Michel Anders

Open Shading Language for Blender
A practical primer

Open Shading Language for Blender

By Michel Anders

Blender Market Edition

Table of Contents

Intro

Who should read this book?

Anyone who wants to extend his or her skill set in designing materials. Blenders node system is already pretty powerful but if you want to have total control over the materials you design for Blenders Cycles renderer or if you want to have some specific nodes to ease your work flow, learning Open Shading Language is a must. Also many shaders written for the Renderman (tm) Shading Language are fairly easy to port to OSL.

What kind of topics will be covered?

This book focuses on writing practical shaders which means you can download the code and use it as is or, even better, learn from it by adapting it to your needs. Often even details in the code are explained to lower the learning curve but on the other hand some specialist topics not specific to OSL are not covered (an example is regular expressions). Each shader that we develop is covered in its own section and specific areas of interest are listed at the beginning of each section.

Is Open Shading Language restricted to Blender?

In principle no: Currently V-Ray supports OSL and Autodesk Beast will also support OSL and maybe other paid 3D applications will support it once it gains momentum, but at this moment Blender is the only open source application that has adapted OSL. This might well change in the future as OSL is well documented and can be readily adapted to other renderers.

What skills do I need to benefit from this book?

You should be be fairly comfortable in using Blender. You need not be a an experienced artist but you should know your way around and know how to create materials for the Cycles renderer with Blender's node system. You should also know how to program. Again you needn't be a seasoned software developer but it certainly helps if you know how to program short pieces of code in any procedural language like C, C++, Java, Python or even (visual) Basic. Open Shading Language is a C-like language and quite easy to learn. It doesn't rely on concepts like object orientation nor does it depend on hard to master things like pointers and its data types are limited but adequate for the job. In short, logical thinking will get you a long way and the introductory chapters will introduce the most important ideas. If you can program a short program that lists the first ten even integers in the programming language of your choice OSL will pose no problem.

Besides programming, writing shaders involves some math and geometry as well but nothing above high school math. We will explain each math issue we encounter and will concentrate on *how* to use things without delving into the small details. Where appropriate links to relevant topics on the web will be provided.

Where can I find more about Open Shading Language?

The single most important site is the GitHub repository of OSL. Not because you need to get the source code but foremost because it is the repository of the OSL Language Specification. Once you mastered learning Open Shading Language for Blender you will find the Language specification a valuable reference.

In the appendices I list some additional sites which are worth visiting to learn more about OSL.

What are OSL shaders good for?

When should you consider programming a shader in OSL? You might do it for fun of course but even then you need to balance the costs versus the profits. After all it takes time to learn a new language and implementing and maintaining shaders takes some effort too. So when is there really a need to implement a shader in OSL? The following criteria may help you decide:

A shader would be impossible or very hard to create in Blender's node system

This is the most clear cut criterion. Blender's node system is immensely powerful but still some things are impossible. For example OSL provides noise types like Gabor noise that you might want to use but that Blender's node system does not provide (yet). Regular patterns are also quite difficult or impossible to generate with nodes because the number of regular texture inputs is limited to wave, gradient, checker and brick. Even if possible a node system might need an enormous amount of nodes. Likewise random distributions of regular elements like ripples on a pond caused by raindrops are hard. Basically anything that is naturally expressed as a repetition (a programmed loop) or has many decisions to make (many if/else statements) is generally easier to program than to construct from nodes.

A shader based on nodes would take to much memory

If your shader would use image textures you would need large ones to be able to zoom in close and get good results even if the shaded part covers only a few pixels. A procedural shader like one implemented in OSL would take the same amount of memory regardless whether you need high or low resolution. Of course it's often easier to use image textures but if size matters you might want to create such textures programmatically. Being able to produce a texture at any level of detail might also prevent aliasing effects and additionally programmed shaders often don't depend on uv-unwrapping. Not depending on uv-maps saves time because you don't have to create one but also might get rid of seams. (when uv-unwrapping you have to make sure that seams are invisible or line up with the edges of a tileable image.)

A shader based on nodes would be too slow

Blender's node system isn't slow at all but if you need very many nodes to implement your shader, a solution that is bundled in a single OSL shader might be quite a bit faster. On the other hand, OSL shaders are currently limited to execution on the CPU and therefore cannot benefit from the potentially must faster GPU.

So the real answer to the question of when to use an OSL shader is, as with most non-trivial questions: "it depends". It is possibly easiest to cobble together a first approximation of the shader you have in mind from nodes. If in doing so the number of nodes needed increases quickly or you encounter things that cannot be done at all, consider writing an OSL node.

Reading hints

Typesetting a book containing a fair amount of source code is fiendishly difficult, especially because you have to allow for the very small width of some reading devices. Therefore the code is formatted to fit very short line widths but that might at times not be very comfortable to read, in which case I recommend looking at the original source code (see below) which is generally not formatted.

It might also be a good idea to turn off overriding the stylesheets associated with this e-book. Many e-readers (notably Aldiko) do this by default, which is fine for non-fiction but for books with lots of illustrations this doesn't look good. Disabling this override let you enjoy the book better. In most readers it is still possible to separately set the size of the text.

Code availability

The source code shown in the book is freely available on GitHub:
https://github.com/varkenvarken/osl-shaders
The source code is licensed as open source under a GPLv3 license.

Note: all example shaders in this book are present as an .osl file in the Shaders directory but for each shader file there is also an accompanying .blend file with the same name that demonstrates its use. Simply by opening this file and selecting 'Rendered' as display mode in the 3d view will showcase the shader. (When opening the file Blender sets the display mode to solid so it is necessary to reset this to rendered to see the effect.)

A very simple shader

Focus areas

- how to prepare Blender to use Open Shading Language
- creating your first OSL shader
- trouble shooting

Preparing Blender

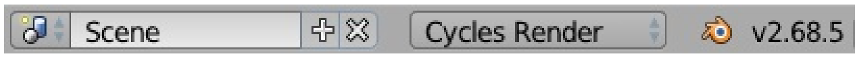
Because most errors that may occur when working with OSL will appear in the console, it is a good idea to make sure the console is visible. For unix-like systems like Linux or OSX this is done by starting Blender from a command line (for example an xterm). For Windows a console created after Blender has started by selecting `Window -> Toggle console`.

Do not confuse this console with Blenders built-in Python console, that is a completely different and unrelated entity.

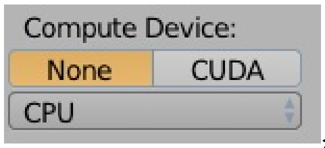
OSL is currently only available for Cycles (not for Blenders internal renderer) so we have to check if all prerequisites are met.

- Open a new Blender file with just a single object, a lamp and a camera.

  If you haven't changed your startup preferences the default startup will do or alternatively you may download and open `blank.blend` from the Shaders directory in the code accompanying this book,

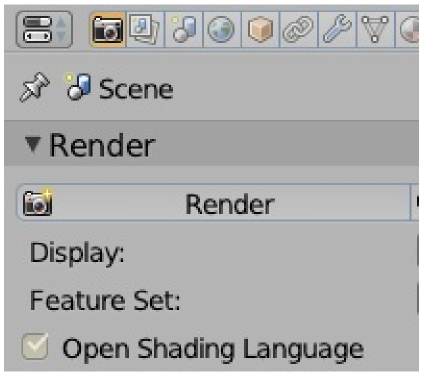
- verify that you have selected Cycles as your renderer as shown in the screen shot
  

- verify that you will use the CPU to render

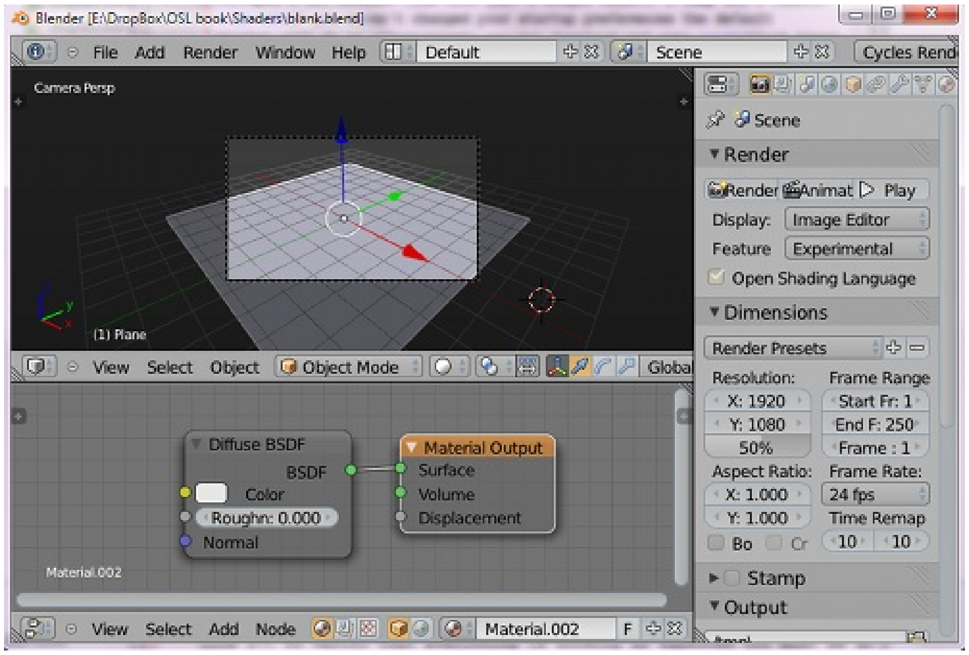
  (not the GPU as OSL is currently only available on the CPU)
  `File -> User Preferences -> System`
  
  Compute Device:
  None CUDA
  CPU
  ,

- verify that OSL is enabled

  Check the Open Shading Language box in the properties.



To verify that everything is working as expected you must be able to work with the node editor. A suggested layout is shown below (that is the layout that blank.blend has)


Blender [E:\DropBox\OSL book\Shaders\blank.blend]
File Add Render Window Help Default Scene Cycles Rend
Camera Persp
(1) Plane
View Select Object Object Mode Globa
Diffuse BSDF
BSDF
Color
Roughn: 0.000
Normal
Material Output
Surface
Volume
Displacement
Material.002
View Select Add Node Material.002 F
Scene
Render
Render Animat Play
Display: Image Editor
Feature Experimental
Open Shading Language
Dimensions
Render Presets
Resolution: X: 1920 Y: 1080 50%
Frame Range Start Fr: 1 End F: 250 Frame : 1
Aspect Ratio: X: 1.000 Y: 1.000
Frame Rate: 24 fps
Time Remap 10 10
Bo Cr
Stamp
Output

`blank.blend` has a default node based material already assigned to its plane but if the object in your startup file has no material you can simply assign a new one by selecting the object and clicking `New` in the node editor.

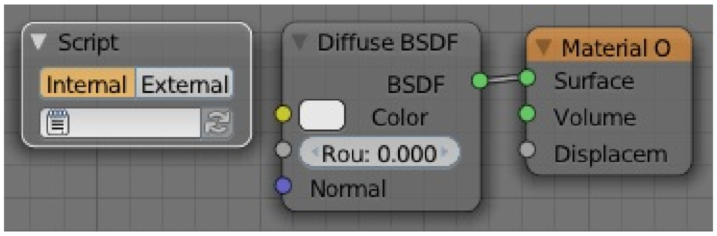
Creating your first OSL shader

If you have written a shader the following steps are needed to use it as part of a node based material:

- add a new script node,
- point it to the code you have written,
- connect the sockets to other nodes.

Lets have a closer look at each of these steps.

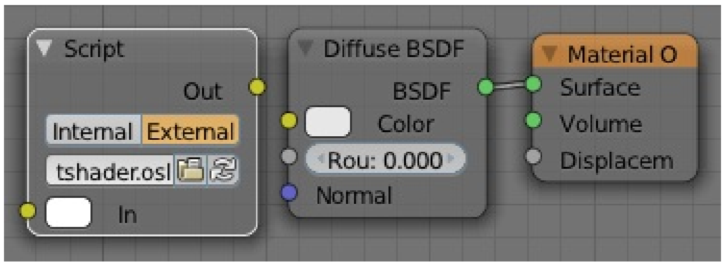
To use a shader as a new node in Blenders Cycles renderer we must add a Script node to a node based material. Click `Add->Script->Script` in the node editor menu and a new node is added to your material. It doesn't have any input or output sockets yet but it does let you choose either an external script (= OSL code that resides in a text file on disk) or an internal script (= OSL code present in one of Blenders text editor buffers)



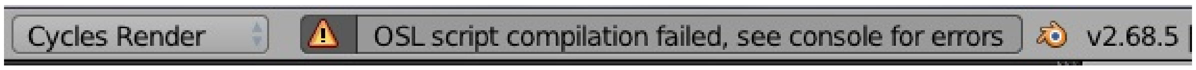
Each method is fine but both have their pros and cons. An external file is easier to distribute on its own and may be maintained as part of a separate collection of shaders while a shader contained in the text editor is a part of your .blend file and will be saved with the rest of your scene. Blenders text editor is sufficient for small shaders but for larger shaders using an external editor is often preferable and if you download shaders from the Shaders directory in the code accompanying this book you can select External and use the file browser that opens to point to the downloaded file.

Select `External` and click on the text field to open a file browser, choose `Shaders/firstshader.osl` and confirm your choice or alternatively, select `Internal`, copy the code shown below to the internal text editor and click on the text field to select your internal text file.

When you select either an internal text buffer or an external file, the shader is compiled automatically and if everything went well some input and output sockets will appear.



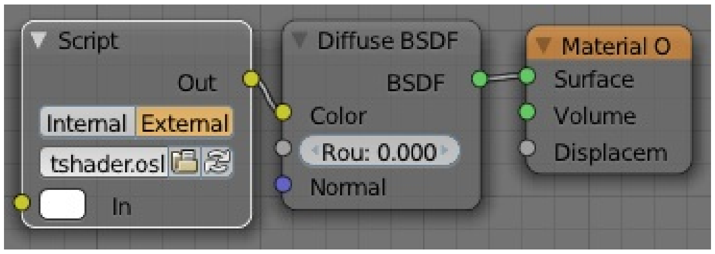
If something went wrong while compiling the shader a short message is shown in the top menu bar



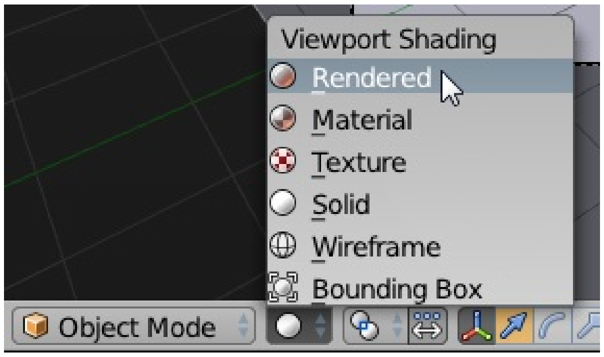
OSL script compilation failed, see console for errors

and possibly additional information is shown on the console. If that happens (and if you write a shader yourself instead of using a downloaded sample shader this is very likely to happen because writing a new non-trivial shader without any syntax errors in one go is highly improbable), fix the errors and click the recompile button (the button on the right side of a script node next to the text field.

`firstshader.osl` is a shader with just one input socket, a color (and hence shown with a yellow dot) and one output socket, also a color. You may connect its output socket to the color input of the diffuse shader of your material as shown in the screen shot.



If you now select 'Rendered' in the 3d view menu



you see that the shader has taken the default bright white input color and darkened it significantly. Lets have a look a the code for firstshader.osl and see how this has come about:

```
01.  shader darken(
02.    color In = 1,
03.    output color Out = 1
04.  ){
05.    Out = In * 0.5;
06.  }
```

Note that the line numbers preceding the code are for reference only and not part of the shader code.

The first line declares a generic shader called `darken` (there are more types of shader in OSL but not all are currently supported by Blender. Most shaders in this book are defined as generic shaders with the `shader` keyword. When we encounter different shaders later on their differences will be explained).

A shader looks like a function with parameters and the second line defines a input parameter called `In`. The type of this parameter is `color` (one of the basic types of OSL) and its default value is 1, meaning that all three components of the color (red, green and blue) will be set to one making the default color bright white.

The next line defines a parameter `Out`, which is also a color with the default set to 1, i.e. white. It is marked as an output parameter by the `output` keyword. When the shader is compiled, Blender adds sockets to the node for each parameter. Input parameters will appear as sockets on the left side of the node and output parameters will appear on the right. Blender will color these sockets based on their types, with yellow for color parameters.

All parameters *must* be defined with a default. The default of an input parameter is used if no other nodes are connected to this socket. If an input socket is not connected its value can be changed by the end user by clicking on it. Depending on the type of socket a suitable editor will pop up, for example a color picker if it is a color. A value of an unconnected input socket can even be animated (by inserting a key frame by hovering over it with the mouse and pressing `I`), just like almost everything else in Blender.

The default value of an output socket is used if nowhere in the body of the shader a value is assigned to it.

The body of this shader consist of a single line (line 5) that multiplies the `In` value by `0.5` and assigns it to the `Out` parameter. Multiplying a color by a single value will multiply each component by that value so in this example we darken all color components by the same amount.

This example shader is of course about as simple as a shader can be and maybe not all that useful but it does show how Blender takes care of much of the work of making a new node usable by adding sockets with appropriate editors to the new node based on the types of the input and output parameters.

Troubleshooting

When you use script nodes a few things may go wrong beside your script having syntax errors. Most configuration errors won't even show up as errors on the console and often all you see then is that your material appears black when rendered. The most common ones are:

- you cannot add a script node

  when you click add in the node editor and you don't have a choice listed called 'Script' you probably forgot to select Cycles as your render engine because the internal renderer doesn't support script nodes. (In Blender versions before 2.68 there was a Script entry for the internal editor but clicking it had no effect)

- compilation goes well but the material shows up as a uniform color

  you forgot to check the OSL button in the render options

- there is no OSL button in the render options

  this option is only available if you use Cycles as your render engine and select CPU rendering

Data types

Focus areas

- Simple data types (numbers, strings, vectors, colors)
- Compound data types (arrays)
- User defined data types (structs)

OSL has several data types that may be used to represent numerical values and strings but also things like points, vector or collections of these values. This section gives a brief overview of the most important issues to get you started quickly. This overview is by no means complete: for details you best refer to chapter 5 of the OSL language specification.

Simple data types

int

an int represents positive and negative integer values. All integer operations found in most programming languages are supported (including modulo % and the bitwise operators and &, or | and xor ^). Integers are at least 32 bits wide (i.e. may hold values about ± 2000,000,000 but depending on your platform much larger 64 bit wide integers may be implemented. Don't depend on this if you want to write portable shaders!). An int is also used as a boolean value where 0 equals false and any other value true (OSL doesn't have an explicit boolean type. In fact all simple types may be used as boolean values, each with its own rules for which values are considered true. In this book we stick to int values for booleans. Refer to the OSL language specification for details).

example:

```
int a = 4;
int b = a * 5;
```

float

a float represents positive and negative fractional values. It can at least represent values as small 1e-35 or as big as 1e35 with 7 decimals of precision (to be precise it is at least a 32 bit float conforming to the IEEE specification. On some platforms this may be a 64 bit float with more precision and a wider range but don't depend on it if portability is important). Anywhere a float is used an int maybe used as well; it will be automatically converted to a float. All the usual operations are supported for floats and there is whole host of standard functions available as well, including trigonometric functions, exponentiation, etc.

example:

```
float a = -1.4;
float b = sin(a*1e-3);
```

string

a string is a sequence of characters. Its main use is in specifying filenames of textures. The operations are chiefly limited to comparisons but a fair number of standard functions are provided as well, including support for regular expressions.

example:

```
string a = "filename.png";
```

void

Void is not a value per se but used to explicitly signify the absence of a value, for example as with a function that has only side effects but doesn't return anything.

point like data types

point like data types have in common that they all consist of three floating point values and that they all share a common set of operations like multiplication and addition. They differ in subtle details under some conditions but for now we can treat them as equal but with a name that relates to their purpose. The components of a point like data type can be accessed with an index that starts a `0` for the first component. Operations between a point like type and a float will perform the operation on all three components individually. This includes assignment: in the code below we assign the value `7` to all components of the variable `b`.

example:

```
point a = point(1,0,0);
point b = point(0,1,0);
vector v = a - b;
normal n = cross(a,b);
a[0] = 3;
b = 7;
b = b + v * 5;
```

point

a `point` is used to represent a location or position in the world, such as the location of the object being shaded. You can add to points to translate them, multiply to scale them or call the built-in function `rotate()` to rotate them around an axis

vector

a `vector` represents a direction or a line segment between points. It is possible to translate, scale and rotate them in the same way as points and OSL provides a number of functions to perform common vector operations including calculation its length, normalizing it and calculating the cross and dot product of two vectors to name a few.

These functions are just as happy to accept points or normals but most often we think of these functions in the context of vectors. Many shaders in this book use vectors and some basic understanding of them is necessary but in most cases their use is explained in detail. Especially the first few shaders that we will encounter in the the next chapter introduce not only how to use OSL as a language but will implement a few shaders that perform some basic vector operations just to get familiar.

normal

a `normal` is a vector that denotes a direction perpendicular to something, often a face of a mesh. Anything you can do with a point or a vector you can do with a normal as well but because the direction perpendicular to a surface plays such an important role in designing shaders its useful to have a separate name for such a vector. (When we encounter closures (= light scattering functions) we will see that these closures define which fraction of the incoming light is reflected relative to the normal. For example when light is reflected from a perfect mirror the angle of the reflected light relative to the surface normal is the same as the angle of the angle of the incoming light relative to the surface normal)

color

a `color` is not point like in the sense that it has a geometrical significance but it does have three components (here representing the red, blue and green components of a color) and shares operations with the geometrical types like indexing, addition, etc. In many cases none of the components will have a value greater than one but this is not mandatory.

matrix

A `matrix` is not point like at all but a collection of 4 x 4 floats that is mainly used to transform point like data types. Because OSL provides a lot of standard functions for point like types such as rotation and transformations between different vector spaces, we will not use the matrix type in this book.

aggregate types

If we want to manipulate more than one simple value as one item we need some way to aggregate them. For this purpose OSL offers arrays and structs.

array

An array is a list of items of the same type. Each of those can be accessed by using an index in the same way as components of point like types may be accessed. The length of an array is fixed once it is declared.

example:

```
float a[4];
float b[3] = { 3, 4, 5 };
point c[5];
a[2]=b[1]*8;
int alength = arraylength(a);
```

Note that the first element of an array has an index of `0`. In the first line we define an array `a` with four elements. The `b` array is defined with three elements and they are each initialized to a value. As the `c` array shows, arrays are not limited to float values but may be point-like types as well.

struct

A `struct` is a collection of items of different types. Each of those can be accessed by name. A struct is somewhat like a new type: Once you have defined a struct you can use it to declare variables.

example:

```
struct ray{
 point start;
 vector direction;
};

ray view = { point(0,0,0), vector(0,0,1) };
view.point[2] = 5;
```

Control structures

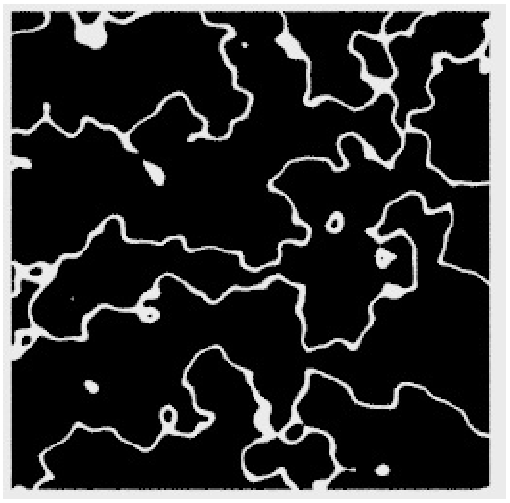
Focus areas

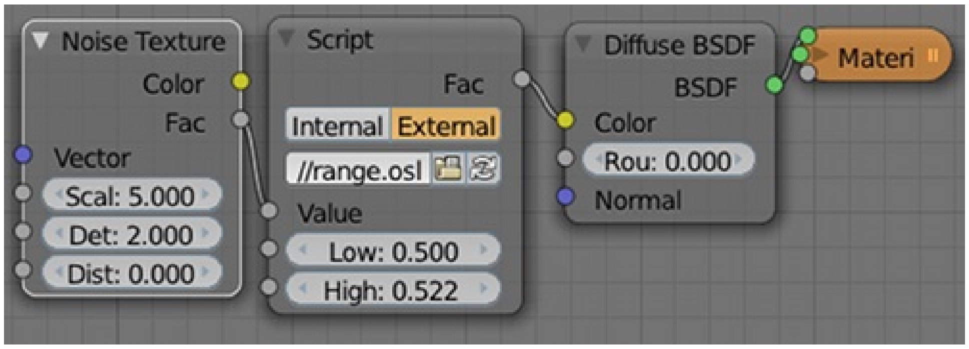
- making decisions, if/else statements
- repeating activities, for and while loops

Like any programming language worth its salt, OSL has its share of control structures to make decisions and repeat actions. These control structures are almost identical to those found in C or C++ and are documented in detail in the language specification. In this section we give a short overview in the context of some simple shaders to get a feeling for what is possible.

Left or right, you decide

Making decisions and generating different output based on some condition is present in almost every shader but the most trivial. In our example shader we want to produce an output value of 1 if an input value lies between to given limits. Such a shader node might be used for example to produce sharp edged areas from a smooth noise input. An example is shown in the image together with a node setup.





Scal: 5.000
Det: 2.000
Dist: 0.000
//range.osl
Low: 0.500
High: 0.522
Rou: 0.000

The code that implements this shader is shown below:

The code is available in Shaders/range.osl

```
shader range(
  float Value = 0,
  float Low = 0,
  float High = 0,

  output float Fac = 0
07.  ){
08.    if( Value >= Low && Value <= High ){
09.      Fac = 1;
10.    }
11.  }
```

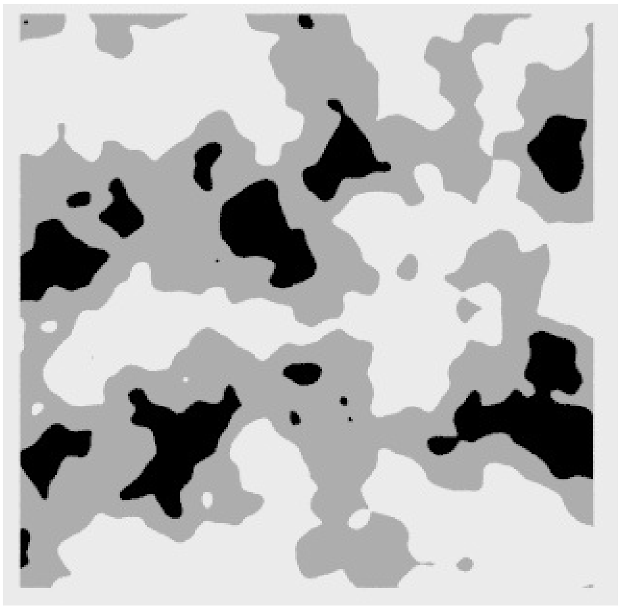
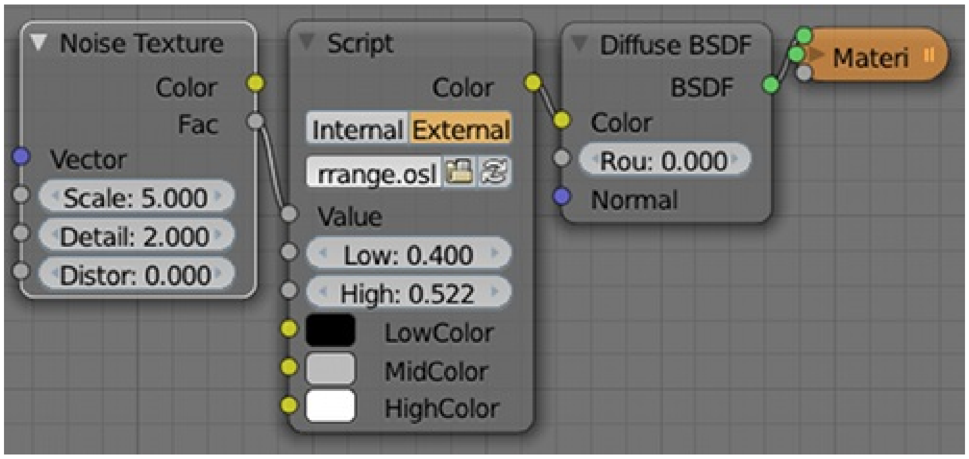
The shader has three input parameters: the `Value` we want to test and the `Low` and `High` values that specify the range that will return a value of `1`. The result, either `0` or `1`, is returned in the single output parameter `Fac`.

The body of the code consists of a single if statement (line 8). An if statement consists of an expression between parentheses and body. The body consists of a single statement or a group of statements between curly braces and is only executed if the expression of the if statement evaluates to non-zero. Because if statements may be nested (the body may contain other if statements) it might become unclear what the structure of the code is so in this book we always use the curly braces, even if there is just a single statement in the body to clearly indicate which piece of code belongs to which if statement.

The expression of an if statement may be arbitrarily complex. Here we check if the value is larger or equal to the lower limit and combine this with a check to see if the value is less than or equal to the upper limit. The && operator (logical and) indicates that both conditions should be true. A list of all operators that can be used can be found in the OSL language specification, chapter 6. If expressions are complex or the precedence of operators makes it necessary, parentheses may be used to group parts of the expression. In this case we assign the value 1 to the output parameter `Fac` if the condition is true.

In the previous example we assigned a default value to an output parameter and changed this value if a condition was met. Sometimes however we want to take different actions if the condition is not met. For this purpose the if statement has an optional else part.

In the shader presented below we want to produce three different output colors, depending on an input value being below, inside or above a given range.



Noise Texture
Color
Fac
Vector
Scale: 5.000
Detail: 2.000
Distor: 0.000

Script
Color
Internal External
rrange.osl
Value
Low: 0.400
High: 0.522
LowColor
MidColor
HighColor

Diffuse BSDF
BSDF
Color
Rou: 0.000
Normal

The code for the shader is given below and compared to the range shader this shader has three extra input parameters, the colors we want to choose from. The single output parameter is the chosen color.

Code available in Shaders/colorrange.osl

```
01.  shader colorrange(
02.    float Value = 0,
03.    float Low = 0,
04.    float High = 0,
05.    color LowColor = 0,
06.    color MidColor = 0.5, // grey
07.    color HighColor = 1,
08.
09.    output color Color = 0
10.  ){
11.    Color = LowColor;
12.    if( Value >= Low && Value <= High ){
13.      Color = MidColor;
14.    } else {
15.      if( Value > High ){
16.        Color = HighColor;
17.      }
18.    }
19.  }
```

The first line assigns `LowColor` to the output parameter as a default. The if statement again checks whether the input value is in the specified range but if this not the case the else part (line 14) is executed. This else part contains again an if statement that checks if the input value is above the upper value if the range and if so, assigns `HighColor` to the output parameter. Note that in this example again none of the curly braces in the if statements were needed but we used them anyway together with consistent indentation to avoid confusion.

I fear I might be repeating myself

Making decisions is a fundamental feature of a programming language and so is the ability to repeat an action a given number of times. To this end OSL offers the for statement which is used in the next shader to produce a given number of stripes.

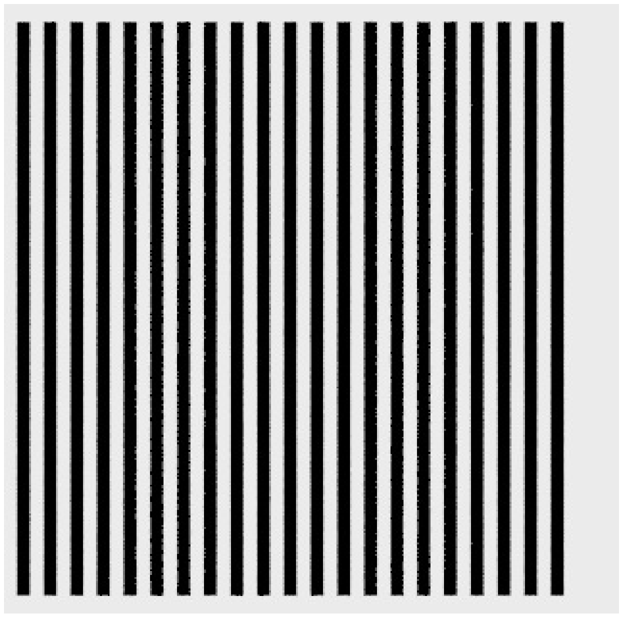
Code available in Shaders/stripes.osl

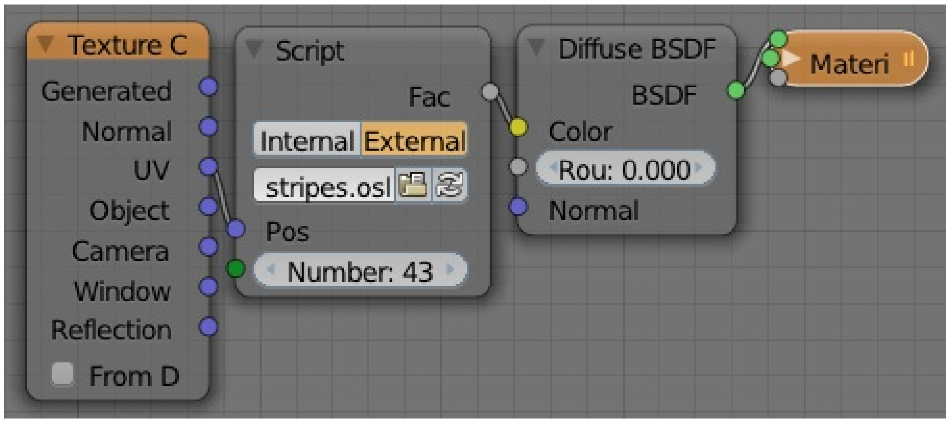
```
shader stripes(
  point Pos = P,
  int Number = 4,

  output float Fac = 0
){
  float x = mod(Pos[0],1);

  int i;
  for(i=1; i <= Number; i++){
    if( x < (float)i/Number ){
      Fac = i % 2;
      break;
    }
  }
}
```

The input parameters of the stripes shader are the position being shaded (Pos) and the number of stripes we want to produce (Number). The single output parameter Fac will be either 0 or 1 for alternating stripes.





In the body of the shader we make sure that the value of the x coordinate is in the range [0,1] by calculating the x component of the position being shaded modulo one. This guarantees that for larger values of x the pattern that we generate for the [0,1] range is repeated.

The for statement repeats the statements in its body as often as specified by the three semi-colon separated expressions between parentheses (line 10). As with the body of the if statement the curly braces are optional if the body consists of a single statement but in this book we always use them.

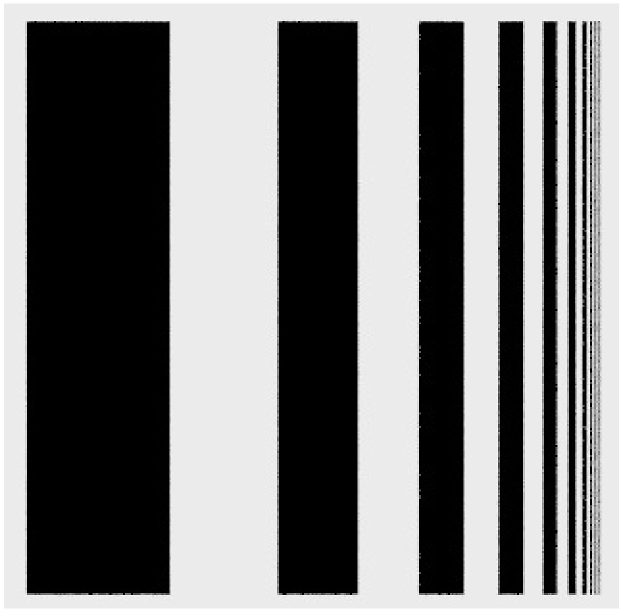
The first expression following the for keyword initializes the control variable `i`. The second expression is a condition and as long as this condition is true the body of the for statement is executed. The third expression is evaluated after each execution of the body and is used to increment the control variable by 1. (The expressions in a for statement are not limited to ones affecting a control variable. Details can be found in the language specification)

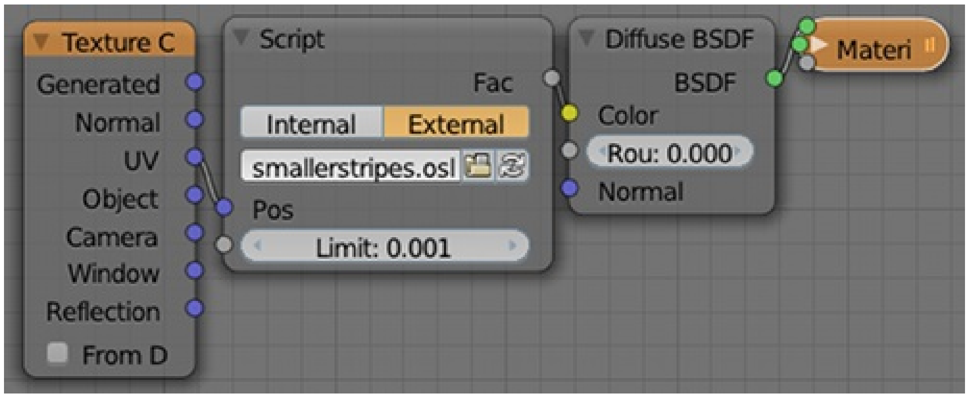
The body of the for statement itself consists of a single if statement. The expression in this if statement checks if the x coordinate is smaller than the upper limit of the stripe we are currently checking. For example, if `Number` is `4` (we want four stripes) the first time we check this expression is `x < 1/4` (`i` starts counting from one because that is the value we gave it in the initialization expression of the for statement). The second time around this expression is `x < 2/4` (because after the first execution of the body of the for statement the control variable `i` was incremented by 1). Note that we had to cast the `i` variable to a float to make sure that the division resulted in a fraction. Had we not done this we would have been dividing integers and we would have got the value `0` instead of a fraction.

If the expression evaluates to true, we assign the output variable `Fac` the value of the control variable `i` modulo 2. This will alternate between `0` an `1` depending on whether `i` is even or odd. If the expression is true we not only assign a value to the output parameter but stop executing the for statement altogether. This is done with the break statement. If we would execute the remaining iterations of for loop the test would evaluate to true again (if x is smaller than 1/4 it is also smaller than 2/4, 3/4, etc.) so we need to break out of the loop to prevent assigning the wrong value to the output parameter.

We are not always in the position to determine beforehand how many repetitions are needed. In those cases it might be better to use a while loop, which will execute its body as long as an expression is true.

In this example shader we do not want to produce a fixed number of stripes but strips that get progressively smaller until they are deemed small enough. This lower limit on the width of the strips is an input parameter.





Internal | External
smallerstripes.osl
Limit: 0.001
Rou: 0.000

The central idea in the implementation presented here is to check if the x-coordinate is below a certain limit and if not, extend this limit with a value dx and check again. On each successive iteration this value dx is diminished by a quarter until it is smaller than input parameter `Limit`.

Code available in Shaders/smallerstripes.osl

```
shader smallerstripes(
  point Pos = P,
  float Limit = 0.01,

  output float Fac = 0
){
  float x = mod(Pos[0],1)*2;
  float dx = 0.5;
  float xlimit = dx;

  float ActualLimit =
12.      Limit>0.001 ? Limit : 0.01;
13.    while( dx >= ActualLimit ){
14.      if( x < xlimit ){
15.        break;
16.      }
17.      Fac = abs(Fac-1);
18.      dx *= 0.75;
19.      xlimit += dx;
20.    }
21.  }
```

To prevent an endless loop we check if Limit is larger than some small value and if so, assign it to the variable `ActualLimit`. If not we take 0.001 as a safe minimum. There are two things worth noting here: We need this extra variable `ActualLimit` because an input parameter like Limit is read only, we cannot change its value. Also you might wonder what the strange expression on the right hand side of the assignment does (line 12). This is in fact just an if statement disguised as an expression, an idiom directly copied from the C language. The value of the expression `a ? b : c` is equal to b if a is true (non zero) and c if a is false.

The while statement consists of an expression between parentheses and a body (line 13). The body is executed as long as the expression is true. The body might consist of a single statement or several statements enclosed in curly braces. For clarity we will always use curly braces, just as we do for if and for statements.

The body of the while loop contains an if statement that checks if the x-coordinate is below the current limit. If so, we break out of the while loop. Since there are no statements following the while loop this means that the output parameter Fac will hold whatever value it is holding at that instant. Therefore, if the x-coordinate was not below the current limit we must not only extend the limit against which we will check in the next iteration but also flip the value of `Fac`. `Fac` is a floating point value and the expression `abs(Fac-1)` is one way to convert a value from 0 to 1 and vice versa.

The final two lines of the body shorten the width dx of the stripe by 25% and add this width to the `xlimit` variable.