Open Shading Language

Cycles Only

It is also possible to create your own nodes using Open Shading Language (OSL). These nodes will only work with the CPU and OptiX rendering backend.

To enable it, select *Open Shading Language* as the shading system in the render settings.

Note

Some OSL features are not available when using the OptiX backend. Examples include:

- Memory usage reductions offered by features like on-demand texture loading and mip-mapping are not available.
- Texture lookups require OSL to be able to determine a constant image file path for each texture call.
- Some noise functions are not available. Examples include *Cell*, *Simplex*, and *Gabor*.
- The trace function is not functional. As a result of this, the Ambient Occlusion and Bevel nodes do not work.

Script Node

OSL was designed for node-based shading, and *each* OSL shader corresponds to *one* node in a node setup. To add an OSL shader, add a script node and link it to a text data-block or an external file. Input and output sockets will be created from the shader parameters on clicking the update button in the Node or the Text editor.

OSL shaders can be linked to the node in a few different ways. With the *Internal* mode, a text data-block is used to store the OSL shader, and the OS bytecode is stored in the node itself. This is useful for distributing a blend-file with everything packed into it.

The *External* mode can be used to specify a `.osl` file from a drive, and this will then be automatically compiled into a `.oso` file in the same directo It is also possible to specify a path to a `.oso` file, which will then be used directly, with compilation done manually by the user. The third option is to specify just the module name, which will be looked up in the shader search path.

The shader search path is located in the same place as the scripts or configuration path, under:

Linux

```
$HOME/.config/blender/4.4/shaders/
```

Windows

```
C:\Users\$user\AppData\Roaming\Blender Foundation\Blender\4.4\shaders\
```

macOS

```
/Users/$USER/Library/Application Support/Blender/4.4/shaders/
```

Tip

For use in production, we suggest to use a node group to wrap shader script nodes, and link that into other blend-files. This makes it easier to make changes to the node afterwards as sockets are added or removed, without having to update the script nodes in all files.

Writing Shaders

For more details on how to write shaders, see the OSL Documentation.

Here is a simple example:

```
shader simple_material(
    color Diffuse_Color = color(0.6, 0.8, 0.6),
    float Noise_Factor = 0.5,
    output closure color BSDF = diffuse(N))
{
    color material_color = Diffuse_Color * mix(1.0, noise(P * 10.0), Noise_Factor);
    BSDF = material_color * diffuse(N);
}
```

Closures

OSL is different from, for example, RSL or GLSL, in that it does not have a light loop. There is no access to lights in the scene, and the material must be built from closures that are implemented in the renderer itself. This is more limited, but also makes it possible for the renderer to do optimizations and ensure all shaders can be importance sampled.

The available closures in Cycles correspond to the shader nodes and their sockets; for more details on what they do and the meaning of the parameters, see the shader nodes manual.

See also

Documentation on OSL's built-in closures.

BSDF

- `diffuse(N)`
- `oren_nayar(N, roughness)`
- `diffuse_ramp(N, colors[8])`
- `phong_ramp(N, exponent, colors[8])`
- `diffuse_toon(N, size, smooth)`
- `glossy_toon(N, size, smooth)`
- `translucent(N)`
- `reflection(N)`
- `refraction(N, ior)`
- `transparent()`
- `microfacet_ggx(N, roughness)`
- `microfacet_ggx_aniso(N, T, ax, ay)`
- `microfacet_ggx_refraction(N, roughness, ior)`
- `microfacet_beckmann(N, roughness)`
- `microfacet_beckmann_aniso(N, T, ax, ay)`
- `microfacet_beckmann_refraction(N, roughness, ior)`
- `ashikhmin_shirley(N, T, ax, ay)`
- `ashikhmin_velvet(N, roughness)`

Hair

- `hair_reflection(N, roughnessu, roughnessv, T, offset)`
- `hair_transmission(N, roughnessu, roughnessv, T, offset)`
- `principled_hair(N, absorption, roughness, radial_roughness, coat, offset, IOR)`

BSSRDF

Used to simulate subsurface scattering.

bssrdf(method, N, radius, albedo)

PARAMETERS:

- **method** (*string*) –

  Rendering method to simulate subsurface scattering.

  - `burley` : An approximation to physically-based volume scattering. This method is less accurate than `random_walk` however, in so situations this method will resolve noise faster.
  - `random_walk_skin` : Provides accurate results for thin and curved objects. Random Walk uses true volumetric scattering inside the mesh, which means that it works best for closed meshes. Overlapping faces and holes in the mesh can cause problems.
  - `random_walk` : Behaves similarly to `random_walk_skin` but modulates the *Radius* based on the *Color*, *Anisotropy*, and *IOR* This method thereby attempts to retain greater surface detail and color than `random_walk_skin`.
- **N** (*vector*) – Normal vector of the surface point being shaded.
- **radius** (*vector*) – Average distance that light scatters below the surface. Higher radius gives a softer appearance, as light bleeds into shadows and through the object. The scattering distance is specified separately for the RGB channels, to render materials such as skin where red light scatters deeper. The X, Y and Z values are mapped to the R, G and B values, respectively.
- **albedo** (*color*) – Color of the surface, or physically speaking, the probability that light is reflected for each wavelength.

Volume

- `henyey_greenstein(g)`
- `absorption()`

Other

- `emission()`
- `ambient_occlusion()`
- `holdout()`
- `background()`

Attributes

Geometry attributes can be read through the `getattribute()` function. This includes UV maps, color attributes and any attributes output from geometry nodes.

The following built-in attributes are available through `getattribute()` as well.

`geom:generated`
: Automatically generated texture coordinates, from non-deformed mesh.

`geom:uv`
: Default render UV map.

`geom:tangent`
: Default tangent vector along surface, in object space.

`geom:undisplaced`
: Position before displacement, in object space.

`geom:dupli_generated`
: For instances, generated coordinate from instancer object.

`geom:dupli_uv`
: For instances, UV coordinate from instancer object.

`geom:trianglevertices`
: Three vertex coordinates of the triangle.

`geom:numpolyvertices`
: Number of vertices in the polygon (always returns three currently).

`geom:polyvertices`

geom:polyvertices

Vertex coordinates array of the polygon (always three vertices currently).

geom:name

Name of the object.

geom:is_smooth

Is mesh face smooth or flat shaded.

geom:is_curve

Is object a curve or not.

geom:curve_intercept

0..1 coordinate for point along the curve, from root to tip.

geom:curve_thickness

Thickness of the curve in object space.

geom:curve_length

Length of the curve in object space.

geom:curve_tangent_normal

Tangent Normal of the strand.

geom:is_point

Is point in a point cloud or not.

geom:point_radius

Radius of point in point cloud.

geom:point_position

Center position of point in point cloud.

geom:point_random

Random number, different for every point in point cloud.

path:ray_length

Ray distance since last hit.

object:random

Random number, different for every object instance.

object:index

Object unique instance index.

object:location

Object location.

material:index

Material unique index number.

particle:index

Particle unique instance number.

particle:age

Particle age in frames.

particle:lifetime

Total lifespan of particle in frames.

particle:location

Location of the particle.

particle:size

Size of the particle.

`particle:velocity`
: Velocity of the particle.

`particle:angular_velocity`
: Angular velocity of the particle.

Trace

CPU Only

We support the `trace(point pos, vector dir, ...)` function, to trace rays from the OSL shader. The "shade" parameter is not supported currently, but attributes can be retrieved from the object that was hit using the `getmessage("trace", ..)` function. See the OSL specification for details on how to use this.

This function cannot be used instead of lighting; the main purpose is to allow shaders to "probe" nearby geometry, for example to apply a projected textu that can be blocked by geometry, apply more "wear" to exposed geometry, or make other ambient occlusion-like effects.

Metadata

Metadata on parameters controls their display in the user interface. The following metadata is supported:

`[[ string label = "My Label" ]]`
: Name of parameter in in the user interface

`[[ string widget = "null" ]]`
: Hide parameter in the user interface.

`[[ string widget = "boolean" ]]` and `[[ string widget = "checkbox" ]]`
: Display integer parameter as a boolean checkbox.