3Delight's OSL Metadata

Shader Metadata

The shader metadata must be provided between the shader name and its parameter list. For instance:

```
surface voronoi

    [[
    string tags[1] = {"texture/2d"},
    string maya_typeID = "0x00"
    ]]

    ( ... )
```

The supported shader annotations are:

`string niceName`

Specifies the string to use for the shader name in the user interface, when possible.

`string tags[n] = { tag1, ... tagN }`

An array of n strings defining tags for the shader. The supported tags are:

`surface`
`displacement`

The shader will be considered a surface shader or a displacement shader, respectively. Tagging a shader with one of these values will have *3 Delight for Katana* create a *Network Shader* node along with the surface shader or displacement shader, and connect it appropriately. The tag will also allow the shader to be listed as surface material or displacement material in a *Material* node.

`texture/3d`

The shader will be considered a 3D texture. 3D texture will have a 3D placement matrix added automatically in all plug-ins.

`texture/2d`

The shader will be considered as a 2D texture.

`utility`

The shader will be considered as a utility node.

`hidden`

The shader will not be listed in the shading node menu.

Shader Parameter Metadata

Shader parameter metadata is provided between a parameter's default value and the comma that ends its declaration. For instance:

```
float i_jitter = 1.0 [[ string attribute = "jitter" ]],
```

Attribute specification Metadata

The following metadata provides details that are used when generating the attribute of the shading node that corresponds to a given shader parameter.

`string attribute`

Specifies the name of the attribute that corresponds to this parameter. There are many reasons to use these attribute - parameter mapping: because the software or OSL imposes restrictions (e.g. an attribute named "color" in *Maya*, which is a reserved word in OSL), because the shader has different parameter naming conventions than what is expected in the software, or because the required attribute in the software is part of a complex attribute structure that does not have an OSL counterpart.

When this metadata are provided, the parameter name is used directly to define the attribute name.

The `*none` special value indicates that there will not be any node attribute generated for that shader parameter.

`string default_connection`

Specifies the name of a shader to be connected to this shader parameter when no connection exists. The only supported value is `uvCoord`. This should be used for `float[2]` shader parameters that receive the `st` coordinates; it centralizes the `s, t` lookup in a single shader which improves performance.

`int hidden`

Maya-specific
When this is set to a non-zero value, a *Maya* attribute will be defined for the given shader parameter, but it will not be shown in the *Attribute Editor*, in the *Channel Box Editor,* nor in the *Node Editor*. If you only want to avoid getting a gadget for the parameter, see the `widget` meta-data below (which also works in *Katana*).

int skip_init

Setting this metadata to 1 will will prevent the attribute value from being passed as a shader parameter value when rendering. Only incoming connections to this attribute will be defined when rendering. This allows passing implicit values to shader parameters, such as:
`normal normalCamera = N [[ int skip_init = 1 ]],`

`string help`

Specifies a string that will be shown in a help box or tool tip. This is currently only supported in *Katana*.

`string label`

Specifies the label of the widget that controls the attribute. In *Maya*, this only has an effect in the *Attribute* Editor. If no label is specified but `niceName` is, the later will be used as a label too.

Using `niceName`, `page` and `label` together allow a parameter named `baseLayerColor` to be labeled simply as *Color* in a *Base Layer* page, and appear as *Base Layer Color* in the *Node Editor* when the DCC application allows such feature (e.g. Maya).

`string lock_op`
`string lock_left`
`string lock_right`

Katana-specific
Define a lock operator, its left operand and its right operand, respectively. Lock operations allow the gadget to become insensitive or locked when the operator returns true. The various operators are listed here. `lock_left` should be set to an attribute name, and `lock_right` to a value appropriate for the chosen operator. For example, this would make the gadget of an attribute insensitive when a `barnDoor` attribute would not be set to 1:
`string lock_left = "barnDoors", string lock_op="notEqualTo", int lock_right=1`

`float min, float max`
`int min, int max`

Define the minimum and maximum values of a numeric parameter, respectively. When both are defined, the attribute gadget will be made of a numeric field and a slider. In *Maya*, this configures the attribute with the specified hard minimum and / or maximum value. In *Katana*, there is no way to enforce minimum or maximum values on attributes. Nevertheless, having a slider shown helps providing information about the parameter's useful range of values. Providing only `min` or only `max` has no effect in *Katana.*

`string niceName`

Specifies the attribute name to use in the user interface, when possible.

`string options`

This metadata provides extra options for specific widget types. See `widget` for details.

`string page`

Specifies the name of a collapsable section that is used to group together related attributes. Nested pages can be defined by setting this metadata to a string containing the path to the given page. For instance:
`string page = "Parent Page.Child Page"`

`float slidermin, float slidermax`
`int slidermin, int slidermax`

Specifies the range of of the slider widget that controls the attribute.

`int texturefile`

This meta-data should be set to 1 for parameters that specify a texture file name.

`string widget`

Specifies the type of gadget that will show the attribute's value(s). By default, a gadget appropriate for the attribute type is generated. In *Maya*, matrix attributes are not shown in the *Attribute Editor*. The following values are supported when it is required to override the default gadget:

`checkBox`

A check box widget. Implies a boolean attribute type.

`filename`

A combination of widgets suitable for a filename. This usually consists of a text field and with a browse button that produces a file browser dialog when clicked.

`mapper`

An option menu with a list of items that have an associated numeric value. The list of item labels and values are defined using the `options` m etadata. In *Maya*, this widget implies an enum attribute. The enumeration list and values is expected to be defined using the `options` metada ta. Set options to a string containing one or more `<label>:<value>`, separated by `|`. For example:

```
string options = "No Operation:0|Multiply:1|Divide:2|Power:3"
```

`navigation`

Maya-specific
A combination of widgets suitable to handle a connection from another node. The attribute is shown as a text field displaying the name of the connected node and a "map" button that brings up a windows allowing the user to select the type of a new node to create and connect to this attribute.

`newScenegraphLocation`

Katana-specific
The attribute will be shown with a gadget suitable for defining a new scene graph location; the created location path is passed as the shader parameter value.

`null`

No widget will be created for this attribute. Note that the attribute will still be defined.

`popup`

The string shader parameter will be presented as an option menu with predefined menu items. The menu items are specified by the `options` meta-data. Set `options` to a string containing the names of the menu items, separated by `|`. For example:

```
string options = "clamp|black|mirror|periodic"
```

`floatRamp`
`colorRamp`

The ramp widgets require more than one shader parameters and are explained in a dedicated section.

Getting the texture coordinates

Because retrieving the `s`,`t` coordinates is a costly process, it is best to centralize this operation in a single shader and share the results with all shaders. For a shader to benefit of this, it may declare a parameter similar to this:

```
float uvCoord[2] = { 0, 0 }
[[
        string default_connection = "uvCoord",
        string label = "UV Coordinates",
        string widget = "null"
]],
```

Ramp widgets

This widget is still DCC dependent and is still WIP. The method shown below will work in all DCCs supported (Maya, Katana and Houdini).

The ramp widgets in Maya and Katana both require 3 different attributes. They also have diverging expectations and features. Note that The triplet of shader parameters required to present a Maya-style float ramp can be declared as follows:

```
float i_value_Position[] = { 0, 1 }
[[
        string katana_attribute = "value_Knots",
        string maya_attribute = "value.value_Position",
        string related_to_widget = "maya_floatRamp",
        string widget = "null"
]],

float i_value_FloatValue[] = { 0, 1 }
[[
        string katana_attribute = "value_Floats",
        string maya_attribute = "value.value_FloatValue",
        string label = "value",
        string widget = "maya_floatRamp"
]],

int i_value_Interp[] = { 1, 1 }
[[
        string katana_attribute = "value_Interpolation",
        string attribute = "value.value_Interp",
        string related_to_widget = "maya_floatRamp",
        string widget = "null"
]],
```

```
float i_color_Position[] = { 0, 1 }
[[
        string katana_attribute = "color_Knots",
        string maya_attribute = "color.color_Position",
        string related_to_widget = "maya_colorRamp",
        string widget = "null"
]],

color i_color_Color[] = { 0, 1 }
[[
        string katana_attribute = "color_Colors",
        string maya_attribute = "color.color_Color",
        string label = "color",
        string widget = "maya_colorRamp"
]],

int i_color_Interp[] = { 1, 1 }
[[
        string katana_attribute = "color_Interpolation",
        string maya_attribute = "color.color_Interp",
        string related_to_widget = "maya_colorRamp",
        string widget = "null"
]],
```

Maya-Specific

Maya requirues both a typeID and a template file. The following meta-data allow you to to automatically take car of these.

`string maya_typeID`

> Specifies the *type ID* used for the node registration. This is a integer that *Maya* uses to identify the node type, most notably when saving a scene in the *Maya Binary* format. Each OSL shader that defines a given shading node type should be assigned a unique type ID. You can chose a value between `0x0000` and `0x007F`, or between `0x7F01` and `0x7FFF` for your shading node type.
>
> The IDs from `0x0000` to `0x7FFF` are reserved for node types that are used internally in a studio. *3Delight for Maya* will generate a type ID between `0x0080` and `0x7F00` if no `maya_typeID` annotation is provided. You can also request your own reserved node ID range to *Autodesk*, for free. This is also the recommended solution if you intend to share your nodes with users outside your studio.
>
> You may use any ID between `0x0000` and `0x7FFF` if you always provide the `maya_typeID` annotation for every custom OSL shader you define. In this case *3Delight for Maya* will never need to generate a type ID.

`int maya_generateAETemplate`

Setting this to 0 allows providing a complete custom template using a MEL file, just like any other node. Setting this metadata to 1 (or not specifying it at all) will have *3Delight for Maya* generate an Attribute Editor template automatically based on the shader parameters' metadata.