OSL Patterns

Critters, using a shading network of nine OSL shaders

RenderMan supports OSL natively with the limitations listed below.

Known Limitations

Not all of the operations of OSL are supported. The following are currently not supported:

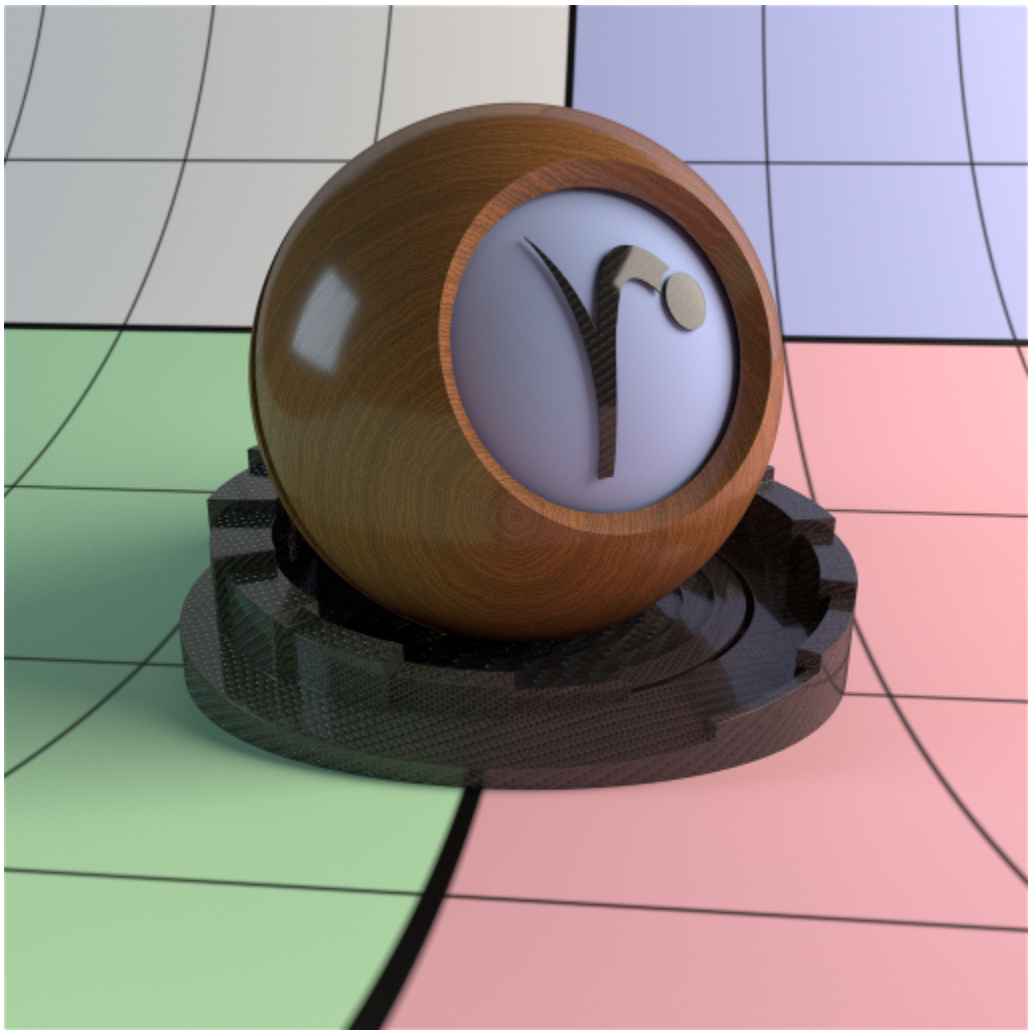
- material closures
- trace
- environment
- pointcloud_search
- pointcloud_get
- pointcloud_write
- setmessage
- getmessage
- surfacearea
- raytype

Also note that currently only the standard three types of OSL texture filters are supported. OSL shaders using incomplete or unsupported features can cause the program to terminate.

OSL patterns currently support many of the features of OSL, but it is important to note that they **do not support the use of material closures** . Instead the plugin is designed to create pattern results that are subsequently utilized by connecting the OSL outputs to other patterns and also feeding the OSL shader results into the various inputs of Bxdfs and Displace shaders. Bxdfs and Displace shaders take the place of closures in an extensible way and allow for more complicated layering than the linear combinations provided by OSL closures.

Example

An example of this is provided below, wherein two different OSL shaders are used: one to create an wood texture (adapted from "Advanced RenderMan") and another to create a 2-layer material that looks like carbon fiber. For each surface type, the same material shader is used, but two very different OSL shaders are used to create the different looks.



Oak and carbon fiber, using two OSL shaders wired into PxrDisney

The RIB file for the image above is provided in the `/lib/examples/RIS/scenes/pattern/osl/` directory of your RenderMan Pro Server installation, the OSL shaders in the `/shaders/` subdirectory therein.

OSL Performance Notes

There are a couple of performance considerations to be aware of when using OSL in RenderMan. Firstly, by default OSL executes all connected nodes of a shader network for a given point before moving on to the next instead of executing each node for all points like other Pattern nodes. If you access a lot of textures in the network, this means the texture cache will have to support having all the textures in the cache at once instead of just a single texture in the cache to maintain the same performance. For situations where a C++ Pattern node could re-use some setup across all points, OSL will sometimes have to repeat that setup for each point.

By default, OSL executes shaders in a "vertical" fashion, instead of across a grid of points like other PxrPattern nodes. This can result in a loss of efficiency if many textures are accessed and the setup for the textures is executed continuously across an group of shading points.

However, there is the *default* to use **"Batched"** mode. This mode will run multiple shading points via hardware supported vectorized instructions. Speedup with batched OSL is workload dependent -- scenes with a high amount of ray divergence and thus less shader coherence will see less improvement. Scenes with high coherence such as single-bounce or PxrVisualizer and OSL shaders of a significant size will see more improvements. We take care to perform the optimization only when necessary to provide the best performance overall.

The SIMD version of OSL is optimized to execute 16 samples at a time with Intel® Advanced Vector Extensions 512 (Intel® AVX-512). It can also execute using Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® Advanced Vector Extensions (Intel® AVX), or Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) at reduced performance levels. The SIMD version of OSL supports all OSL 1.8 language features and library calls supported by RenderMan.

It is enabled with the following *user* option

```
Option "user" "int osl:batched" [1]
```

Also, the cost of compiling the shaders and shader networks at runtime can become a significant, especially since this compilation phase will prevent multiple threads from running at their highest efficiency. Normally for scenes that take a while to render, this cost is effectively amortized. However, if there are hundreds of thousands of unique shader instances, that cost can start to impact final render times. The Batched mode may have a greater impact on startup time than non-batched but for complex scenes the overall benefit is worthwhile.

One of the benefits of using OSL, however, is the ability to create optimized, native machine code based on the uniform input parameters to the shader. For complex networks and code that is written to take advantage of these optimizations, the performance improvements can be significant. This makes it possible to write an shading nodes with all of the bells and whistles you could want, and as long as you're willing to pay for the compilation at the beginning of the render, you can have it efficiently re-compiled for each different use with only the features that are needed at render time.

You can better understand OSL in your scene by outputting stats at varying levels. Note that higher verbosity will impact render times as there is overhead to print messages during render. So it's advised only to use for data mining your scene and not final rendering or lookdev tests where speed is important. The verbosity matches that of standard OSL messaging: 0 nothing, 1 only severe, 2 only severe and error, 3 severe,error,message, 4 severe,error, message,warning, 5 severe,error,message,info These stats are written to the Plugins tab of the Advanced XML stats in RenderMan. Stat output at level 3 and higher adds timing to the stats and can impact performance more than lower levels. Levels 4 and higher should be used in tandem with RenderMan support as well as information on your use of the batched mode or default.

Basic OSL shader writing

Note a number of OSL attributes are available to be set within a RIB file. The following attributes are supported:

You create a shader in one or more text files, typically with the .osl extension. You then compile these into shader object files with a .oso extension using the osl compiler, oslc. For convenience, oslc is shipped with RenderMan distribution. These shader object files can then be used directly as Pattern shaders in RenderMan.
We'll go through the following simple shader as an example to illustrate the basics of writing an OSL shader and how that shader is integrated into RenderMan:

```
shader myMixShader (color a = color(1,0,0),
                    color b = color(0,1,0),
                    float mixVal = .5 [[ int lockgeom=0 ]],
                    output color resultRGB = color(1,1,1) ) {
    float finalMix = clamp(mixVal,0.0,1.0);
    float doMix = 1.0;
    getattribute("allowMixing", doMix);
    float mixExponent = 1.0;
    getattribute("builtin", "curvature", mixExponent);




    if (doMix != 0) {
        finalMix = pow(finalMix, mixExponent);
        float mixRemainder = 1.0 - finalMix;
        resultRGB = a * finalMix + b * mixRemainder;
    } else {
        resultRGB = a;
    }
}
```

The first line contains the shader type, the shader name, and the beginning of the parameter list. RenderMan only supports the generic “shader” shader type, not the more specific types of “surface”, “displacement”, “light”, and “volume” shaders used by other renderers.

Next we have the list of parameters. Each parameter must have a default initializer. Metadata can be specified within the double brackets [[ ]] to provide extra information about the parameters to your renderer and other content creation applications. In this case, the metadata int lockgeom = 0 means that the value mixVal can be interpolated from a mixVal variable attached to the geometry. Currently, RenderMan allows attaching variables of all types supported by OSL except for int and structs.

The results of the shader, in this case the color resultRGB, are connected to the rest of the shading network via parameters declared as outputs. These outputs can be connected to Bxdfs and Displace shader inputs, or other OSL or C++ Pattern shader inputs. This interface is provided in place of the standard closures, which are not supported within RenderMan.

One of the first things the shader does is to declare a local variable, finalMix. It applies the OSL function clamp() to the mixVal input, and stores it in finalMix. Inputs are not writeable in OSL, so we must store it in a second temporary value. Don't worry about the extra variable, though – the render-time optimization step in OSL is good at getting rid of them, so feel free to use temporary variables like this to make your shader code clearer as well.

RenderMan supports almost all of the standard functions built into OSL like clamp() -- you can refer to Known Limitations for the list of unsupported functions and the the OSL documentation for usage of the supported functions. You can write your own functions in OSL, however there is currently no interface for directly linking in functions written in another language such as C++. For that you'll need to write a C++ pattern and pass values via the parameter lists between C++ and OSL. See the section below for details and limitations of mixing C++ and OSL Patterns.

The next bit of interest is the getattribute() call. It allows fetching attributes set for the currently set piece of geometry by attribute name. It also supports an object name to indicate where the attribute should come from. RenderMan supports using “global,” “primvar,” and “builtin” object names, as well as no name or the empty string “”. Using no name or the empty string will fetch RenderMan attributes for the current piece of geometry. Using “global” will fetch options. Since RenderMan Attributes and Options are divided up into categories, so to fetch them you prefix the attribute name with the category and a colon. For exampe, Attribute “dice” “float micropolygonlength” can be fetched using getattribute(“dice:micropolygonlength”). Using “primvar” will only fetch values stored on the geometry itself, separate from the RenderMan attribute state. Using “builtin” will fetch the additional built-in variables provided by RIS but not part of the default global variables of OSL.

Mixing C++ and OSL Patterns

In order to get C++ and OSL patterns to interact with each other, we need to tell the OSL patterns to not assume their inputs are constant and may be fed by a C++ Pattern output. If you want to allow connecting a non-OSL output to an OSL input, you must set the metadata flag [[int lockgeom = 0]] for that input.

One of the advantages of using OSL is that large networks of Patterns are compiled together on the fly as they are used in the renderer. That means the compiler can take advantage of the current settings for the shaders and optimize away things that aren't used for any given object. However, the OSL compiler can't “see” into C++ based Pattern nodes, and so having them mixed in can defeat some of the optimization process. Additionally, OSL Patterns are grouped together and executed together, while C++ Patterns are executed one at a time. If a C++ Pattern is in the middle of a network, RenderMan may have to split the OSL Patterns into two groups – those that must execute before the C++ Pattern and those that must execute afterwards. Unfortunately, RenderMan currently cannot pass implicit derivatives between these two groups of nodes, and so if something like your texture filtering depends on derivatives of signals upstream, they may be distrupted by mixing in C++ nodes. For this reason, interleaving C++ nodes in the middle of networks is discouraged – though they can be downstream of all OSL nodes without detriment, and entirely upstream of all OSL nodes with little impact.

OSL implementation notes:

Binding primvars on geometry in PRMan is fairly simple: just mark the parameters in the shader with `metadata [[int lockgeom = 0]]`. This will tell the shading system to consider this variable as varying input and it will bind the variable in the shader if it is present on the geometry. Also note you can bind outputs of other Pattern nodes and the OSL pattern node will automatically bind the output of one Pattern node to the parameter of the OSL shader. Likewise the output of an OSL shader can be referenced by other nodes, be they Pattern, Bxdf or Displace nodes. Simply reference them by name.

`[[int lockgeom = 1]]` means roughly, "allow OSL jit to lock out geometric binding and assume that the shader parameter is the constant value". That allows constant propagation to work harder, but prevents ever executing the shader with a varying in put for that parameter. We need to force this behavior at jit time if we're going to wire in C++ inputs that vary, but at this point, prman is already taking care of that for you in that case.

The second part indicates that if you've locked out geometric binding, lockgeom=1, then the automatic binding of a primvar to a shader input won't happen. That's the default, so you need set an [[int lockgeom=0]] metadata tag if you have an input that was 'color primColor', and a vertex variable on some of geometry named 'primColor', and you expect that to be wired into the shader.

There are a couple of options that control the behavior of OSL and it's usage in RenderMan.

- Option “user” “int osl:verbose” [0-5] – Controls the verbosity of OSL message echoed by RenderMan. 0 is off, and 1-5 each add one more class of OSL messages to what is printed. Those classes are 1 – SEVERE, 2 – ERROR, 3 – MESSAGE, 4 – WARNING and 5 – INFO.
- Option “user” “int osl:statisticslevel” - This is similar to the “osl:statistics:level” attribute, but it controls the level of statistics tracked by RenderMan for the integration of the OSL shading system as a whole.

Note a number of OSL attributes are available to be set within a RIB file. The following attributes are supported:

- `Attribute "user" "uniform int osl:debug" [0]`
- `Attribute "user" "uniform int osl:optimize" [1]`
- `Attribute "user" "uniform int osl:lockgeom" [1]`
- `Attribute "user" "uniform int osl:debug_nan" [0]`
- `Attribute "user" "uniform int osl:debug_uninit" [0]`
- `Attribute "user" "uniform int osl:compile_report" [0]`
- `Attribute "user" "uniform int osl:statistics:level" [0]`

Refer to the OSL documentation for details on their usage. You can read more about the current version of OSL by visiting OpenShadingLanguage.

Accessing Shading Contexts

It's possible to access RenderMan specific contexts that C++ shaders can see on the RixShadingContext via the `getattribute` call with "context" as the object.

```
string mymode = "unknown"; getattribute("context", "shadingMode", mymode);
if (mymode == "displacement") ...
```

The values for "shadingMode" correspond to the RixSCShadingMode in RixShading.h:

- presence
- opacity
- scatter

- volumeTransmission
- volumeScatter
- emission
- bake
- displacement

If you desire grid-like shading instead of ray hits, rather than use shadingMode, you can query a boolean context attribute which returns either 1 or 0 and includes:

- eyePath
- lightPath
- primaryHit
- missContext
- reyesGrid