want to replace that number). If it is off, the number is simply inserted before the extension dot, making the file name 4 characters longer. If **Do Preview** is checked, the Preview output will contain the image. This is purely a trick to make the image show up in the thumbnail of the 3ds max SME editor, and not intended to be used for actual rendering.
String	Returns a string value
Vector	Returns a vector (point, or normal) value