3Delight's OSL Support

3Delight supports all the required functions to properly run OSL shaders. That being said, the philosophy of writing OSL shaders for *3Delight* differs very slightly from other renderers. In a nutshell, the 3Delight rendering core is organized so that OSL shaders can remain as abstract as possible. For example, it is discouraged (and indeed wrong) to use functions such as `backfacing()` to write shaders. Also, some shadeops have seen their definition slightly changed to simplify shader writing or to allow 3Delight make a better job at sampling the final image.

Content:

Differences in *Form*

Consider the following simplified "glass" shader and compare it to the "glass" shader distributed with OSL:

3Delight

```
surface glass
    [[ string help = "Simple dielectric material" ]]
(
    float Ks = 1,
    color Cs = 1,
    float eta = 1.5
  )
{
    Ci = Ks * reflection(N, eta) + Cs * refraction(N, eta);
}
```

Other Systems

```
surface glass
    [[ string help = "Simple dielectric material" ]]
(
    float Ks = 1,
    color Cs = 1,
    float eta = 1.5
  )
{
        float _eta = backfacing() ? 1/eta : eta;
        Ci = Ks * reflection(N, _eta) + Cs * refraction(N, _eta);
}
```

As you can see, there are no fresnel terms and no `backfacing()` call. *3Delight* will take the proper decision, based on many factors including the fresnel factors, to properly sample the surface.

Supported Closures

3Delight supports all the most advanced closures. Some of the BRDF went through extensive research in order to extend them beyond the original specs. As an example, 3Delight's GTR can also model refractions, allowing to render realistic frosted glass and other effects.

Closure	Description	Ray Types
microfacet – *ggx*	Models isotropic or anisotropic GGX BRDF. This model can handle reflection, refraction or both at the same time.	reflection, refraction, glossy

microfacet – *gtr*	Models isotropic GTR BRDF. A "gamma" parameter can be supplied to control the "tail" of the highlight to model highly realistic materials.	reflection, refraction, glossy
microfacet – *cook-torrance*	Models an anisotropic *Cook-Torrance* BRDF.	reflection, glossy
microfacet – *blinn*	Models a Blinn specular BRDF	reflection, glossy
oren_nayar	Models a diffuse reflector based on the Oren-Nayar model.	diffuse
diffuse	Models a diffuse reflector.	diffuse
reflection	Models a perfect reflector. Note that fresnel factor is automatically computed by *3Delight.* If no fresnel component is wanted, one can pass 0 as the "eta" parameter.	reflection
refraction	Models a refraction. Fresnel factor is included by 3Delight.	refraction
hair	Models a Marschner BRDF for hair. Simulates the R, TT and TRT lobe as suitable for a monte carlo simulation.	reflection, refraction, glossy.
subsurface	Starts a subsurface simulation to model a BSSRDF.	subsurface
emitter	omnidirectional emitter	--
hair	Marschner hair model. This closure works with sub-closures. Refer to The Hair Closure chapter below.	hair

3Delight Extension to Closures

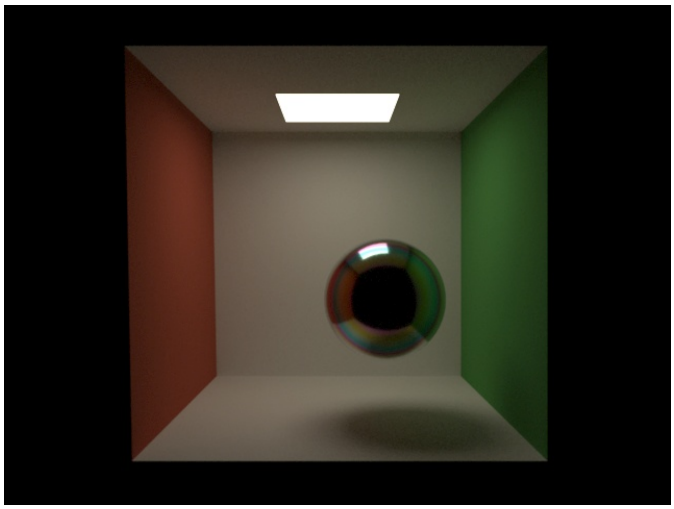
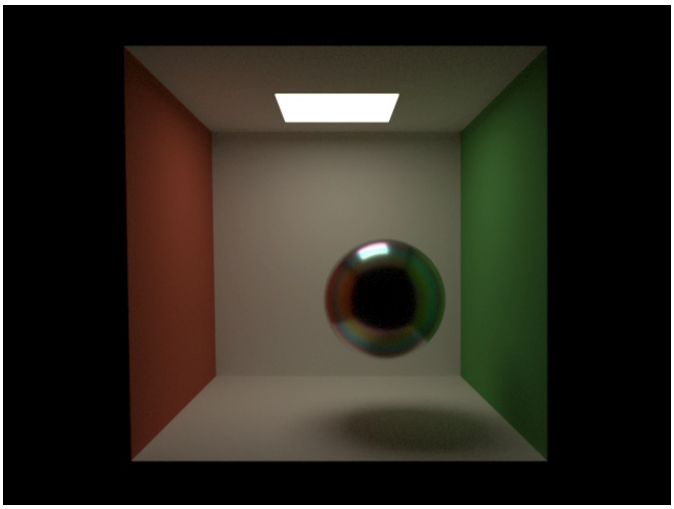
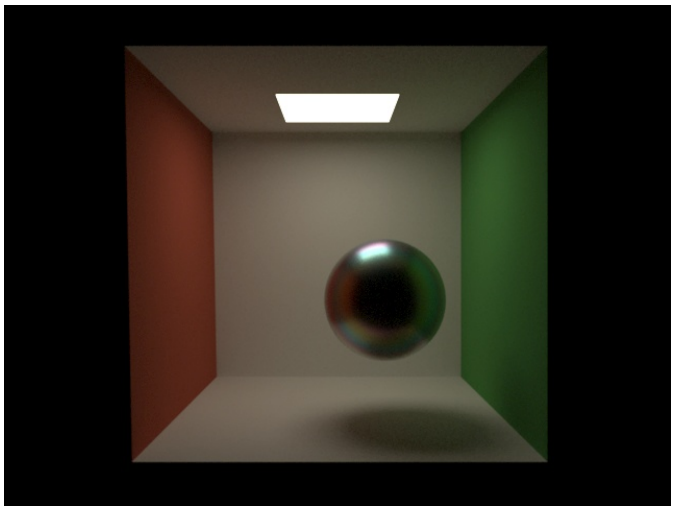
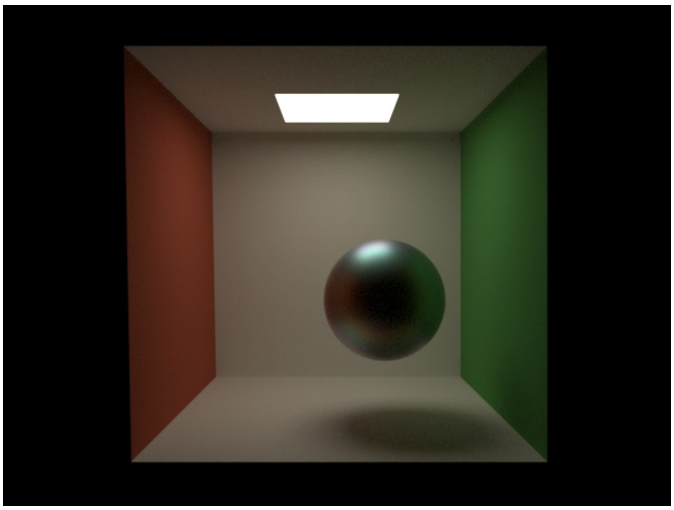
Some of the closures, for examples GGX and GTR, have been extended to render some relatively difficult effects. In particular, a lot of research have been done to render "thin film interference" on metallic surfaces. The 3Delight Metal material is a good example of usage.

The following parameters are recognised for the GGX and GTR micro-facet distributions:

Parameter	Description
`color realeta`	Real part of the index of refraction of the base layer.
`color complexeeta`	Imaginary part of the index of refraction the base layer.
Note that the pair (realeta, complexeeta) replaces the eta parameter. One can still use 'eta' to describe non-metallic surfaces.	
`float thinfilmthickness`	Thickness of the film on the surface. *As an example, on metals, this corresponds to the thickness of the oxide.*
`float thinfilmeta`	The index of refraction of the film. As an example, o*n metals, this corresponds to the index of refraction of the oxide.*
`float mediumeta`	Index of refraction of the outside medium. Defaults to 1 (vacuum) if not specified. An example scenario where the this needs to be change: a varnish
`float gamma`	Pass this to the GTR closure to control the tail of the specular highlight. Setting this value to 2 renders a GGX distribution exactly.

Here is an example render of a steel sphere with a thin film of oxide ferum. It is rendered with varying roughness.

R=0.05	R=0.1

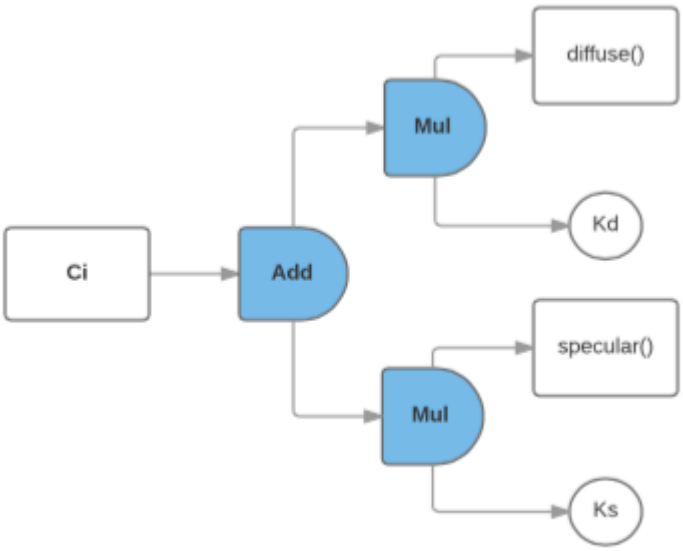
	
R=0.2	**R=0.4**
	

Arbitrary Output Variables

Any closure can be output to a special `outputvariable()` function to be output as an AOV.

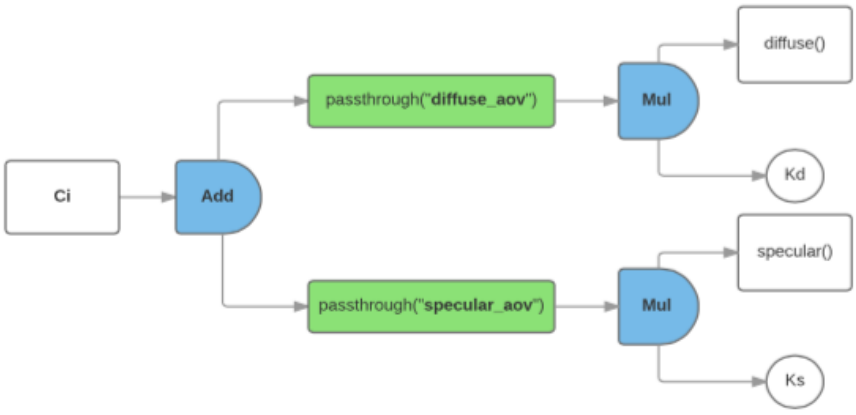
Consider the following simplified line of code and the corresponding evaluation:

```
Ci = Kd * diffuse() + Ks * specular();
```



To store the specular and diffuse components, along with weights, into the `specular_aov` and `diffuse_aov` output variables the code has to be changed in the following manner:

```
closure diffuse_aov = Kd * diffuse();
closure specular_aov = Ks * specular();
Ci  = outputvariable("diffuse_aov", diffuse_aov) +
        outputvariable("specular_aov", specular_aov);
```

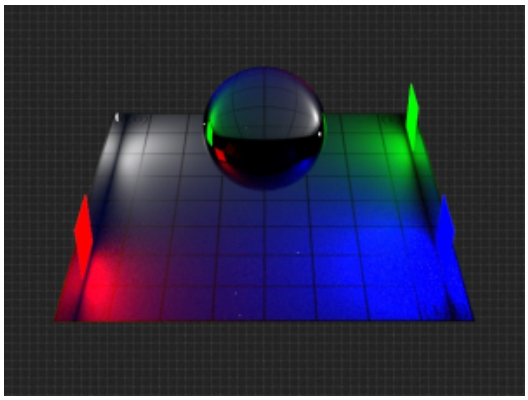
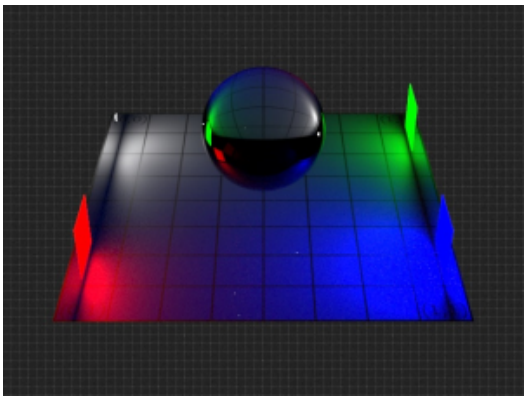
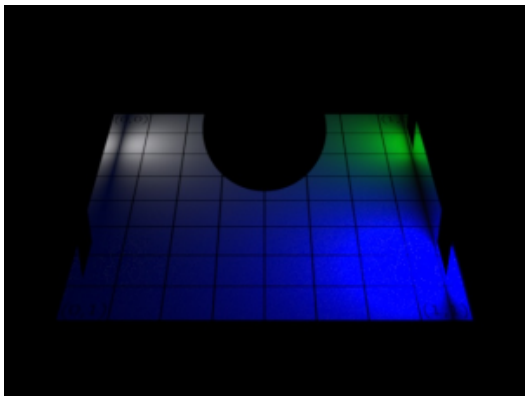
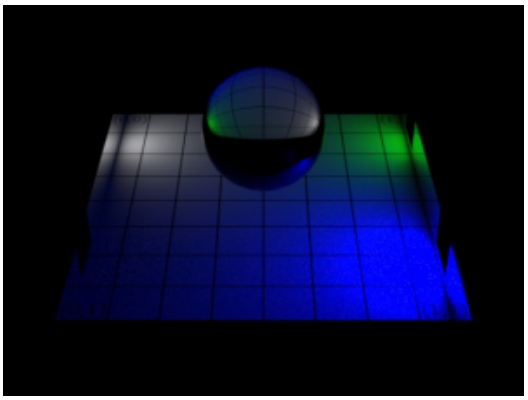
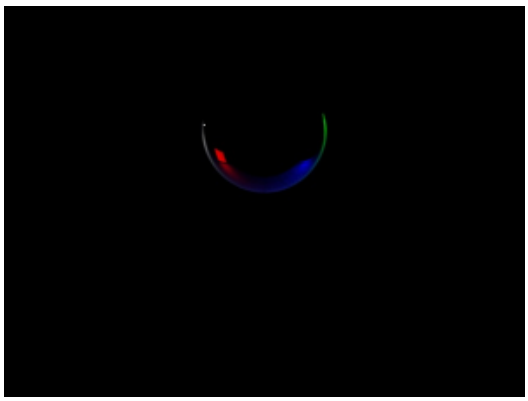
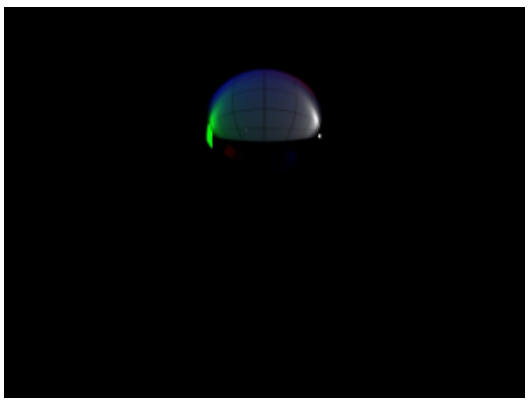


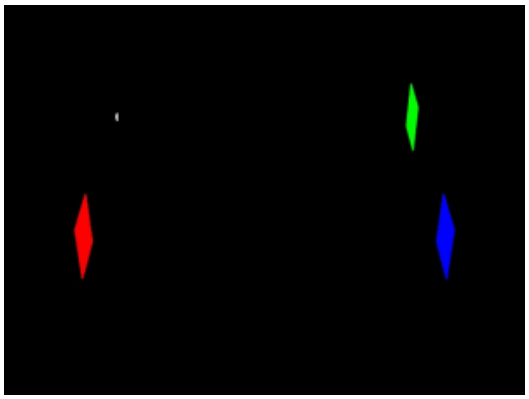
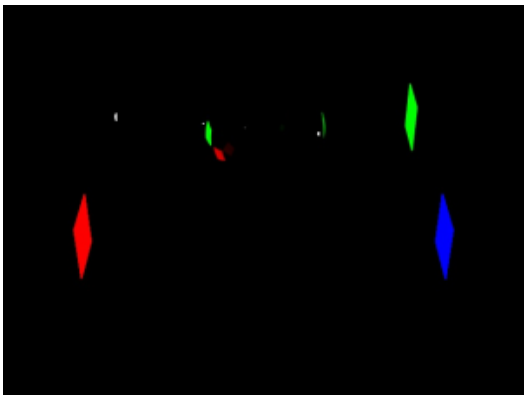
Note that AOVs can be output in any node of the OSL tree. If two different OSL nodes contribute to the specular_aov variable, *3Delight* will correctly blend their contribution together in the final output and in the AOV.

AOV Forwarding

3Delight introduces the concept of AOV forwarding to solve a classical problem with reflection, refraction and transmission AOVs. For example, a diffuse surface reflected by a mirror will usually appear in the reflection AOV. In some cases it is much more practical to see that diffuse surfaces in the diffuse AOV, leaving the reflection AOV black for that surface. This feature is not part of OSL language but is an attribute that users can set on a per-object basis. This means that there is no need to change your shaders in order to benefit from this feature.

Here is an example of some common AOVs with and without forwarding:

	Without forwarding	With forwarding
RGBA		
diffuse		
reflection		
refraction		

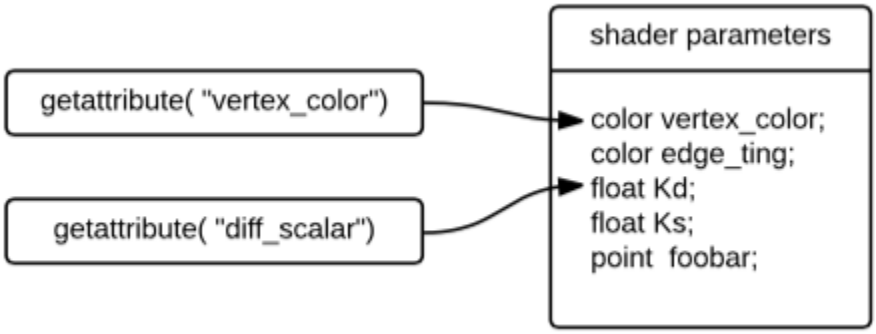
emission		

Gone is `lockgeom`

A remnant of the RenderMan way of doing things is the '`lockgeom`' parameter which hints of *symbolic linking* of data defined on the geometry to parameters defined in the shader. From our point of view, this feature is useless, as-is, in the OSL world:

- Symbolic linking is *weak* in that it is ill-defined; what happens if you have many parameters with the same name in an OSL network?
- `lockgeom` can be elegantly implemented by a ... connection (*explicit linking*).

In *3Delight*, any parameter with no incoming connection will be optimised out (folded). In order to read any primitive variable from the geometry, the OSL network must contain a reader node that calls the `getattribute()` function and output it to the input parameter. This clearly states the behaviour of the network.



In this diagram, "vertex_color" and "Kd" are connected to an OSL node which executes getattribute(). All the other variables are "folded".

A "drawback" of this method is that one can't easily get the default parameter value if the primitive variable is undefined. But bear in mind that 99% of shader networks are machine generated and this gives us two possible solutions to this problem:

1. Don't output the `getattribute()` node when the primitive variable is not present.
2. Output the proper default value in the `getattribute()` node in case the primitive variable is not present. This involves shader node interrogation.

The Hair Closure

The hair BSDF is a fairly complex function which simulates several effects observed on real hair fibers. It supports a variable number of major lobes, usually named R, TT, TRT, etc which specify different paths that a light ray can take inside a hair fiber. We choose to break down the model into sub-components with each component being one lobe of the BRDF and is specified by a closure. The general form of the hair closure looks like this:

```
hair( dPdv, eta, absorption, sub-components, optional parameters );
```

Follows a description of each parameter of the hair closure:

dPdv

This parameter gives the direction of the hair strand at the sampling point.

eta

Index of refraction of the hair strand.

absorption

Absorption of the hair strand at the sampling point

sub-components

An expression of the form: weight1 * hair_component1 + weight2 * hair_component2 + ... + weightn * hair_componentn

optional parameters

accepts "vector eccentricity" for now. It specifies the direction and the eccentricity of the cross section of one hair. This parameter should be the same for the entire hair strand for realistic results. An example code to generate such a vector would look like this:

```
vector eccentricity = eccentricity_scale * rotate( normalize(dPdu), random_hair * 2 * PI, 0, point(dPdv) );
```

Accepted sub-components call the `hair_component` closure with a parameter specifying the lobe to sample:

```
weight * hair_component( lobe, longitudinal_roughness, azimuthal_roughness, hair_scales_tilt );
```

Here is a description of each parameter of the hair_component closure:

weight

This scales the contribution of each lobe. Note that with values greater than 1, internal normalization might be done in order to avoid energy amplification by the BSDF. This means changing these weights can change the look of the hair but will generally not make it brighter overall.

lobe

Can be either "R", "T", "TRT" or "TRRT"

longitudinal_roughness

This will change the size of the lobe along the length of the hair. It behaves just like roughness for other BSDFs.

azimuthal_roughness

This will change the size of the lobe across the hair fiber. As the simulated hair is cylindrical, this parameter has little effect on the R and TT lobes. Its effect is most visible on the sharpness of the TRT lobe (glints).

hair_scales_tilt

The angle of the scales which form the surface of the hair fiber, in radians. It affects the position of the highlight on the strand. This will typically be in the range of -0.05 to -0.1 radians for human hair (negative to tilt towards the root). Note that the final position is computed from this angle differently for each lobe so using the same value for all lobes will produce distinct highlights. Note that have different scale tilts per sampling lobe does't make geometric sense (the hair fiber doesn't change for each lobes) but this could be needed for artistic control.

A very basic OSL hair shader is presented below. Note that we use the same roughness in the longitudinal and azimuthal directions.

```
#include "3delightosl.h"

surface basic_hair(
    color absorption = 1 - color( 0.99, 0.8, 0.6 ),
    float weight[3] = { 1, 1, 1 },
    float roughness[3] = { 0.03, 0.4, 0.6 },
    float highlight_position[3] = { 0.05, 0.05, 0.05 }
    )
{
    closure color hair_ci =
        hair(
            dPdv, 1.55, absorption,
            weight[0] * hair_component( "R", roughness[0], roughness[0], -highlight_position[0] ) +
            weight[1] * hair_component( "TT", roughness[1], roughness[1], -highlight_position[1] ) +
            weight[2] * hair_component( "TRT", roughness[2], roughness[2], -highlight_position[2] )
            );

    Ci = hair_ci;
}
```