Michel Anders

Open Shading Language for Blender
A practical primer

Open Shading Language for Blender

By Michel Anders

Smashwords Edition

In remembrance of Kaatje,

you will be in our hearts, forever.

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.

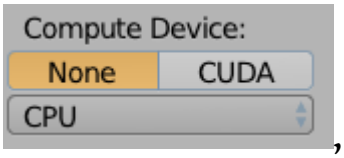
OSL is currently only available for Cycles (not for Blenders internal renderer) so we have to check if all prerequisites are met.

- 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,

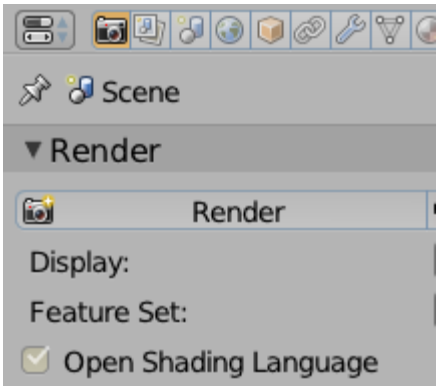
- verify that you have selected Cycles as your renderer as shown in the screen shot

- verify that you will use the CPU to render

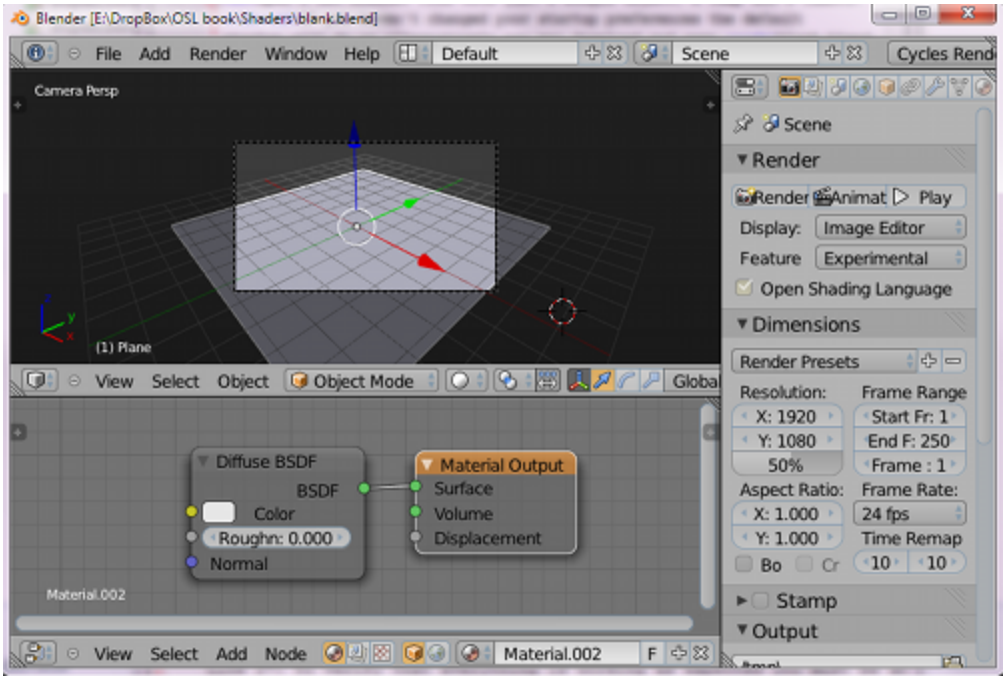
  (not the GPU as OSL is currently only available on the CPU)
  `File -> User Preferences -> System`
  

  ,

- 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)



`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

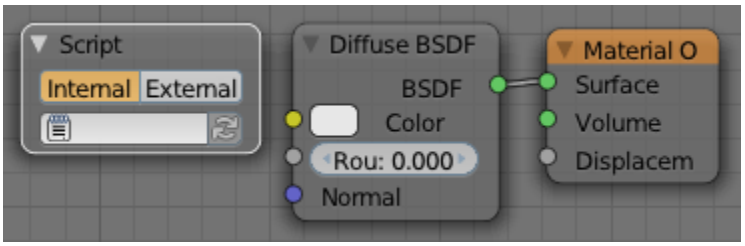
If you have written a shader the following steps are needed to use it as part of a node based material:

- 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

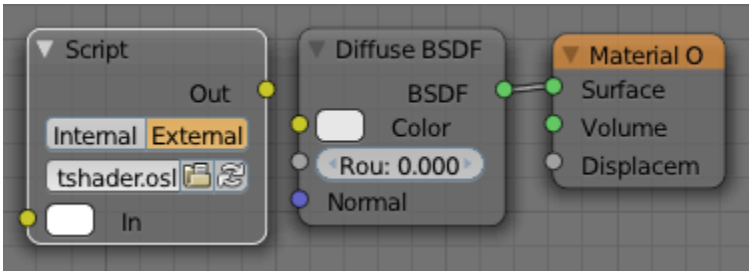
either an external script (= OSL code that resides in a text file on disk) or an internal script (= OSL code present in one of Blenders text editor buffers)



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.

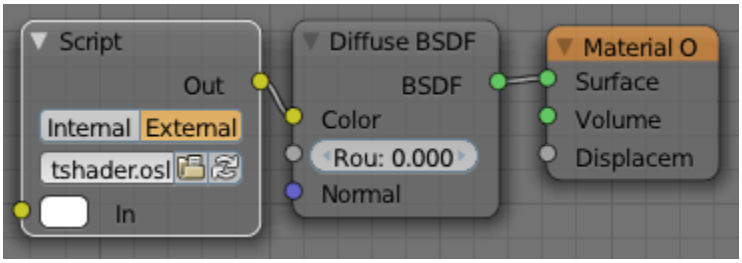
When you select either an internal text buffer or an external file, the shader is compiled automatically and if everything went well some input and output sockets will appear.



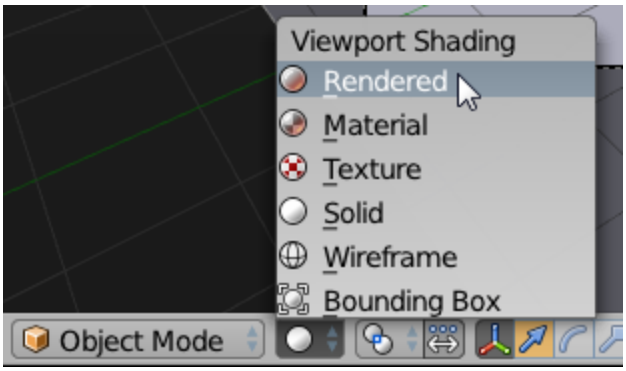
If something went wrong while compiling the shader a short message is shown in the top menu bar

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

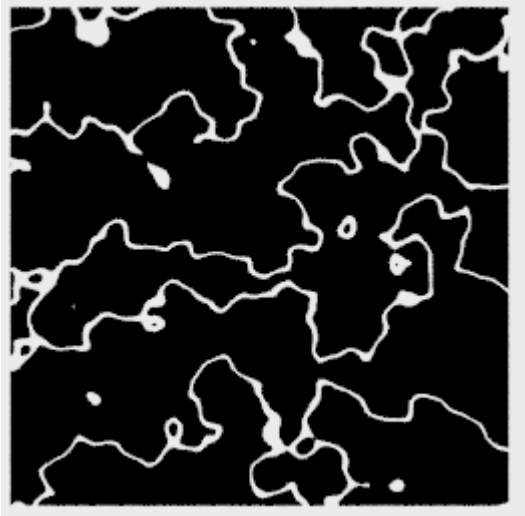
Focus areas

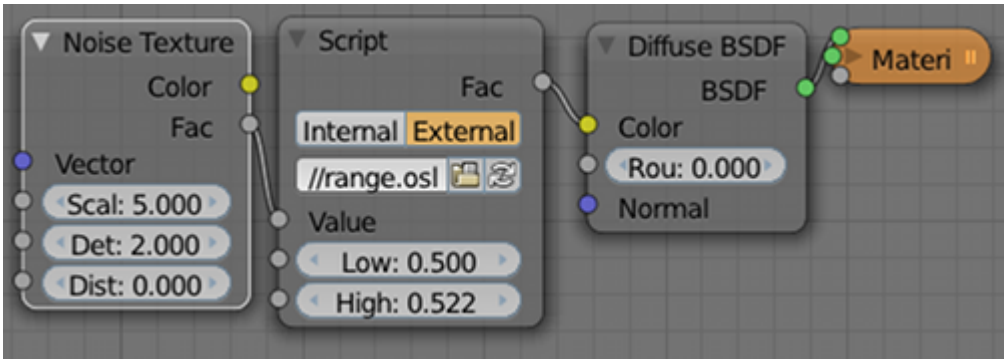
- making decisions, if/else statements
- repeating activities, for and while loops

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.





The code that implements this shader is shown below:

The code is available in Shaders/range.osl

```
01.  shader range(
02.    float Value = 0,
03.    float Low = 0,
04.    float High = 0,
05.
06.    output float Fac = 0
07.  ){
08.    if( Value >= Low && Value <= High ){
09.      Fac = 1;
10.    }
11.  }
```

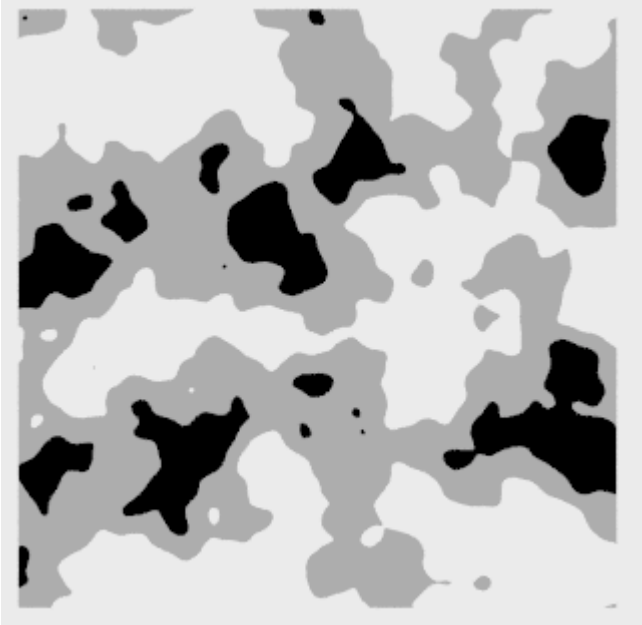
The shader has three input parameters: the `Value` we want to test and the `Low` and `High` values that specify the range that will return a value of 1. The result, either 0 or 1, is returned in the single output parameter `Fac`.

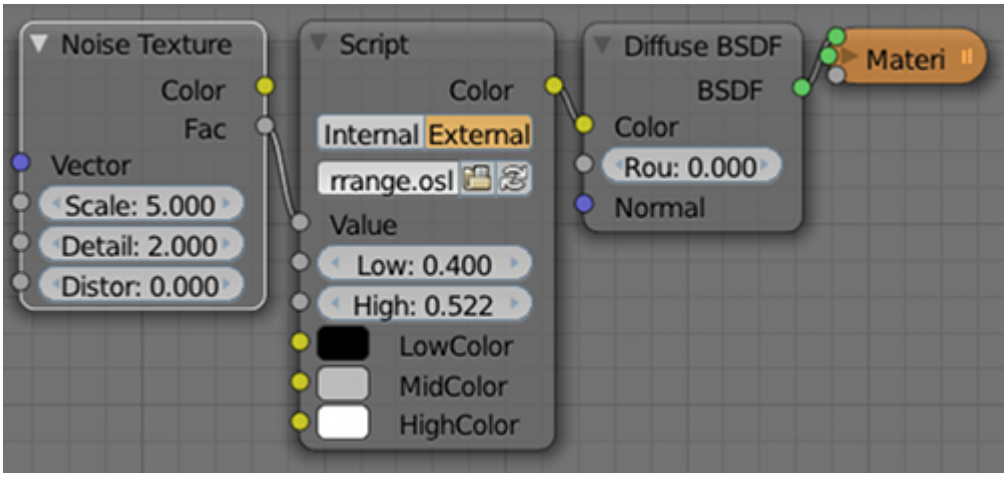
The body of the code consists of a single if statement (line 8). An if statement consists of an expression between parentheses and body. The body consists of a single statement or a group of statements between curly braces and is only executed if the expression of the if statement evaluates to non-zero. 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.





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

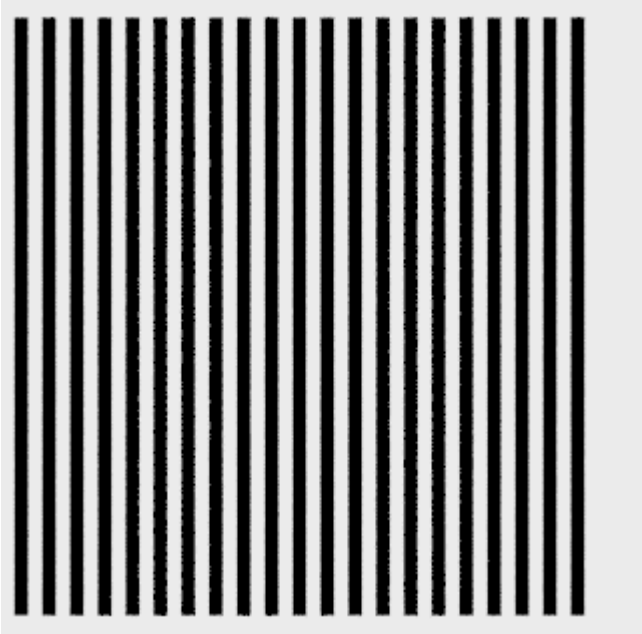
Making decisions is a fundamental feature of a programming language and so is the ability to repeat an action a given number of times. To this end OSL offers the for statement which is used in the next shader to produce a given number of stripes.

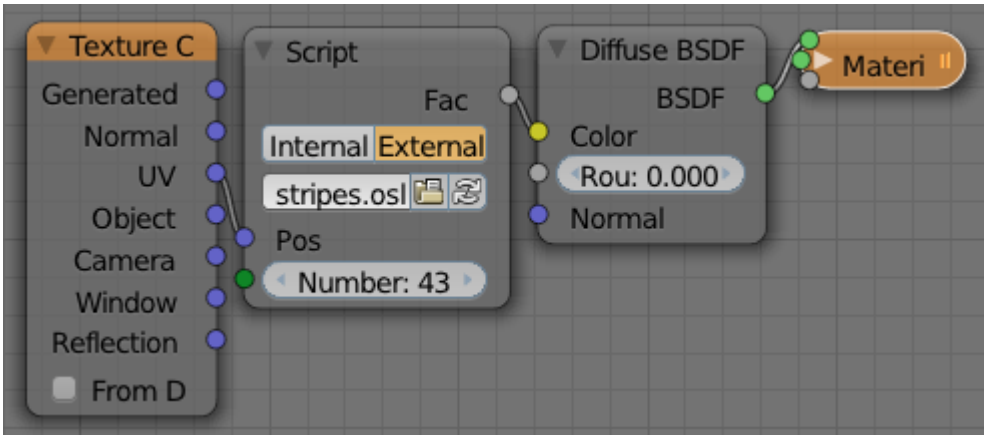
Code available in Shaders/stripes.osl

```
01.  shader stripes(
02.    point Pos = P,
```

```
03.    int Number = 4,
04.
05.    output float Fac = 0
06.  ){
07.    float x = mod(Pos[0],1);
08.
09.    int i;
10.    for(i=1; i <= Number; i++){
11.      if( x < (float)i/Number ){
12.        Fac = i % 2;
13.        break;
14.      }
15.    }
16.  }
```

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)

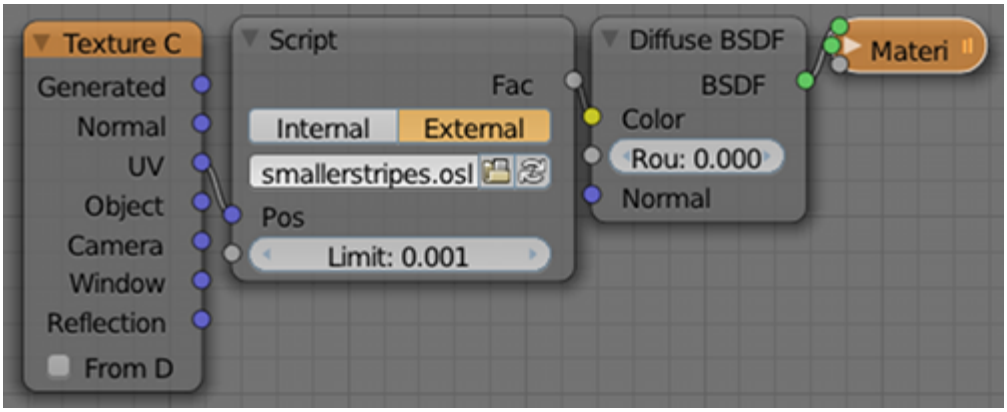
The body of the for statement itself consists of a single if statement. The expression in this if statement checks if the x coordinate is smaller than the upper limit of the stripe we are currently checking. For example, if `Number` is 4 (we want four stripes) the first time we check this expression 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.



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

```
01.  shader smallerstripes(
02.    point Pos = P,
03.    float Limit = 0.01,
04.
05.    output float Fac = 0
06.  ){
07.    float x = mod(Pos[0],1)*2;
08.    float dx = 0.5;
09.    float xlimit = dx;
10.
11.    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.

Points, vectors and normals

In this section we explore the point-like types of OSL. These include `point` itself but also `vector`, `normal` and even `color`. These types are all basically a list of three numbers and can often be used interchangeably.

Point-like types

In a point these three numbers represent the x, y and z coordinates and in vectors and normals they represent the x, y and z-components of a direction while in colors they are the red, green and blue parts. A point can be understood as a direction as well: it is the direction to travel from the origin at (0,0,0) to the point. The difference between a general vector and a normal is the context: a normal is a vector perpendicular to a plane (for example a face of a mesh) and although there are some subtle differences in the way they are handled during a transform (like a rotation) most of the time they can be used interchangeably. See the OSL language specification footnote on page 49 for the exact details. In the examples below and in most places in this book we most of the time refer to point, normals and vectors as just 'vectors'.

Vector operations

Vectors can be added and subtracted from each other: for example if you have two points and subtract point one from point two, you get a vector that represents the direction from point one to point two. The other way around, if you subtract point two from point one, you get a vector that represents the direction from point two to point one. The directions are reversed but the length of both vectors (the distance between the two points) is the same.

```
point p1 = {1,2,0};
point p2 = {2,1,0};

vector directionfrom1to2 = p2 - p1;
vector directionfrom2to1 = p1 - p2;

float distance12 = length(directionfrom1to2);
float distance21 = distance(p2,p1);
```

The last two lines in the snippet of code above demonstrate some of OSL's built in functions: if you have a vector and you want to know its length you can use the length() function. If you do not need the direction vector between two points but just the distance the built in function distance() will calculate that for you.

Vectors can be added as well:

```
point source = {1,0,0};
vector direction = {1,1,1};

point destination = source + direction;
point farbeyond = source + 100 * direction;
```

The final line of code shows that vectors can be multiplied as well. This is often called scaling and in this example the direction vector is scaled in a uniform manner, all three components are scaled by the same amount. It is also possible to scale each component of a point-like type by a different amount as illustrated in the following lines where we make the green component of a color 20 percent larger:

```
color grey = {0.5, 0.5, 0.5};
color greener = {1, 1.2, 1};

color greenish = greener * grey;
```

For vectors there are some operations that are performed quite often calculating the dot product and normalizing the length of a vector. The first may be used for example to determine if two vector are perpendicular (the dot product of those vector will be zero) while the second preserves the direction of a vector but makes its length equal to one. To appreciate how these operation work some understanding of mathematics is necessary and we cover that when we encounter them in real shaders but for now we just show a code snippet illustrating how the can be used:

```
vector a = {3, 0, 0};
vector b = {0, 3, 0};

if( dot(a,b) != 0.0 }{
   // they are not perpendicular
}

vector c = normalize(a); // c will be {1, 0, 0}
```

The cross product of two vectors will result in a vector that is perpendicular to both. The length of the resulting vector is the area of the parallelogram defined by the two vectors. Again we wont concern us with the mathematics here but being able to construct a vector that is perpendicular to two other vectors is an operation we will encounter quite a few times in real shaders so fortunately that is a built-in function:

```
vector a = {3, 0, 0};
vector b = {0, 3, 0};
// will be perpendicular to a and b. c = {0,0,9}
vector c = cross(a,b);
```

The final operation we encounter quite often an is provided as a built-in function is rotating a point around an arbitrary axis. The code snippet

below shows how to rotate a point around the z-axis (which originates in the origin an extend in the z-direction):

```
point p = { 1,1,1 };
point origin = { 0,0,0 };
point direction = { 0,0,1 };

// rotate by 45 degrees
point p2 = rotate(p, radians(45),
                  origin, direction);
```

Many more functions are available that operate on point-like types and they are all documented in the OSL Language specification. The ones presented in this section are the most common ones and should give you an indication of what is possible.

Mapping vectors

Focus areas

- Vectors
- Coordinate spaces

Cycles does have a mapping node to transform vectors but all the values for scaling, translating and rotating a vector are part of the node interface and not implemented as sockets. This way they might be animated but because they are not input sockets we cannot change them based on the output from some other node. Cycles does have a vector math node as well but that is rather limited as it doesn't provide options for rotation or scaling.

In this section we develop two separate nodes, one for multiplying each component of a vector with each corresponding component of another vector (i.e. scaling a vector), and another one for rotating a vector around another vector. We don't implement a translate node as this can be done by adding two vectors, an option that is already present in the vector math node.

Scaling

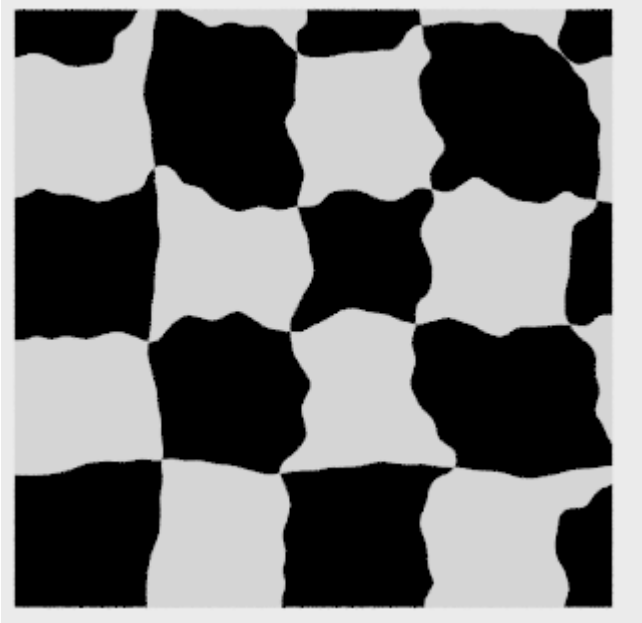
Scaling component by component is as simple using the built-in multiplication operator. The main task of this shader is defining input and output parameters with sensible defaults. Specifying defaults for shader parameters is mandatory in OSL both for input and output.

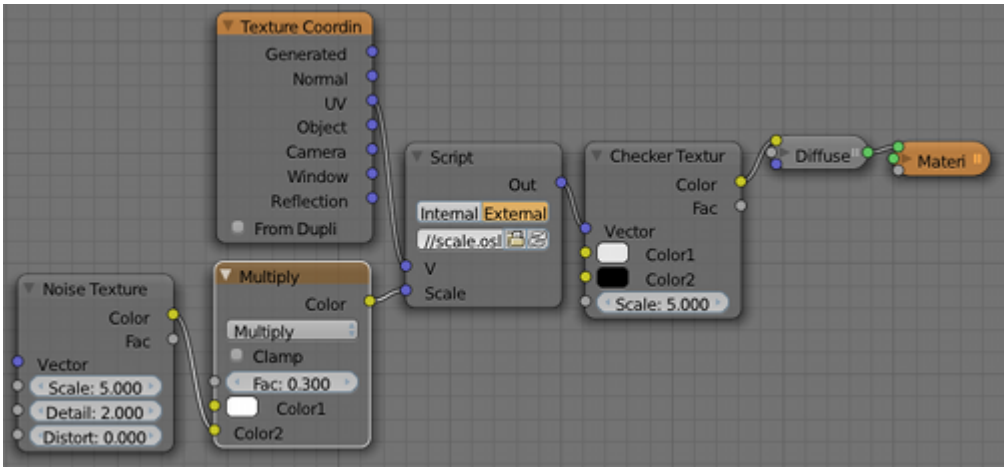
The following code is available in Shaders/scale.osl

```
01.  shader scale(
02.    vector V=0,
03.    vector Scale =1,
04.
05.    output vector Out=0
06.  ){
07.    Out = V * Scale;
08.  }
```

The input and output vectors `V` and `Out` are both initialized to `0`. Specifying a single number as an initializer for a vector parameter will initialize all three components of the vector to this same number. (The same is true for the vector-like types point, normal and color). The `Scale` vector defaults to all ones. If we don't connect anything to this socket there will be effectively no scaling.

In the example below we used the scale node to distort a regular checker pattern by scaling the uv-coordinates by a some small amount of random noise.





Rotation

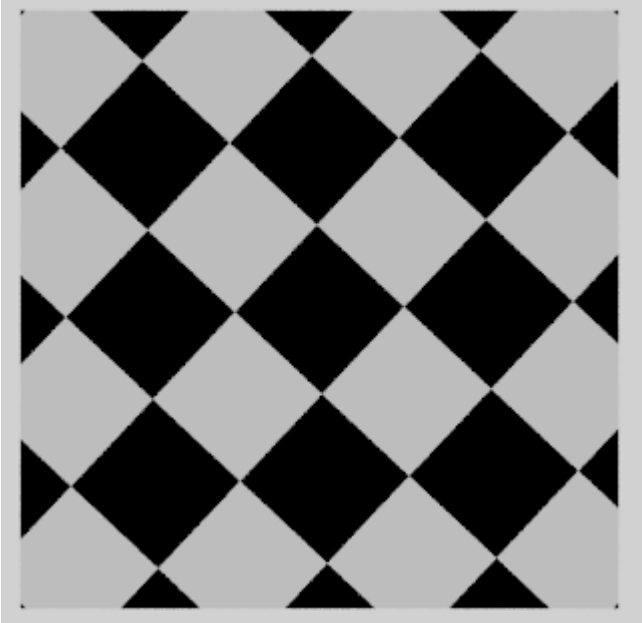
Like for scaling the work for this shader is mainly specifying sensible initial values but with a little twist: many people are not comfortable working with angles specified in radians so here we opt to convert from degrees to radians. The axis to rotate the vector around is specified by the the start and end points of a vector, labeled `From` and `To` in the shader. The start defaults to all zeros so if we want to rotate around some vector that starts at the origin (the most common case) we do not have to connect anything to the start input socket but if we want to

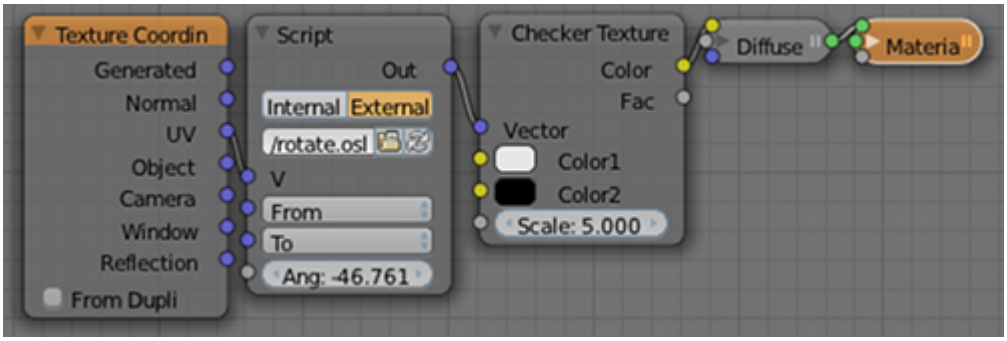
rotate around a vector that is not centered on the origin we still have the possibility.

The following code is available in Shaders/rotate.osl

```
01.  shader rotate(
02.    vector V = 0,
03.    vector From = 0,
04.    vector To = point(0,0,1),
05.    float Angle = 0, // in degrees
06.
07.    output vector Out = 0
08.  ){
09.    float rads=radians(Angle);
10.    Out = rotate(V, rads, From, To);
11.  }
```

With all the parameters in place the body of the shader concerns itself with just the conversion to radians from degrees (line 9) and calling the built-in `rotation()` function. In the example below we used the rotate node to rotate a checker pattern:





Coordinate systems

When we talk about the point being shaded the position of this point might be clear enough but the actual values of the x, y and z components depend on what we take as a reference point.

The built-in variable `P` is defined relative to the origin of the scene, i.e. it is defined in *world space*. World space is by no means the only coordinate system available in Blender and quite a few are available from the texture coordinate node (`Node->Input->Texture coordinates`), The most commonly used ones are:

- object

coordinates in object space are defined relative to the origin of the object being shaded. Using object space coordinates is useful if we want the texture that we are generating to stay at a fixed position on the object so that the position of the texture relative to the object does not change when the object moves

- generated

  coordinates in generated space are also fixed to the object being shaded but are relative to the center of the bounding box of the object and scaled to fit that same bounding box in such way that each coordinate will lie in the range [-1,1]

- camera

  coordinates in camera space are measured relative to the origin of the camera so they will not change when the camera moves.

- uv

  All coordinate system so far are three dimensional, but it is also possible to render based on uv-coordinates. To use this the object must have been uv-unwrapped, that is the surface must be projected onto a flat plane. uv-unwrapping is somewhat of an art in itself and although Blender has a quite extensive set of tools for this purpose, mastering those tools will require some effort. Fortunately there are quite a few tutorials available, a simple search will find you dozens.

- others

  There are even more coordinate systems available to use in Cycles and all of them can be found in Blender's user manual, they are listed under Texture Coordinates.

Patterns

Focus areas

- working with coordinates
- built-in global variables
- control structures

Checkers

Although shaders written in OSL can be used to create all sorts of nodes as we saw in the previous chapter, most people associate shaders with patterns that contribute to a material.

Shaders in OSL can be used to implement complete and highly configurable materials or simple reusable building blocks. The latter are often referred to as procedural textures or patterns and the checkers pattern is good example.

The checker texture is of course already part of Blender's collection of nodes (`Add -> Texture -> Checker`) but by implementing it ourselves we have the opportunity to get acquainted with control structures and some other language features and the all important concept of coordinates.

The code for the checkers is shown below:

Code is available in Shaders/checker.osl

```
01.  shader checker(
02.    point Pos = P,
03.    float Scale = 1,
```

```
04.    color Color1 = color(1,0,0),
05.    color Color2 = color(0,1,0),
06.
07.    output color Col = 0
08.  ){
09.    point p = Pos * Scale;
10.    int x = (int)mod(p[0],2.0);
11.    int y = (int)mod(p[1],2.0);
12.    int z = (int)mod(p[2],2.0);
13.
14.    if( ((x%2) ^ (y%2)) == (z%2) ){
15.      Col = Color1;
16.    } else {
17.      Col = Color2;
18.    }
19.  }
```

The checker shader that is shown here is implemented in pretty much the same manner as the one built into Blender. The first argument is the position that is being shaded, that is the position of the pixel for which this shader is called to return a color.

The position has as default the value of the global variable `P`. `P` is a variable that is made available by OSL which holds the position of the pixel we are currently rendering (or to be more exact, the sample. There may more samples per pixel). This position is in world space, that is the position is relative to the origin of the scene (actually it's in shader space which is renderer dependent. In Blender shader space equals world space).

We could leave out the `Pos` parameter and simply generate our checker pattern based on `P` but by this way it is possible for the end user to supply different coordinates by connecting one of the output sockets of a texture coordinate node (`Add -> Input -> Texture coordinates`). Choosing Object coordinates for example would ensure we generate the

pattern relative to the center of the object we are shading, which would make the pattern to stick to the object if the object would move.

The second input parameter, `Scale`, is a float value that allows us to scale the input coordinates in a uniform manner. A larger value will result in smaller checkers. If non uniform scaling is necessary we can always use a Mapping node (`Add -> Vector -> Mapping`, or use the vectormap node from the previous section) to scale a texture coordinate before connecting it to the checker node.

The checker pattern consists of two alternating colors, therefore we supply two color input parameters. The two slashes (`//`) mark the beginning of a comment in OSL. Everything after these slashes till the end of the line is ignored by the compiler. It's always a good thing to explain things in code that aren't immediately obvious.

A single output parameter `Col` will receive one of the input colors depending on which checker we are in.

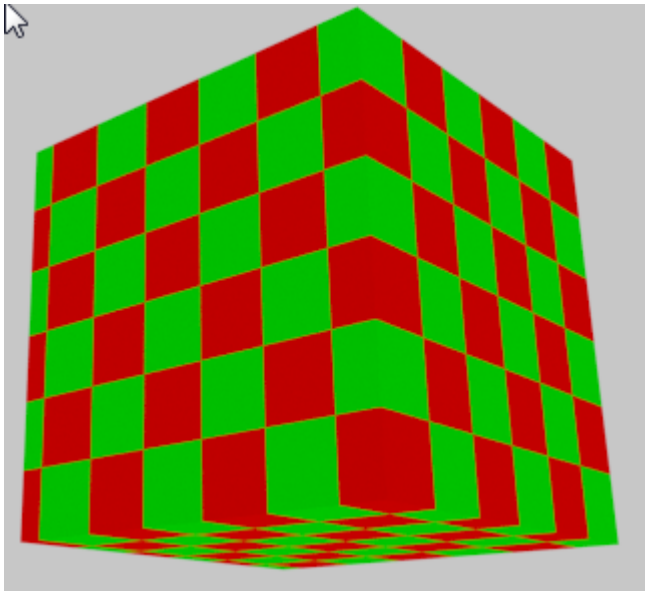
The body of the shader starts with scaling the input position and declaring three integer variables `x, y, z` that will hold the integer part (the part before the decimal point) of each individual coordinate after we calculated its modulus of two which will leave each coordinate in the range [0.0, 2.0) (exclusive 2.0, that means it will never be equal to 2.0). The result is still a float value and to convert a float value to an integer we need the *cast operator* `(int)`. (The other way around, converting an int to a float does not need a cast.)

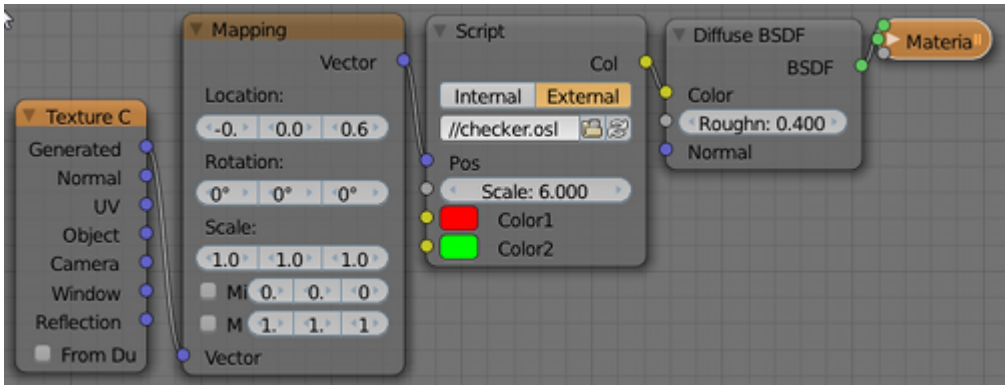
At this point we have three integer coordinates which will either 0 or 1. We can use these values to determine which checker color we want. If we think of the two dimensional situation with just the x and y coordinates we see that in even rows we color the odd columns while in odd rows we color the even colors.

We can use the modulo operator % to determine if an integer is odd and use the xor operator ^ to see if either the x coordinate or the y coordinate is odd (but not both). By comparing this result to the z coordinate being odd/even we can alternate the checker pattern in three dimensions (line 14). If you are not certain of the precedence of operators it never hurts to be liberal with parentheses as we did here.

We act on this result with an if statement. An if statement in OSL has a condition between parentheses. If this condition is true, the following statement is executed. This may be a single statement terminated by a semicolon or a compound statement surrounded by curly braces. An optional else part is executed if the condition is false. if/else statements may be nested but then it may become unclear which statement belongs to which if/else part therefore it is advisable to always use curly braces to prevent confusion (as is done throughout this book).

The result might look like this. An example node setup is given as well.





Triangles

Regular patterns are one of the reasons why you would use OSL because although some patterns might be constructed from combinations of built in nodes, other patterns would require an inordinate amount of nodes or are impossible to create.

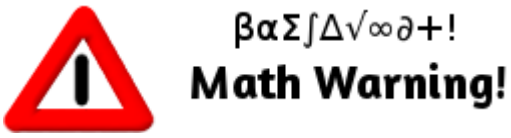
Not all patterns are as straightforward as the checkers pattern and in the following section we will see how the transformation of coordinates can make even geometrically quite complex patterns simple to implement. This is demonstrated with a triangles shader which will produce a pattern of equilateral triangles. The code looks like this:

Code available in Shaders/triangles.osl

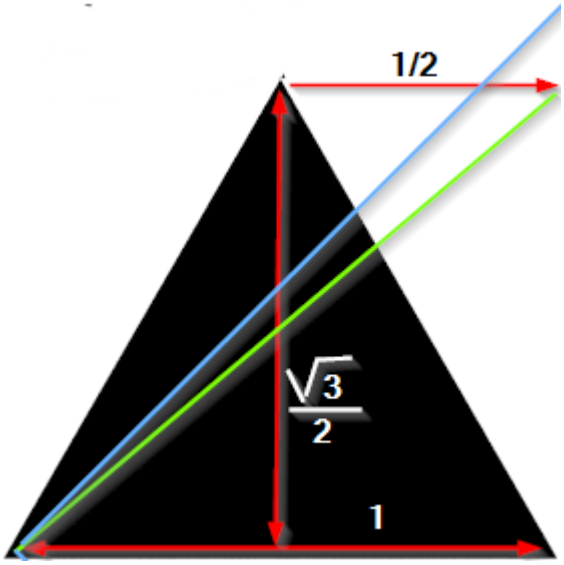
```
01.  shader triangles(
02.    point Pos = P,
03.    float Scale = 1,
04.    color Color1 = color(1,0,0),   // red
05.    color Color2 = color(0,1,0),   // green
06.
07.    output color Col = 0
08.  ){
09.    point p = Pos * Scale;
10.
11.    // ignore the z component, this is a 2D pattern
12.    float y = p[1] / (sqrt(3)/2);
13.    float x = mod(p[0] - y/2, 1.0);
14.    y = mod(y,1.0);
15.    if( y > 1 - x ){
16.      Col = Color1;
17.    } else {
18.      Col = Color2;
19.    }
20.  }
```

The input and output parameters are identical to those of the checkers shader and the body starts of in a similar way by scaling the input coordinates.

This pattern is two dimensional, that is it looks the same regardless what the z-coordinate is. We therefore only look at the x and y coordinates. We skew the x coordinate depending the y coordinate (line 13) and scale the y coordinate before we force both coordinates to lie in the range [0,1] with help of the `mod()` function. (the mod function divides a value by another value and returns its remainder. For example, `mod(1.7,1.0)` will return 0.7)



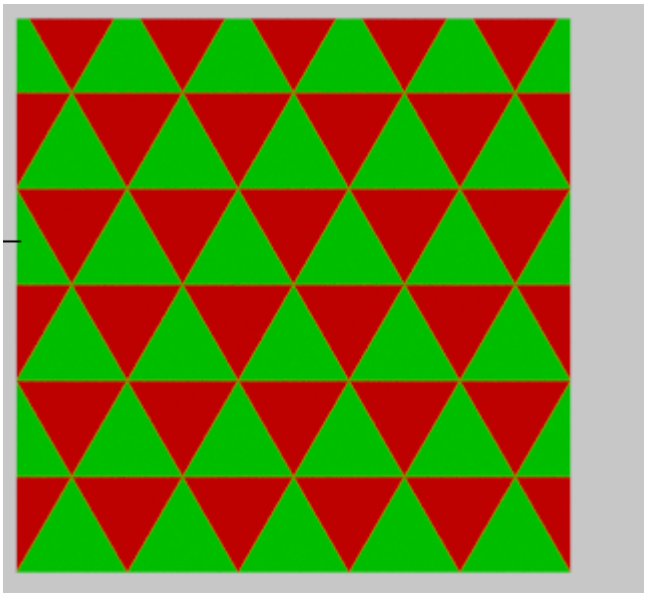
The constants used in the transformation are not magical but easy to derive with a bit of geometry.

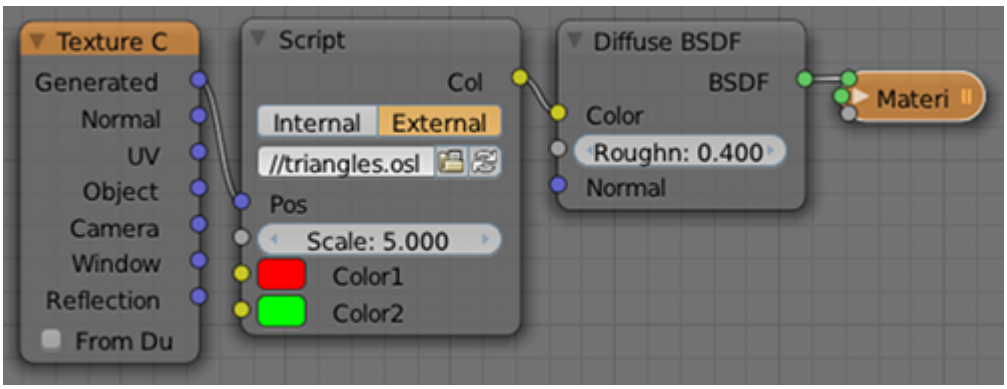


If we imagine an equilateral triangle (black) with sides of length 1 to lie with its base along the x-axis we see that the amount we have to skew when the y-coordinate is equal to the height of triangle is 1/2. The height is calculated with Pythagoras' formula as sqrt(1^2-1/2^2) = sqrt(3)/2. In other words, we have to skew x by half a unit for every sqrt(3)/y to get a rectangular triangle. The sloping side of this triangle is indicated in green. The subsequent scaling of y by the inverse of sqrt(3)/2 will ensure we now operate on a one by one square which is evenly divided by its diagonal (in blue).

These preparations of the input coordinates reduce the question of which color to use to determining whether the y coordinate lies above the diagonal that runs from the top left to the bottom right, which is exactly what the if statement in the body of the shader does.

The result with an example node setup is shown below.




Texture C
Generated
Normal
UV
Object
Camera
Window
Reflection
From Du
Script
Col
Internal
External
//triangles.osl
Pos
Scale: 5.000
Color1
Color2
Diffuse BSDF
BSDF
Color
Roughn: 0.400
Normal
Materi

Polka dots

Focus areas

- control structures
- cell noise

Randomness

When designing material shaders randomness plays an important role. Shaders that mimic naturalistic materials rely heavily on all sorts of randomness (or noise as it is often called) because nature never seems to repeat herself. Regular patterns that mimic man made materials often benefit from some random variations too as 100% perfection is near impossible to attain. For these kind of materials some noise may be used to make the material look weathered or dirty or convey a hint about when it was made (for example iron beams made in Victorian times are a lot less smooth than those made in the 21st century).

Given the importance of randomness it is not surprising that OSL comes equipped with a fair number of built-in noise functions and in the next section we encounter a special one called cell noise.

A randomly positioned circle

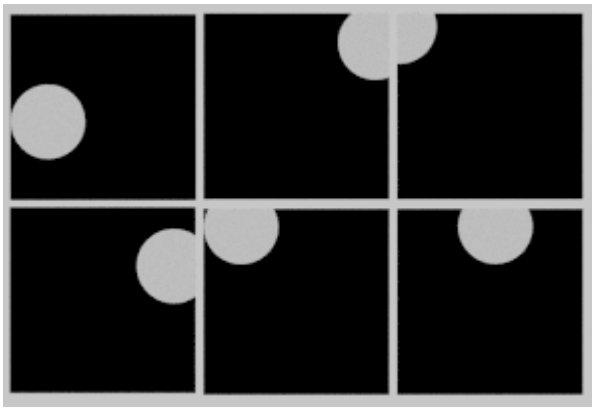
Most random number functions will return a new random number every time it is called but this often not what we want if we are rendering patterns. This may sound strange a first but lets have a look at what happens when our shader is called to shade a point. Imagine our shader generates a single white circle that is centered at some random location. Now each time the shader is called it should calculate a random position

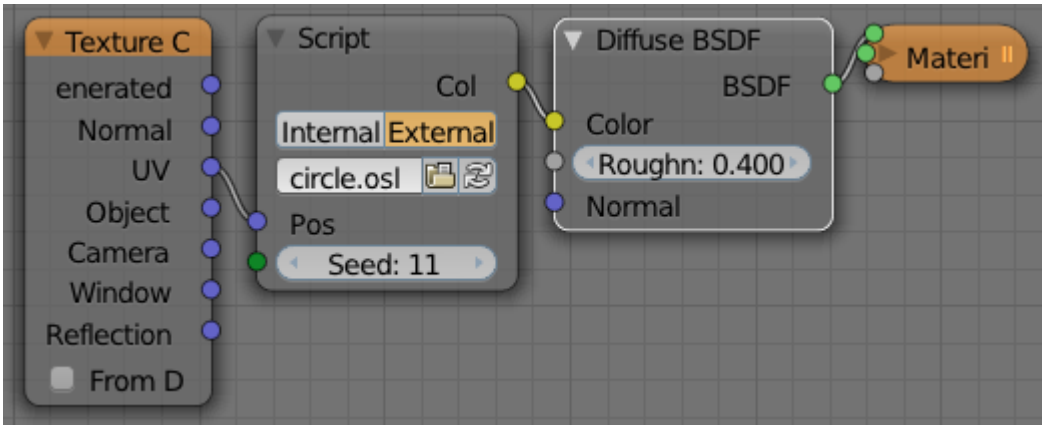
for the circle but each time this should be the same position. This sounds like a contradiction but let us look a shader which implements this scenario:

Code available in Shaders/randomcircle.osl

```
01.  shader randomcircle(
02.    point Pos = P,
03.    int Seed = 1,
04.
05.    output color Col = 0
06.  ){
07.    point center=(vector)noise("cell",
08.                               Seed);
09.    center[2] = 0;
10.    if(distance(Pos, center) < 0.2){
11.      Col = 1;
12.    }
13.  }
```

The result for several different values of `Seed` is shown in the image together with the node setup.





The magic is in the `noise()` function. This is a very versatile function which can generate all kinds of noise with different characteristics. In this case we choose "cell". Cell noise will return a random value based on the integer part of its argument.

Lets first look at the value that the `noise()` function returns. In OSL functions may be overloaded which means that it is possible to define several functions with the same name. Which variation is executed depends on the *signature* of the call. The signature is the combination of the type of the return value and the number and types of the arguments passed to the function. The `noise()` function in OSL has a large number of signatures. Not only may it return either a float or a point-like type but it may be passed several different types of arguments. In the randomcircle shader we assign the return value of the `noise()` function to a point variable so it will return a point with three random components (line 7. While writing this book the OSL library that is used

by Blender contains a bug: if the return type is a `point` all components would receive the same value. Therefore we use a cast to a vector type here which will cause noise to really return three distinct values). Each component will have a value in the range [0,1].

Now the `noise()` function does not return a new random value every time it is called but it returns the same value if it is passed the same arguments. In our example we pass a single value `Seed` that can be changed by the user. As long as `Seed` is the same, the same random point will be generated but for a different `Seed` value a different random point will be returned. In the next example we will elaborate on this and pass different arguments to the `noise()` function. For now it is important to understand that returning a random point in a reproducible way makes it possible to compare the position of the point currently being shaded to the same circle center, no matter what this location is or how often we call the noise function.

Lots of random dots

Lets suppose that instead of a single randomly placed circle we want to generate a whole field of small colored dots. This means we have to generate a great number of random positions and also a great number of random colors.

In the shader shown below we tackle this problem by imagining that the space we are shading is divided into small cubes of unit size. For each of those cubes we may generate a random point inside it that acts as the center of our dot (that is a sphere because we're working in three dimensions here).

In the previous section we saw that the `noise()` function returns the same random point if passed the same arguments. This is fine but we want this point to be the same for all positions inside some unit cube.

For example, we should receive the same random point if we pass the `noise()` function the point (4.7, 3.2, 2.5) or the point (4.3, 3.4, 2.3) which are different but lie inside the same unit cube.

This is where the "cell" variant of the `noise()` function comes in handy: it will round down its arguments to the nearest integer before calculating a random value so if passed the point (4.3, 3.4, 2.3) or the point (4.7, 3.2, 2.5) it will in both cases return a random value based on the point(4, 3, 2).

However, for the dots shader we need two random values; one for the position and a different one for the color of the dot. The `noise()` function may be passed an extra argument besides our position so we may generate more than one random value for the same position by varying this extra argument.

Code available in Shaders/dots.osl

```
01.  shader dots(
02.    point Pos = P,
03.    float Scale = 1,
04.    float Radius = 0.05,
05.
06.    output float Fac = 0,
07.    output color Col = 0
08.  ){
09.    point p = Pos * Scale;
10.
11.    float x,y;
12.    for(x=-1; x<=1; x++){
13.      for(y=-1; y<=1; y++){
14.        for(float z=-1; z<=1; z++){
15.          vector offset=vector(x,y,z);
16.          point dotposition =
17.            floor(p+offset) +
```

```
18.             (vector)noise("cell",
19.             p+offset,1);
20.           float d = distance(p,
21.                              dotposition);
22.           if( d < Radius ){
23.             Col = noise("cell",p+offset,2);
24.             Fac = Radius - d;
25.             y=2; x=2;
26.             return;
27.           }
28.         }
29.       }
30.     }
31.   }
```

The input parameters of the dots shader are the position, a scale factor and the radius of the dots. It has two output parameters, one float parameter `Fac` that will be non-zero if we're inside a dot and a color parameter `Col` that will be assigned a random color when inside a dot.

After scaling the position in the first line of the body we encounter several new language features of OSL. First we declare two variables in one statement. In fact any number of variables of the same type may be declared in a single statement. We use these variables in the for statements that follow it. A for statement has three expressions that control how often the statements in its body are executed.

The first expression is used to give a start value to a variable. Here we initialize the `x` variable with the value `-1`. The second expression of a for statement is a condition. As long as this condition evaluates to true (or something non zero) the body of the for statement will be executed. After each execution the third expression will be evaluated. In this example this means that the value of the `x` variable will be `-1` at first, then `0` and finally `1` inside the body.

The body consists of a similar for statement that does the same for the `y` variable and the innermost for statement does likewise for the `z` variable. This last for statement shows that is possible the declare a variable and assign it an initial value in one go.

At this point we arrive at the statements in the body of the innermost loop with three variables `x, y` and `z` that each hold some unit offset and you may wonder why we need these values (line 15). When we generate a random point inside a neighboring unit cube this point may be positioned so close to the side of the cube that its radius may extend into to the unit cube where the point we are shading is situated. This implies that we not only have to check whether the point being shaded is inside a sphere with its origin inside its own unit cube but also that we have to check if we are in range of nearby spheres in neighboring cubes.

And that is exactly what we do: we take unit steps in all three principal directions and add this offset to the position of the point being shaded. This new position is passed to the `noise()` function whose "cell" argument will round these positions down to nearest integer and return a random point based on the combination of this position and an extra argument with the value 1.

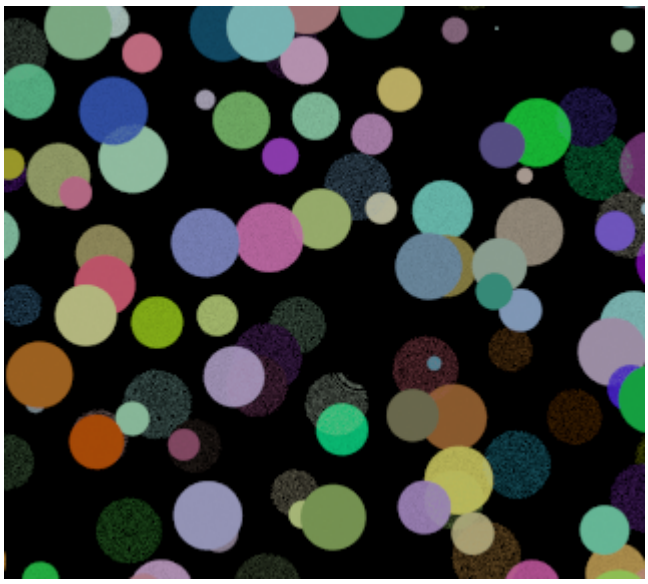
If the distance between the point being shaded and the random point returned by the `noise()` function is smaller than the chosen radius we are inside the sphere. We then assign a random color to the `Col` output parameter. This color is returned by the `noise()` function based on the same calculated position plus a different additional value of 2. This way the position of the sphere inside the unit cube and the color it is assigned will be completely independent.

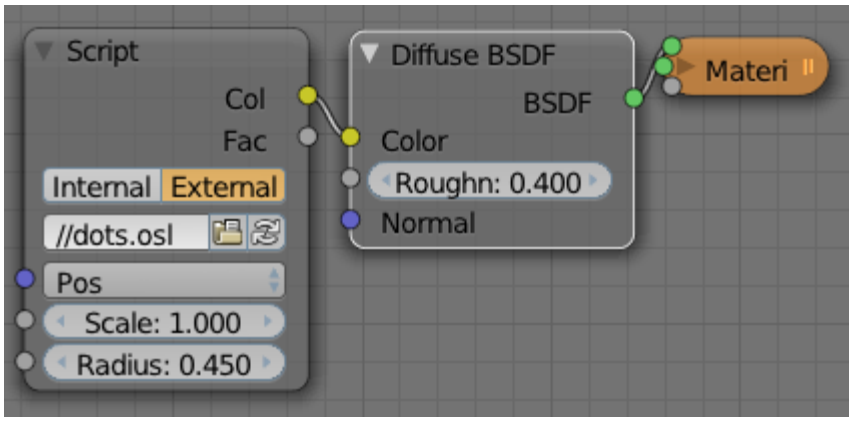
The value we assign to the `Fac` output variable is not simply a non-zero value but the distance to the edge of our dot. This is an approach we will use often: it costs hardly anything to supply this additional bit of

information but it allows an artist to use this shader in ways we might not have thought of (this could for example be used to add a color gradient or a displacement to the dot).

Finally the return statement ensures we stop executing the for statements and in fact makes us leave the shader altogether (line 26). It is a good thing to stop looking for spheres as soon as we have found one because those nested loops are time consuming so we better take a shortcut when we can.

An example of the dot pattern and an example node setup are shown below.





Leopard shader

Focus areas

- geometric operations with vectors
- randomizing patterns

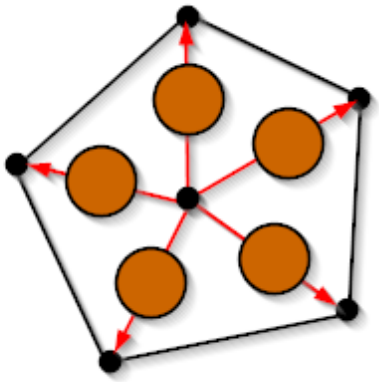
The spots on the pelts of felines come in a variety of shapes. One of the most striking patterns to me are the rosettes of the leopard. The slightly irregular deep black outlines of the rosettes on the lighter background create a very appealing pattern.


(credit: U.S. Fish and Wildlife Service)

On close inspection we see that often the the rosettes are almost a polygon, albeit with irregular edges. The rosettes often have five sides, but four and six can be found as well. Such a distribution of polygons is something reminiscent of voronoi noise but it is a bit more complicated than that. (There seems to be consensus that most coat patterns are formed by reaction diffusion reactions during the development of the embryo but that is completely impractical to simulate in a shader language)

The tools

The idea is to find out first which point of a collection of randomly positioned seed points is closest and then find the points from the same collection that surround the first one and construct a convex polygon from those points. We then position the dots that make up the rosette along the lines from the center point to the polygon vertices.



Finding the seed point closest to where the point we are shading is located is done by dividing space in cells, generating a random point (seed) in each cell and check which of these points is closest.

Code available in Shaders/leopard.osl

```
01.  // bubble sort. fast enough
02.  // for small collections
03.  void sort(int np,
04.             output point p[],
05.             output float f[])
06.  {
07.      int n=np;
08.      for(int j=0; j<np; j++){
09.         for(int i=1; i < n; i++){
10.            if( f[i-1] > f[i] ){
11.               float tf=f[i-1];
12.               f[i-1]=f[i];
13.               f[i]=tf;
```

```
14.                 point tp=p[i-1];
15.                 p[i-1]=p[i];
16.                 p[i]=tp;
17.               }
18.           }
19.           n--;
20.       }
21.   }
```

`sort()` is just a utility function used to sort an array of points based on their corresponding values in a second array. This second array can hold anything but in the shader we will use it for values representing distances and angles. The algorithm used to sort the array is a simple bubblesort. Much faster algorithms exists but they are both more complex and unlikely to be faster for the very small arrays (8 or 9 points) we encounter in our shader.

Code available in Shaders/leopard.osl

```
01.  void voronoip9d(
02.    point q,
03.    float T,
04.    output point vp[9],
05.    output float vd[9])
06.  {
07.    point p = point(q[0],q[1],0);
08.
09.    float xx, yy, xi, yi;
10.
11.    xi = floor(p[0]);
12.    yi = floor(p[1]);
13.
14.    int i=-1;
15.    for(xx=xi-1; xx<=xi+1; xx++){
16.      for(yy=yi-1; yy<=yi+1; yy++){
17.        i++;
```

```
18.         vector ip =
19.           vector(xx, yy, 0);
20.         vector np = cellnoise(ip);
21.         vp[i]=ip + np*T +
22.          vector((1-T)/2,(1-T)/2,0);
23.         vp[i][2] = 0;
24.         vector dp = vp[i] - p;
25.         vd[i] = dp[0]*dp[0] +
26.                 dp[1]*dp[1];
27.       }
28.     }
29.     sort(9,vp,vd);
30.   }
```

In the code above we simply sort the distances to the nine random points. The first point in the result array is the one closest to our location. For reasons that will become clear we have an additional parameter `T` that controls how much a random point may deviate from the center of the cell. A value of 1 will result in a completely uniform distribution while smaller values will confine a random point closer to the center of the cell (lines 21/22).

Once we have the point closest to the position being shaded we need the points that make up the vertices of the polygon. We therefore generate the random seed points in the cells around the center point we selected earlier. These points are not necessarily all the same one we generated earlier but one of them will be the center point itself, while the other eight will returned as a sorted list. The function `voronoip8a()` collects and sorts those points and looks a lot like the previous function but instead of by distance the points will be sorted on their relative orientation to some arbitrary reference axis (line 26 in the code below calculates the angle relative to the x-axis). This way these neighboring points will circle the center point in clockwise fashion.

Code available in Shaders/leopard.osl

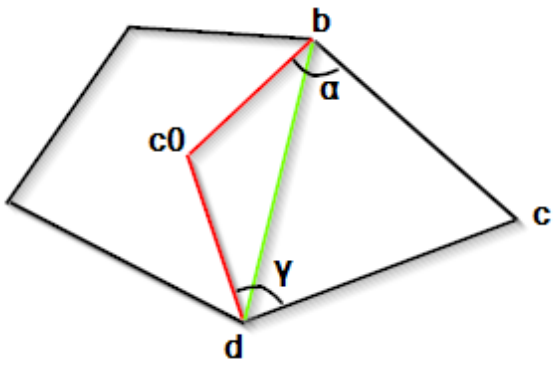
```
01.  void voronoip8a(
02.    point q,
03.    float T,
04.    output point vp[8],
05.    output float angles[8])
06.  {
07.    point p = point(q[0],q[1],0);
08.
09.    float xx, yy, xi, yi;
10.
11.    xi = floor(p[0]);
12.    yi = floor(p[1]);
13.
14.    int i=-1;
15.    for(xx=xi-1; xx<=xi+1; xx++){
16.      for(yy=yi-1; yy<=yi+1; yy++){
17.        vector ip =
18.          vector(xx, yy, 0);
19.        vector np = cellnoise(ip);
20.        np[2] = 0;
21.        vector vvp = ip + np*T +
22.         vector((1-T)/2,(1-T)/2,0);
23.        vector d = vvp-p;
24.        if( dot(d,d)>1e-7 ){
25.          float angle =
26.            atan2(d[1],d[0]);
27.          i++;
28.          vp[i]=vvp;
29.          angles[i]=angle;
30.        }
31.      }
32.    }
33.    sort(8,vp,angles);
34.  }
```

The sorted list of vertices defines a polygon that surrounds our center point. This polygon may have all sorts of spikes that we will have to eliminate. The function `closest_polygon()` will do that but will not actually guarantee that the list of vertices it returns will define a convex polygon but it will suffice for our purpose. (To guarantee this we would have to take a completely different approach and calculate the actual voronoi tessellation which would hurt performance and complicate the code a lot.)

```
01.  int closest_convex_polygon(
02.    point c0,
03.    point vp[8],
04.    output point mp[8])
05.  {
06.    int imp[8] = { 0, 1};
07.    int nmp = 2;
08.
09.    int delauney(int b,int c,int d)
10.    {
11.      float alpha = acos(
12.        dot(vp[b]-vp[c],vp[b]-c0));
13.      float gamma = acos(
14.        dot(vp[d]-vp[c],vp[d]-c0));
15.      return (alpha+gamma) <= M_PI;
16.    }
17.
18.      for(int i=2; i<8; i++){
19.        if( delauney( imp[nmp-2],
20.                      imp[nmp-1],
21.                      i) )
22.        {
23.           imp[nmp]=i;
24.           nmp++;
25.        }else{
26.           imp[nmp-1]=i;
```

```
27.         }
28.       }
29.       while( (nmp > 3) &&
30.             ! delauney(imp[nmp-2],
31.                        imp[nmp-1],
32.                        imp[0]) )
33.       {
34.           nmp--;
35.       }
36.
37.       for(int i=0; i<nmp; i++){
38.           mp[i] = vp[imp[i]];
39.       }
40.       return nmp;
41.   }
```

The `delauney()` function (line 9) that is used to weed out vertices that form spikes returns true if the sum of the angles alpha and gamma (see illustration) is smaller than 180°. If not the two triangles (c0 - b -d and b - c - d are too narrow and the middle vertex c is discarded and the shorter side (in green in the diagram) selected.



In the function `closest_convex_polygon()` we add the first two points to our collection at the start and then we step along each sequence of 3 corner points and check if the the delauney condition is met (line 19). If true we add the new point (d) to our collection and if not (i.e. c is to

'spikey') we let replace d the previous point (line 26). Because we added the first two points of the polygon without checking, we still need to look if on closing the polygon those two points are not sticking out and if so remove them from our collection (line 29). We wrap up by copying the collected points to the output array (line 37).

The shader

The final task is to construct a shader using the tools we developed so far.

Code available in Shaders/leopard.osl"

```
01.  shader leopard(
02.    point Pos = P,
03.    float Scale = 1,
04.
05.    color Blob = 0,
06.    color Inside = 0.5,
07.    color Outside = 1,
08.    color Center = 0,
09.
10.    float R = 184,
11.    float Radius = 2.4,
12.    float S = 1.4,
13.    float E = -2,
14.    float Shape = 0.5,
15.    float F = 0.01,
16.    float G = 11.0,
17.    float Cp = 0.2,
18.    float Cr = 0.005,
19.
20.    output color Color = Blob,
21.    output float D = 0,
22.    output float Edge = 0
```

```
23.  ){
24.    color colors[8]={color(1,0,0),
25.                     color(0,1,0),
26.                     color(0,0,1),
27.                     color(.6,0,0),
28.                     color(0,.6,0),
29.                     color(0,0,.6),
30.                     color(.3,0,0),
31.                     color(0,.3,0)
32.                     };
33.    point p = Pos * Scale;
34.    p[2]=0;
35.    point vp[9];
36.    float dp[9];
37.
38.    voronoip9d(p,Shape,vp,dp);
39.
40.    point nvp[8];
41.    float angles[8];
42.    voronoip8a(vp[0],
43.               Shape,nvp,angles);
44.
45.    point mp[8];
46.    int m = closest_convex_polygon(
47.        vp[0], nvp, mp);
48.
49.    float sump = 0;
50.    float mind = 1e7;
51.
52.    point mmp[8];
53.    for(int i=0; i<m; i++){
54.      mmp[i]=mix(vp[0], mp[i],
55.                 Radius/10);
56.      float d = distance(mmp[i],p);
57.      sump += S * pow(d, E);
58.      d= distance(vp[0],mp[i]);
```

```
59.       if( d < mind ){ mind = d; }
60.     }
61.
62.     if( noise("cell",p) < Cp
63.       && dp[0] < Cr )
64.     {
65.       Color = Center;
66.     }else if( sump < R ){
67.       float sumangle;
68.       if(point_inside_polygon(
69.            p+noise("snoise",G*p)*F,
70.            mmp,m,sumangle) )
71.       {
72.         Color = Inside;
73.       }else{
74.         Color = Outside;
75.       }
76.     }
77.
78.     // the distance from the
79.     // center of the rosette
80.     D = dp[0];
81.
82.     // for debugging, highlight
83.     // the edge of a cell
84.     if ( mod(p[0],1.0)<0.02
85.       || mod(p[1],1.0)<0.02)
86.     {
87.       Edge=1;
88.     }
89.   }
```

First it follows the steps in the outline sketched earlier: it finds the center point and constructs a polygon from the surrounding points (line 46). Then it calculates a field based on the distance to points of the rosette, which lie some distance away from the center on the lines from

the center point to the vertices of the polygon. If the value of the calculated field is larger than some configurable threshold Rwe are in a dark part of the rosette. Note that this is analogous to how implicit surfaces (or 'metaballs') are constructed.

How far the blobs are positioned from the center is controlled by the `Radius` parameter which is used as a mix factor to mix the position of the center point and the point on the polygon edge (line 54). It's dived by 10 to give some more control. The `S` and `E` parameters probably could have been given more descriptive names but together to control how much the individual blobs will run together. You'll have to experiment a bit to see their effect but the work by shaping the 'field' that originates in the polygon points.

Now that we have calculated the field value that defines the blob shapes the final step is to see if we are within the blobs. First however we generate another random number and check it against the center probability `Cp` to see if we have to draw a center spot as well and if so, check if we are within the center radius `Cr` (line 62).

If not, we check if we have a value lower than the threshold `R`. If so we leave the output color alone (meaning it will have the blob color because we made that the default), if not we check whether we inside or outside the polygonal area.

Because we want to generate a different color for the internal area of the rosette we define an additional function `inside_polygon()` that returns true if a given point is inside a polygon defined by a list of vertices. It does so by calculating the winding number, basically the sum of all angles between the point we are testing and the vertices on the edges. If that is a whole number of turns, i.e. a multiple of 2 * Pi, the point lies inside the polygon. More details can be found in the additional reading section below.

Code available in Shaders/leopard.osl"

```
01.  int point_inside_polygon(
02.    point p,
03.    point verts[],
04.    int nverts,
05.
06.    output float sumangle
07.  ){
08.    int i;
09.    sumangle=0;
10.    vector ref=
11.        normalize(verts[0]-p);
12.    vector ref2;
13.    float amod;
14.    for(i=1; i<nverts; i++){
15.      ref2 = normalize(verts[i]-p);
16.      sumangle += acos(dot(ref, ref2));
17.      ref = ref2;
18.    }
19.    ref2=normalize(verts[0]-p);
20.    sumangle+=acos(dot(ref, ref2));
21.    amod=abs(mod(sumangle,M_2PI));
22.    return (amod < 1e-2)
23.        || (M_2PI - amod < 1e-2) ;
24.  }
```

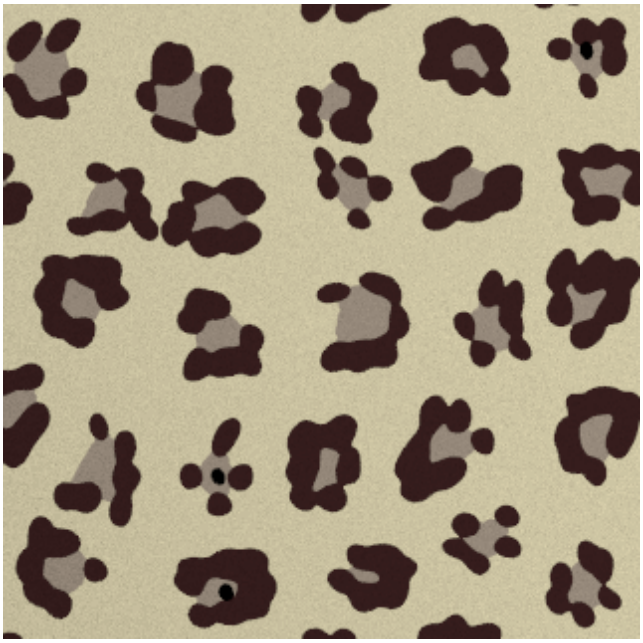
Because we do not want the edges of this interior to be defined by straight lines we perturb the point a bit by adding Perlin noise. The shape of this distortion is controlled by the `F` and `G` parameters, where the first primarily controls how large distortion there will be while the latter will influence the amount of wiggling.

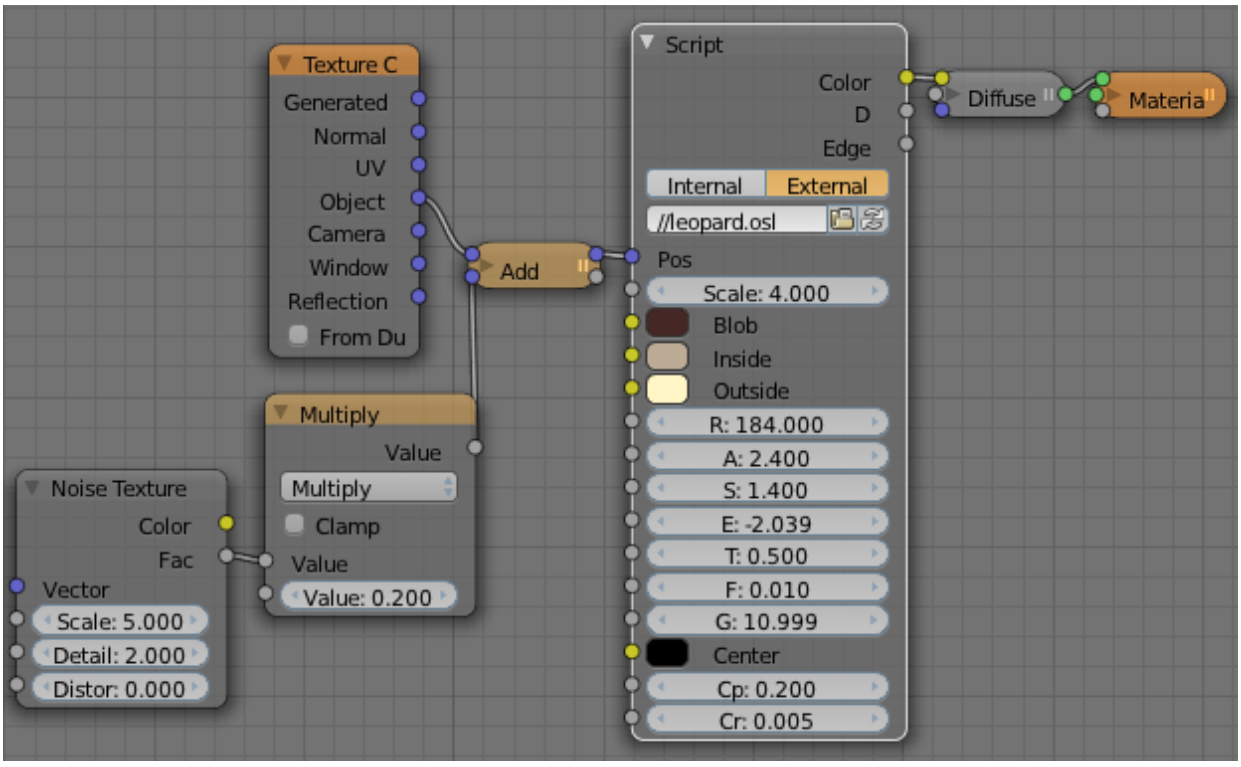
Because this is such a large and complex shader I deliberately left in some debugging aids in the form of the `D` and `Edge` output parameters.

The first is calculated to hold the distance to the center of the rosette while the latter can be used to visualize the grid cells of the pattern, which may also be useful for the artist to visualize the effect of adding coordinate distortion as we have done in the example node setup below.

Sample result and node setup

A sample texture created with the leopard shader and a node setup with the values used is shown below.





Note that the Shape parameter which controls the spread for the random points in the example node setup is less than one because the rosettes of a leopard's coat are not entirely uniform in their distribution. We also perturb the underlying geometry a little bit. This will result in spots that are not perfectly circular.

Additional reading

There is a lot more to tell on how the leopard got its spot and a an interesting introduction is this article. The underlying mathematics are not all that difficult but the results can be surprising. This WikiPedia article introduces reaction-diffusion systems.

Worley noise (called Voronoi noise in Blender) is described in somewhat more detail here.

More details on checking if a point lies within polygon by looking at the winding number can be found in this article.

Diamondplate

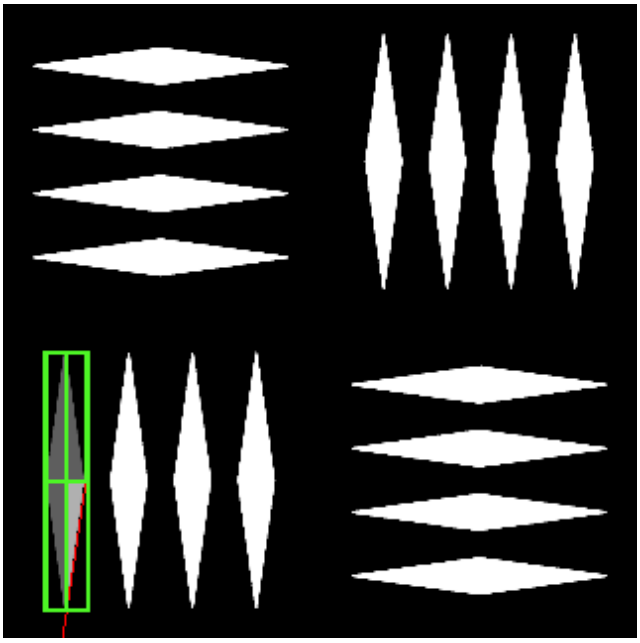
Focus areas

- Using symmetries in patterns
- Normals and displacement

Diamond plates are used to prevent slipping on truck floors and access ramps for wheelchairs but also as decorative element in store interiors and in modern kitchens. The base material may vary from aluminum to steel or even rubber but the distinctive pattern of diamond shaped elevations is a recurrent element.

The most common pattern seems to be the variant with four parallel diamonds but anything from one through five can be found in the wild. A shader that produces these patterns has to account for these variations but otherwise the number of input parameters is limited.

As outputs we provide both a displacement and a surface normal. Currently the support for true render-time displacement in Cycles is experimental, so we either need a bump node (`Add->Vector->Bump`) to convert a displacement to a surface normal or provide a surface normal ourselves. Here we opt for both to encourage reuse of the shader once true displacement becomes available for Cycles. (Blender does support mesh deformation with a displacement modifier but although this modifier can work with node based textures there is currently no option to create a programmable node based texture with OSL. The newly introduced (Blender 2.67) Python nodes might be able to this but this is outside the scope of this book).



The code is an excellent example of how you may reduce a problem to a few very simple lines once you remap coordinates. If you take a look at the diagram you see that the squares with vertical diamonds are copies from the ones with horizontal diamonds but with x and y coordinates interchanged. Likewise within a single diamond you see the quadrants outlined in green that have dark gray shapes all can be mapped to the lighter one by a combination of mirror operations around the x or y axis. Each diamond in a set of diamonds can of course be seen as a translated copy. So aside from the remapping the problem basically reduces to determining if we are above or below a single diagonal line (indicated in red in the diagram.

We go one step further though: because we want to give the diamonds a sloping side we will check whether the current point being shaded lies above, below or between two parallel lines.

Code available in Shaders/diamondplateshader.osl

```
01.  shader diamondplate(
02.    point Pos = P,
03.    float Scale = 2,
```

```
04.    int Diamonds = 4,
05.    float Margin = 0.1,
06.
07.    output float Bool = 0,
08.
09.    output int Fac = 0,
10.    output float Disp = 0,
11.    output normal Normal =
12.                        normal(0,0,1)
13.  ){
14.    point p = Pos * Scale;
15.    int xi = (int)floor(p[0]);
16.    int yi = (int)floor(p[1]);
17.
18.    float x = abs(fmod(p[0],1.0));
19.    float y = abs(fmod(p[1],1.0));
20.    int flipxy=0;
21.    float flipx=1, flipy=1;
22.    if( (xi%2)^(yi%2) ){
23.      float t = x;
24.      x = y;
25.      y = t;
26.      flipxy=1;
27.    }
28.
29.    if( (x < Margin)
30.      ||(x > 1-Margin)
31.      || (y < Margin)
32.      || (y > 1-Margin) )
33.    {
34.      Fac = 0; // do nothing
35.    }else{
36.      x -= Margin;
37.      if(y>0.5){
38.        y=1-y;
39.        flipy = -1;
```

```
      }
      y -= Margin;
      float w =
             (1-2*Margin)/Diamonds;
      x = mod(x,w)/w;
      if(x>0.5){
        x=1-x;
        flipx = -1;
      }

      float topx=0.15;
      float topy=0.5;
      float topy2=0.55;
      float botx=0.5;
      float boty=0;
      float a=(boty-topy)
             /(botx-topx);
      if(x>=topx){
        float uppery =
                  (x-topx)*a+topy2;
        float lowery = (
                   x-topx)*a+topy;

        if(y > lowery){
          if(y > uppery){
            Fac = 1;
            Disp= 1.0;
          }else{
            Fac = 2;
            Disp= (y-lowery)
                /(uppery - lowery);
            if(flipxy){
              Normal=normalize(
                   vector(-flipy,
                          flipx/a,
                          1));
```

```
76.             }else{
77.               Normal=normalize(
78.                    vector(flipx/a,
79.                           -flipy,
80.                           1));
81.             }
82.           }
83.         }
84.       }
85.     }
86.   }
```

After declaring input and output parameters with sensible defaults we reduce the scaled position to one-by-one squares (line 15). An exclusive or of the integer parts of the original coordinates determines whether we are in a square with vertical diamonds or in one with horizontal ones (line 22). In the latter case we flip the x and y coordinates and remember that we did so, information that we will use later when we generate normals.

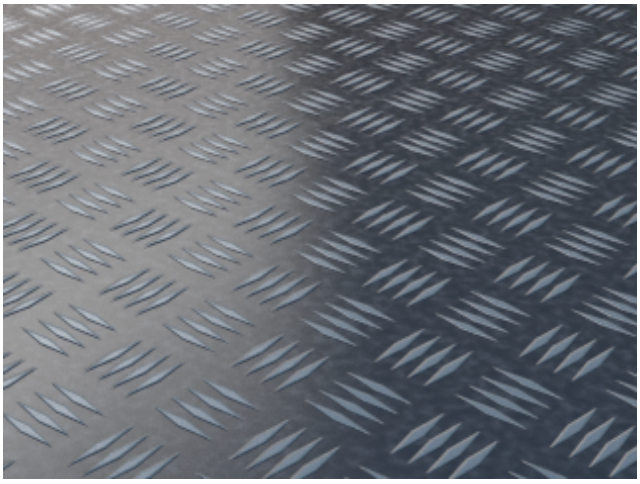
Next we check if we are actually within the area that contains the diamonds (line 29). If not we simply return. We adjust the x coordinate in such way that it starts at zero by subtracting the margin and then we mirror the y coordinate if its in the top half. Before we do the same for the x coordinate we determine the width of a single diamond to make sure the x coordinates for all diamonds are mapped to the range [0:1].

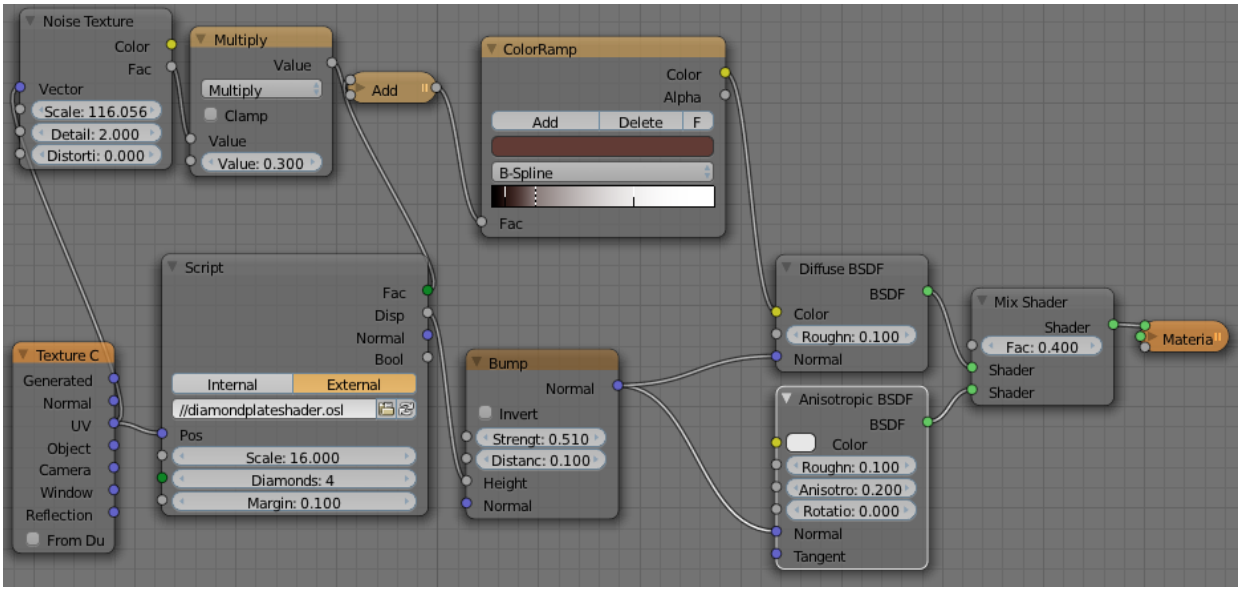
At this point we have both the x and the y coordinates in the range [0:0.5] so now we can calculate the slopes of the lines we are interested in. The slope for both lines is the same (they are parallel) and are calculated in line 55. When we are below the lowest of the two lines we are outside the diamond and if we are above them we are inside the diamond. Between the lines we're on the sloping side of the diamond and in that situation we do not only have to calculate the displacement but the normal as well and that's why we kept track of all the flips we made because the normal

on the slope does not only point up but should be perpendicular to that slope as well.

Examples

In the images below some noise was added in such a way that only the color between the diamonds was affected (because there the `Fac` is 0 and adding a tiny bit of noise will only sample the left part of the color ramp.





Symmetry operations

Focus areas

- Symmetry transformations
- Include files

In many instances when replicating a pattern in the form of a shader, programming that pattern might be greatly simplified by utilizing all symmetries available. We already exploited that for the diamond plate shader but if we want model more and more complicated patterns it might be a good idea to separate those symmetry operations into a library.

In all the following examples we assume both x and y to lie in the range [0,1], which is a range suitable for dealing with most (unscaled) uv-maps. All symmetry functions are available in the include file `Shaders/symmetry.h` for reuse in other shaders but each sample shader presented below is self contained so it's easier to keep an overview.

The letter F

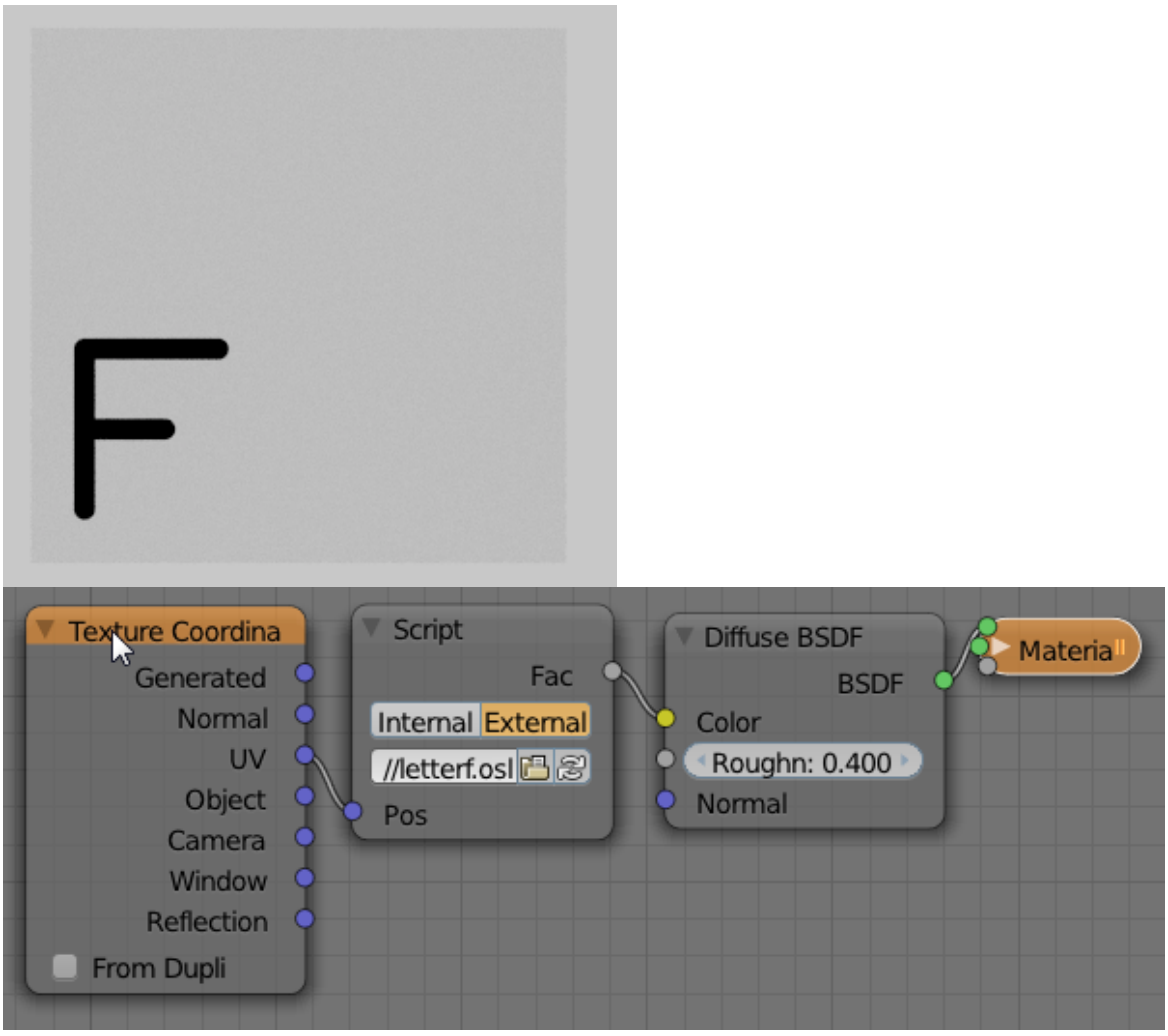
Before we can demonstrate the different symmetry operations we need a simple shader that produces an asymmetrical pattern so that we can see the effects of the operations. For this purpose we produce the letter F, albeit a rather ugly one.

Code available in Shaders/letterf.osl

```
01.  shader F(
02.    point Pos = P,
03.
04.    output float Fac = 1
05.  ){
```

```
06.    point a=point(0.1,0.1,0);
07.    point b=point(0.1,0.25,0);
08.    point c=point(0.1,0.4,0);
09.    point d=point(0.25,0.25,0);
10.    point e=point(0.35,0.4,0);
11.    float w=0.02;
12.
13.    if( (distance(a,c,Pos) < w)
14.    ||  (distance(b,d,Pos) < w)
15.    ||  (distance(c,e,Pos) < w) ){
16.      Fac = 0;
17.    }
18.  }
```

A sample scene with a simple uv-mapped square is shown below together with the node setup.



Vertical mirror

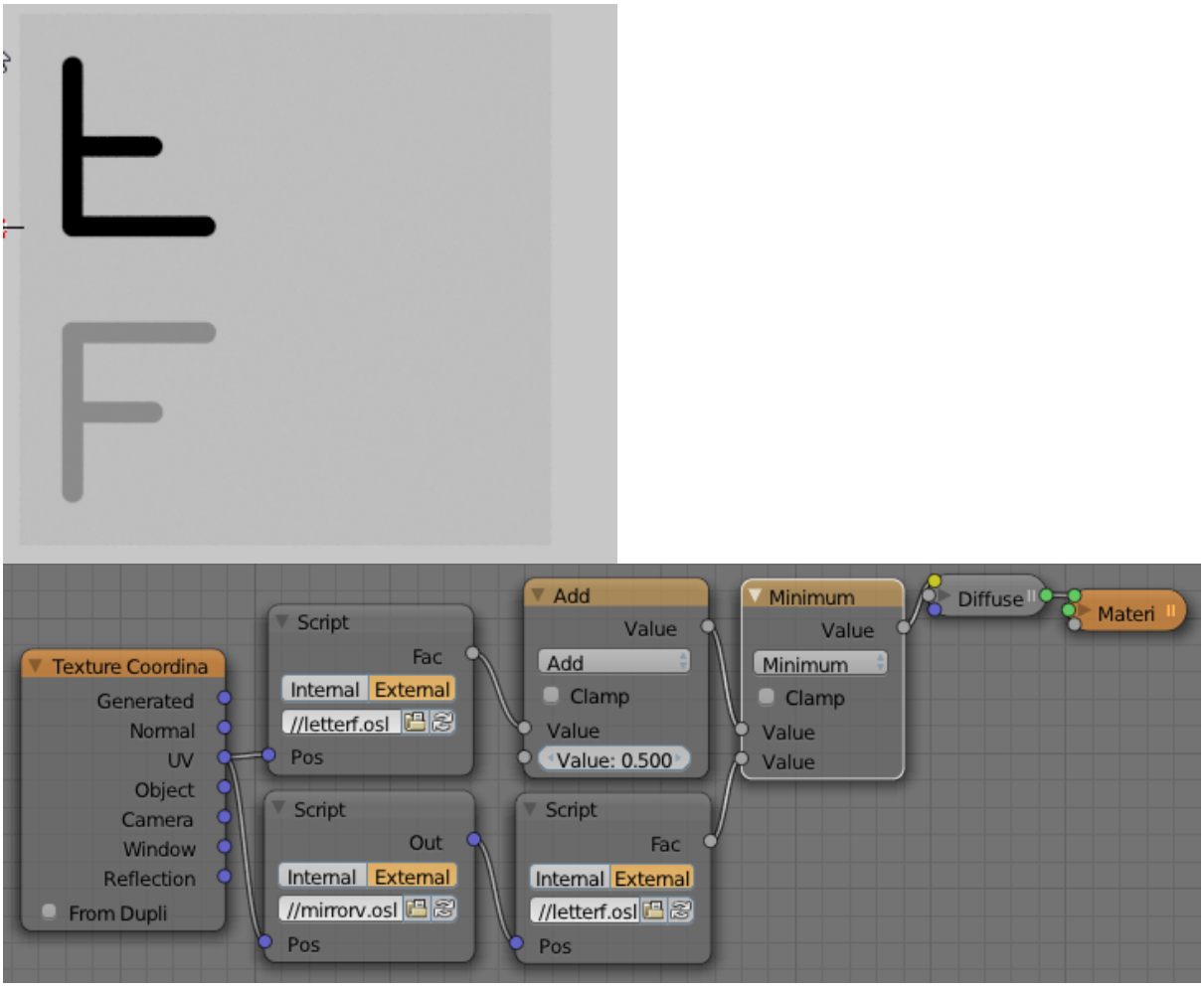
Because we assume the y coordinate to be in the range [0,1] the mirror operation about the line y = 0.5 reduces to subtracting y from 1. For example a y coordinate of 0.8 becomes 0.2.

Code available in Shaders/mirrorv.osl

```
01.  point mirror_v(point p){
02.    return point(p[0],1-
03.  }
04.
05.  shader MirrorV(
```

```
06.     point Pos = P,
07.
08.     output point Out = P
09.   ){
10.     Out = mirror_v(Pos);
11.   }
```

The image below shows the result. The node setup shows both how to use the node and a way to combine the results, where we have toned down the original letter F to a shade of gray.

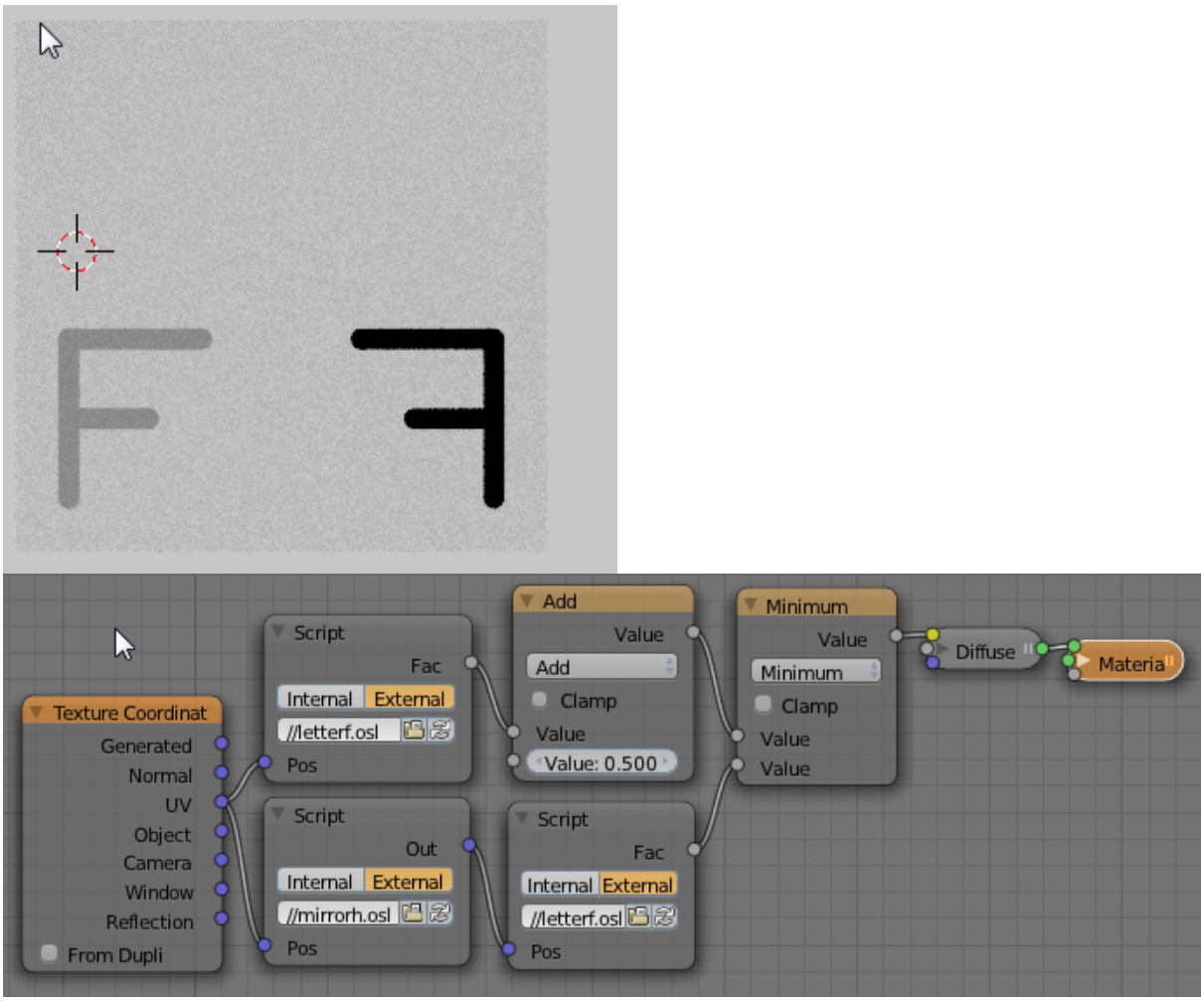


Horizontal mirror

A horizontal mirror operation about the line x = 0.5 is of course completely analogous to vertical mirroring.

Code available in Shaders/mirrorh.osl

```
01.  point mirror_h(point p){
02.    return point(1-p[0],p[1],p[2]);
03.  }
04.
05.  shader MirrorH(
06.    point Pos = P,
07.
08.    output point Out = P
09.  ){
10.    Out = mirror_h(Pos);
11.  }
```



Mirror about x = y

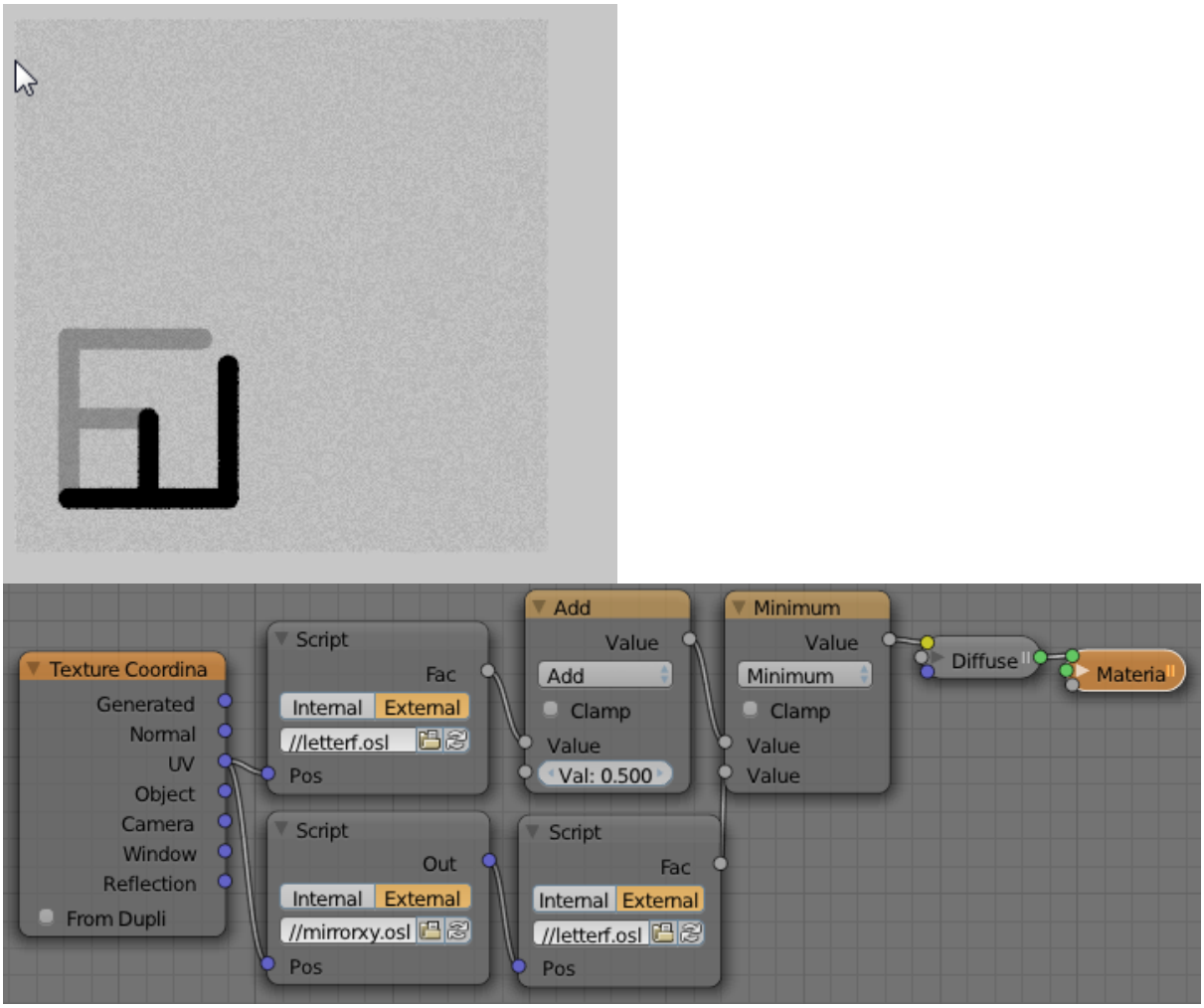
Mirroring about the diagonal that goes from 0,0 to 1,1 is a straightforward swap of the x and y coordinates.

Code available in Shaders/mirrorxy.osl

```
01.  point mirror_xy(point p){
02.    return point(p[1],p[0],p[2]);
03.  }
04.
05.  shader mirrorXY(
06.    point Pos = P,
07.
```

```
08.     output point Out = P
09.   ){
10.     Out = mirror_xy(Pos);
11.   }
```

The result and node setup are shown below.



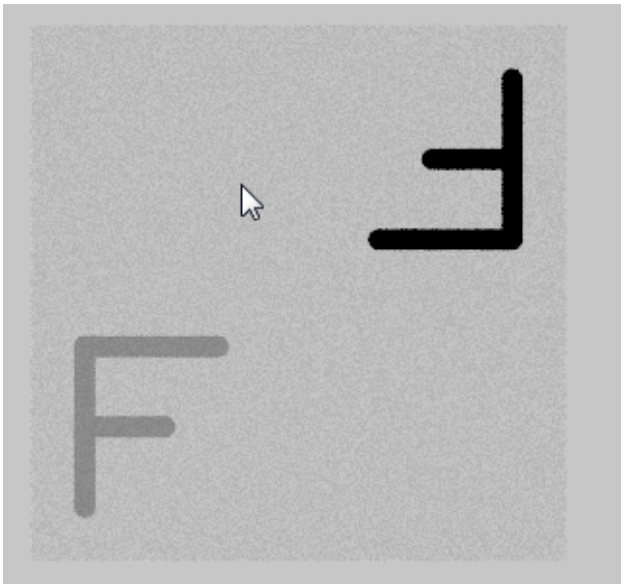
Inversion (rotation by 180°)

Inversion, or mirroring about a point instead of a line, is equivalent to rotating about that point by 180°. Another way to look at this operation is by combining a horizontal mirror operation with a vertical mirror operation. The implementation combines both actions in a single function.

Code available in Shaders/rotate180.osl

```
01.  point rotate_180(point p){
02.    return point(1-p[0], 1-p[1], p[2]);
03.  }
04.
05.  shader rotate180(
06.    point Pos = P,
07.
08.    output point Out = P
09.  ){
10.    Out = rotate_180(Pos);
11.  }
```

The result is shown below (the node setup is similar to the previous examples and not shown here).



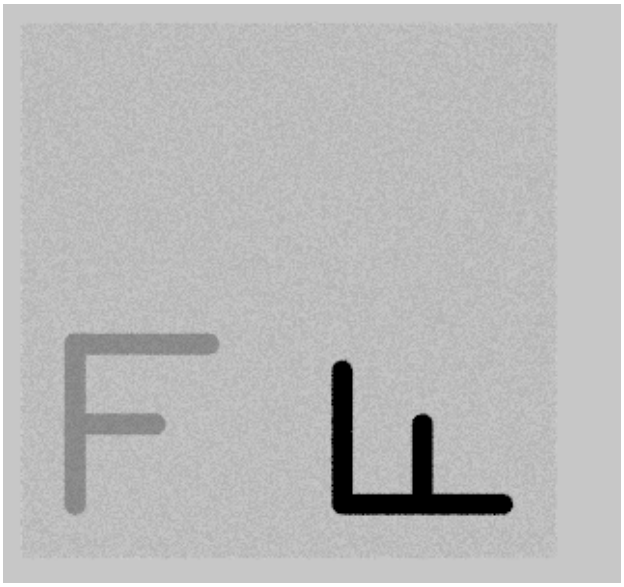
Rotation by 90°

Rotation by 90° counter clockwise around the point (0.5, 0.5) is easiest to understand by treating it as a rotation around the origin. This is accomplished by first translating by (-0.5, -0.5), then performing the rotation and finally translating back again by (0.5, 0.5).

Code available in Shaders/rotate90.osl

```
01.  point rotate_90(point p){
02.    vector v = vector(0.5, 0.5, 0);
03.    point q = p - v;
04.    q=point(q[1],-q[0],0);
05.    return q+v;
06.  }
07.
08.  shader rotate90(
09.    point Pos = P,
10.
11.    output point Out = P
12.  ){
13.    Out = rotate_90(Pos);
14.  }
```

An example is shown below. Note that `Shaders/symmetry.h` contains a function `rotate_270()` as well that might be used to perform a clockwise rotation.



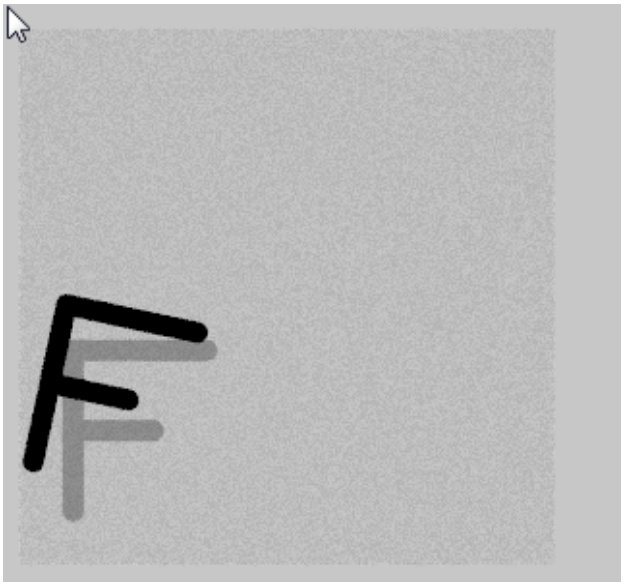
Rotation around an arbitrary center

With some trigonometry it is possible to work out a rotation around an arbitrary center by an arbitrary angle but there is no need to reinvent the wheel. OSL has a built-in function `rotate()` that can do all the heavy lifting for us.

Code available in Shaders/rotatepointangle.osl

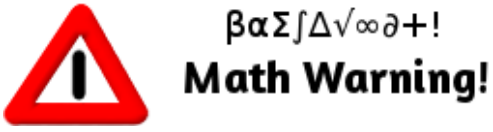
```
01.  point rotate_point_angle(
02.              point p,
03.              point q,
04.              float angle
05.  ){
06.    vector z = vector(0, 0, 1);
07.    return rotate(p, radians(angle), q, q + z);
08.  }
09.
10.  shader rotatePointAngle(
11.    point Pos = P,
12.    float angle = 0,
13.
14.    output point Out = P
15.  ){
16.    Out = rotate_point_angle(Pos, point(0.5, 0.5, 0), angle);
17.  }
```

An example for a 12° clockwise rotation is shown below.

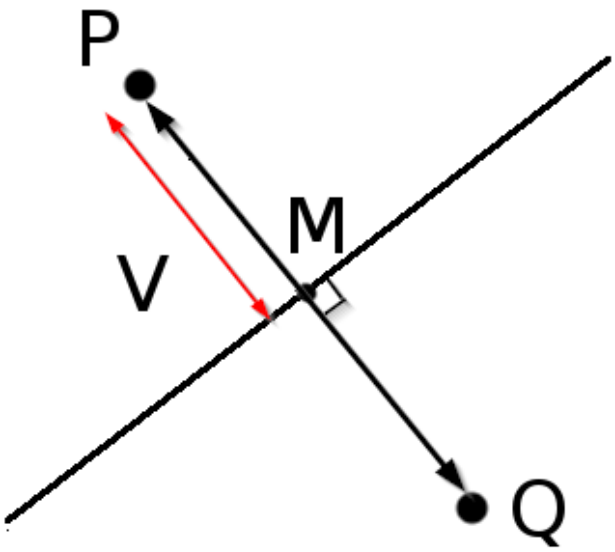


Mirror about an arbitrary line

The final operation we discuss in this section is mirroring about an arbitrary line. This is not so simple and unfortunately we have to do most of the work ourselves.



If you have a line with a slope b and an offset a and you want to mirror point P(x,y), the idea is to extrapolate the line from P to the intersection M by the same length to get Q. A diagram illustrating this concept is shown below.



The straight line perpendicular to the line we are mirroring about passes through the known point P that we want to mirror. This means we have enough information to determine the intersection and slope of this perpendicular line.

Now that we have two lines that cross at point M we can calculate the x and y coordinates of this point and calculate the mirrored point Q. The vector from P to M is V = M - P, so Q = P + 2V = 2M - P.

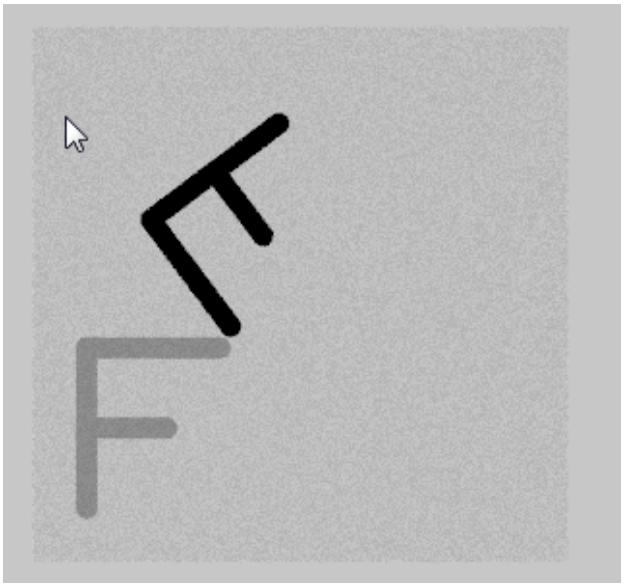
The function implementing this construct is given below, along with a sample shader.

Code available in Shaders/mirrorline.osl

```
01.  point mirror_line(point p,
02.                    float a,
03.                    float b){
04.    float Mx = abs(b)>1e-7
05.             ? (b*p[1]+p[0]-a*b)
06.              /(b*b + 1)
07.             : p[0];
08.    float My = a + b*Mx;
09.    return point(2*Mx-p[0],
```

```
10.                 2*My-p[1],
11.                 0);
12.  }
13.
14.  shader mirrorLine(
15.    point Pos = P,
16.    float a = 0,
17.    float b = 1,
18.
19.    output point Out = P
20.  ){
21.    Out = mirror_line(Pos, a, b);
22.  }
```

The code that implements the intersection algorithm is obtained by solving the various equations for the x and y coordinates of M. See the Additional reading section for a pointer on the details. We do check for the special case of b = 0 (mirroring about a horizontal line) to prevent a division by zero (line 4). Mirroring about a horizontal line implies that the x-coordinate stays the same. The y coordinate My of the intersection is derived by plugging Mx into the equation of the mirror line. The point returned is then constructed from the interpolated coordinates.



Note that all the functions shown in the shader examples in this section are also available in a separate include file `Shaders/symmetry.h` and that include file will we used in some of the shaders we encounter later on in this book.

Additional reading

Mirroring about an arbitrary line is mostly about finding the shortest distance to that line and then extending it to to opposite side. The following two articles have a lot of information on different methods, but both are quite math heavy:

An article on Stack Overflow. (Stackoverflow is a treasure trove of answers to programming problems and you can pose about any question there if you can't figure out the answer yourself.

More rigorous but also even more math heavy answers can be found on Mathworld.

Victorian era cast iron covers

Focus areas

- Symmetry
- Copying real life patterns

When I was drafting the outlines of this book I had the pleasure of visiting Ireland's capital Dublin. While staying there I had the opportunity to visit two marvelous museums both part of the National Museum of Ireland, the archeology museum and the natural history museum.

Both museums have a spectacular collection but for a lover of Victorian era artifacts there is even more to enjoy as both museums are housed in splendid buildings specifically built in the second half of the 19th century to display these collections. And in the Victorian era there was time, money and appreciation to foster exquisite craftsmanship.

Both buildings have beautiful woodwork even in the doors to the loo! And there are vivid mosaics on the floors and stairs of smooth marble but what struck me most was that even the cast iron covers of the recesses in the floor that contain pipes for the central heating are beautifully made and a joy to look at. An example is shown in the picture.

This kind of craftsmanship is a challenge to replicate in a shader and that is just what we will do in this section.

Symmetry

If you want to replicate a fairly complicated pattern it pays to look at the symmetries. The first thing we see is that the top part is identical to the lower part except for a 180 rotation.

The circular designs are identical and have fourfold rotational symmetry (each quadrant is identical to its neighbor when turned by 90 degrees).

The rosettes are identical as well and besides their fourfold rotational symmetry exhibit an additional symmetry: each quadrant consist of two halves that are mirrored about a diagonal line.

What's left are the trimmings at the top and the small lily-like ornaments that connect the edges to the central pattern. All these motifs together look like they can be approximated with circular shapes.

When we look at the code for the shader later we will follow this breakdown of symmetries exactly and will make use of the hard work in the previous section to put all those symmetry operations to work.

Code

Even while reusing the symmetry code developed in the previous section a pattern like this still needs a substantial amount of code but for clarity each design feature (the lily, the rosette, etc.) is encapsulated in a separate function. This allows for easy tweaking of individual features without affecting anything else. In the breakdown below we split the code into several parts to keep the discussion focused but the shader in the code bundle is a single file, except for the `symmetry.h` header.

Code available in Shaders/castironcover.osl

```
#include "symmetry.h"

int incircle(float x, float y, float r, float cx, float cy){
  float rx = x - cx;
  float ry = y - cy;
  return rx*rx+ry*ry < r*r;
}

... code for features ...

shader castironcover(
  point Pos = P,
  float Scale = 1,

  output float Fac = 0
){
  point p = mod(Pos*Scale, 1.0);

  if(p[1] > 0.5){
    p = rotate_180(p);
  }
  p *= 2;
  if( p[1]<1 ) {
    lily(mod(p, 1.0), Fac);
  }
  if( Fac == 0 ){
    rosette(mod(p, 1.0), Fac);
  }
  if( Fac == 0 ){
    dottedband(mod(p, 1.0), Fac);
  }
  if( Fac == 0){
    if(p[1]>0.5){
      rosette(mod(p-0.5, 1.0),
              Fac);
    }
  }
```

```
38.     if( Fac == 0 ){
39.       dottededge(p, Fac);
40.     }
41.
42.   }
```

The code begins with the inclusion of the `symmetry.h` header and the definition of a utility function `incircle()`. This function checks whether the `x` and `y` coordinates lie within the radius `r` of the a circle with center coordinates `cx` and `cy`.

The shader itself is straightforward: we scale the input coordinates and map them to the range [0,1] so the user can choose as many repetitions of the pattern as needed (line 17). We the check if the y coordinate is larger than 0.5 and if so rotate the coordinates by 180 degrees. The resulting coordinates are then multiplied by two because the separate patterns appear both on the left and right side and working with whole numbers is more convenient (although that's a matter of taste of course). The quadrant we are looking at is now basically restricted to:

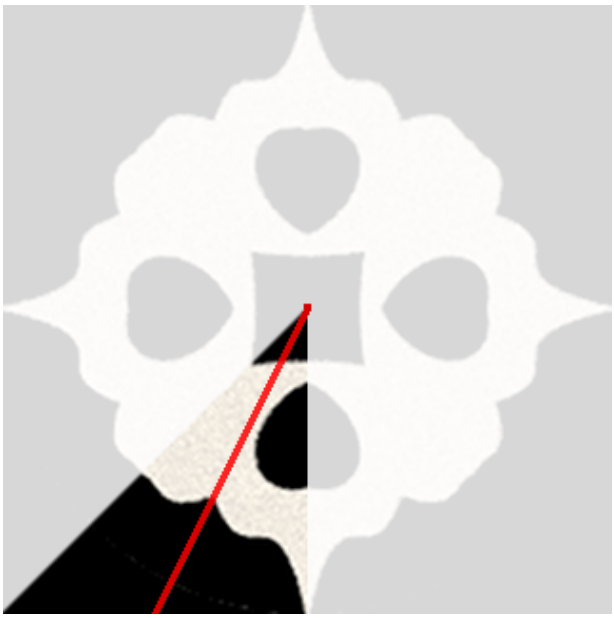
Next we check if we are in a region where a lily pattern might appear and if so call the `lily()` function to determine the details. The value that will be

returned represents the height of the pattern and will be zero outside the area covered by the lily. Only if this return value is zero do we check if we are in the rosette pattern by calling the `rosette()` function and this recipe is repeated for the other patterns. This way we quit the shader as soon as we have determined we are inside some pattern which might help the performance of the shader because we cannot be in two patterns at the same time so it us not sensible to do unnecessary and possibly lengthy calculations.

```
01.  void rosette(point Pos,
02.                output float Fac){
03.    Fac = 1;
04.
05.    point p = Pos;
06.    if(p[0] > 0.5){
07.      p = mirror_h(p);
08.    }
09.    if(p[1] > 0.5){
10.      p = mirror_v(p);
11.    }
12.    if(p[1] > p[0]  ){
13.      p = mirror_xy(p);
14.    }
15.
16.    float a=0.3;
17.    float b=135;
18.    float w=0.12;
19.
20.    if(  p[1]>w
21.      && !incircle(p[0], p[1],
22.                   0.5-w/2,
23.                   0.5, 0.0) )
24.    {
25.      Fac = 0;
26.    } else if(incircle(p[0],
27.                 p[1]*(1+0.5-p[0]),
28.                 w/2,
```

```
29.                 0.5,0.5-1.2*w))
30.     {
31.       Fac = 0;
32.     } else {
33.       if( p[1]+0.5 > p[0]*2)
34.       {
35.         p = mirror_line(p,-0.5,2);
36.       }
37.       float x = p[0]-0.5+w/2;
38.       if ((p[1]-2*w)*a < -x*x*x*b)
39.       {
40.         Fac = 0;
41.       }
42.     }
43.   }
```

The rosette function expects to be operating on coordinates that lie in the range [0,1] and the center of the rosette is positioned at (0.5, 0.5). The rosette has eight-fold symmetry so we mirror the coordinates horizontally, vertically and about a diagonal line from lower left to top right as required (lines 6-14). So we end up checking just this small segment:



The next step is to define a few constants that determine the shape of the rosette. These are tweaked to get the shape in examples shown in this section but could be altered to change the shape of the rosette.

First we check if we are in the area of the central hole (line 20). If not we check if we are in the range of the heart shaped hole (line 26). We use the `incircle()` function again but with morphed coordinates for this to get a heart shaped figure (or 'cardiod', see Additional reading section at the end for more information.)

If we are not in one of the holes we finally check if we are outside the rosette's s-shaped edge (line 38).

```
01.  void lily(point p,
02.             output float Fac){
03.
04.    float x = p[0];
05.    float y = p[1];
06.
07.    Fac = 0;
08.
09.    float w=0.12;
10.
11.    if (y < w/2  ) { return ; }
12.    y -= w/2;
13.    if ( x > 0.5 ) { x = 1 - x; }
14.    if ( y > 0.5 ) { return; }
15.
16.    if ( y < x - 0.05 ) { return; }
17.
18.    if ( incircle(x,y,0.1,0.15,0) )
19.    {
20.      return;
21.    }
22.    if(!incircle(x,y,0.15,0.15,0) )
23.    {
24.      if(y > 0.05*cos(x*30)+0.2-x )
```

```
25.       {
26.         return;
27.       }
28.     }
29.     Fac = 1;
30.   }
```

The lily-like motif is quite a simple pattern. It is mirrored about the vertical axis and is composed of a leaf (basically a circular band) a a tip shaped like a cosine. The diagram shows the components:

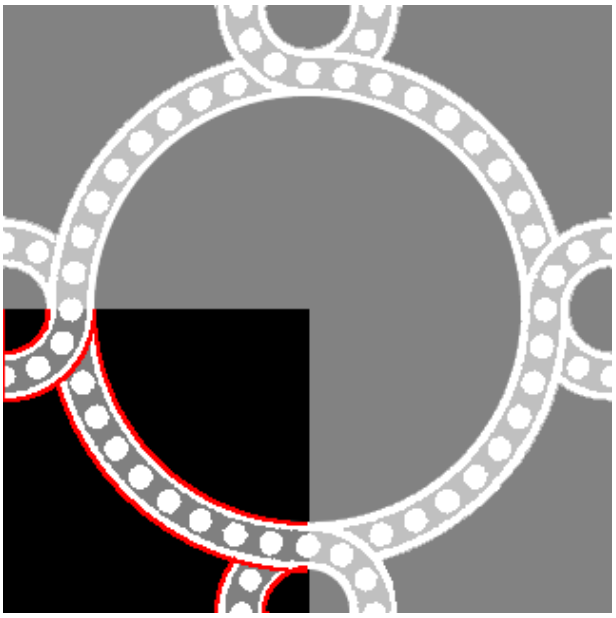
The code primarily checks if we are above the diagonal line from bottom left to top tight (line 16) and within the the boundaries of the two circles (lines 18 and 22). If not within the circles we check if we are below the cosine like curve that makes p the central lily leaf (line 24).

The dotted circular motif around the rosettes is implemented as quite a big chunk of code because we have to account for the smaller circular ornaments at the edges. The toplevel function is called `dottedband()` and is shown below

```
01.  void dottedband(point q,
02.                  output float Fac)
03.  {
04.    Fac = 0;
05.
06.    float rs = 0.15, rl = 0.43;
```

```
07.    float w=0.08;
08.
09.    if(q[1]>0.5){ q = rotate_180(q); }
10.    if(q[0]>0.5){ q = rotate_90(q);  }
11.
12.    // small up curl
13.    Fac=dottedcircle(q,0,0.5,rs,w,12);
14.    slant(q[0], rs-w, Fac);
15.    // main circle
16.    if( Fac == 0 ){
17.      Fac=dottedcircle(q, 0.5, 0.5,
18.                       rl, w, 36);
19.      slant(q[0], rl-2*w, Fac);
20.    }
21.    // small down curl
22.    if( Fac == 0 ){
23.      Fac=dottedcircle(q, 0.5, 0.0,
24.                       rs, w, 12);
25.      slant(q[1], rs-w, Fac);
26.    }
27.  }
```

We take again advantage from the fourfold symmetry (line 9) and then check if we are in the small up curl (line 13, the small curl near the left edge in the image below). If not we check if we are in the large curl (line 17) or the lower small curl (line 23). As you can see in the diagram, the large circle seems to duck under the smaller curl on the upper left just like the lower small curl seems to duck under the larger circle.



To achive this effect we alter the the height that is returned by the `dottedcircle()` function base on the size of either the x or y coordinate. This is implemented by the `slant()` function:

```
01.  void slant(float coord,
02.              float limit,
03.              output float Fac)
04.  {
05.    if( Fac > 0 && coord < limit)
06.    {
07.      Fac -= 0.2*(limit-coord)
08.                 /limit;
09.    }
10.  }
```

The `dottedcircle()` function implements the actual circular bands with small raised dots:

```
01.  float dottedcircle(point p,
02.    float cx, float cy,
03.    float r, float w, int n)
04.  {
05.    float Fac = 0;
```

```
06.
07.    float e=w/4;
08.    float x=p[0], y=p[1];
09.    if(  incircle(x,y,r  ,cx,cy)
10.      &&!incircle(x,y,r-w,cx,cy))
11.    {
12.      if(incircle(x,y,r-e,cx,cy)
13.        &&!incircle(x,y,r-w+e,cx,cy))
14.      {
15.        Fac = 0.5;
16.        for(int i=0; i<n; i++){
17.          float s,c;
18.          float angle = radians(i*360/n);
19.          float m = r - w/2;
20.          sincos(angle,s,c);
21.          if(incircle(x,y,
22.                      e,cx+m*s,cy+m*c))
23.          {
24.            Fac = 1;
25.            break;
26.          }
27.        }
28.      }else{
29.        Fac = 1;
30.      }
31.    }
32.    return Fac;
33.  }
```

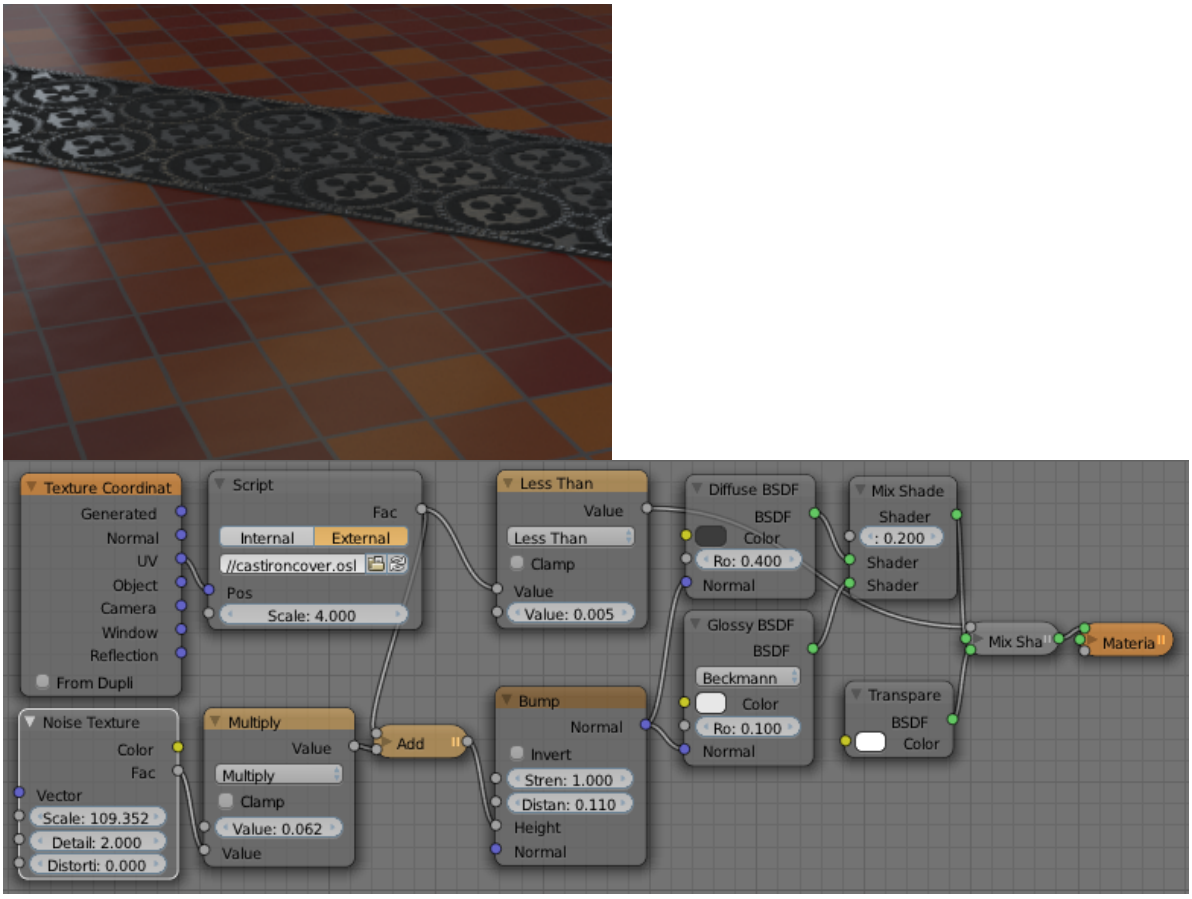
It first checks if we are within the circular band at all (line 9) and the if we are in the innermost part where the dots are located (line 12). Then for as many dots we want to check on the circle, the angle along the circle of that smaller dot is calculated (line 18) and then we determine whether we are within the small dot (line 21). The result is that `dottedcircle()` will return a value of 1 on the edges of the band and within the small dots and 0.5 for the deeper part between the dots.

The dotted band on the edge of the overall pattern is rather simple compared to the other motifs. First we check if we are within the band at all and then if we are within the raised edges. If not we check if we are within the small dots:

```
01.  void dottededge(point p,
02.                  output float Fac)
03.    Fac = 0;
04.
05.    float Ndots = 15;
06.    if(p[1]<0.08){
07.      if(p[1]<0.015 || p[1]>0.065)
08.      {
09.        Fac = 1;
10.      }else{
11.        if( incircle(
12.              mod(p[0],1/Ndots),
13.              p[1],
14.              0.02,
15.              (1/Ndots)/2,
16.              0.04) )
17.        {
18.          Fac = 1;
19.        }else{
20.          Fac = 0.5;
21.        }
22.      }
23.    }
24.  }
```

Examples

The result of all that code is shown in the image below together with the node setup used to implement the material for the cast iron over.



The node setup consists of two rows. In the upper row we feed the uv coordinates of the plane that represents our cast iron cover into the script node and then check if the `Fac` output is less than some small number. This result is used to drive a mix node. If the result is 0 a transparent shader is chosen, otherwise a mix shader. This mix shader is primarily a very dark diffuse shader mixed with some glossiness. Both diffuse shader and glossy shader are fed a normal that is calculated by the node in the lower row. These nodes combine some noise with output of our script (which will be non zero for solid parts) and use a bump node to convert this displacement to a normal.

Additional reading

The many ways to define a cardioid shape are listed in this Wikipedia article.

Ripples

Focus areas

- Displacement
- Improving performance

The shallow, rapidly expanding ripples caused by falling droplets are an excellent example of the utility of programmable shaders. Because these ripples manifest themselves mainly because of reflections they are not so simple to create from photographic sources (the source image will reflect something different than what will be reflected in our scene and automated tools have a hard time producing a displacement or normal map from an image with very visible reflections). Also, these dynamic patterns are at their best when animated and when the expanding rings overlap.

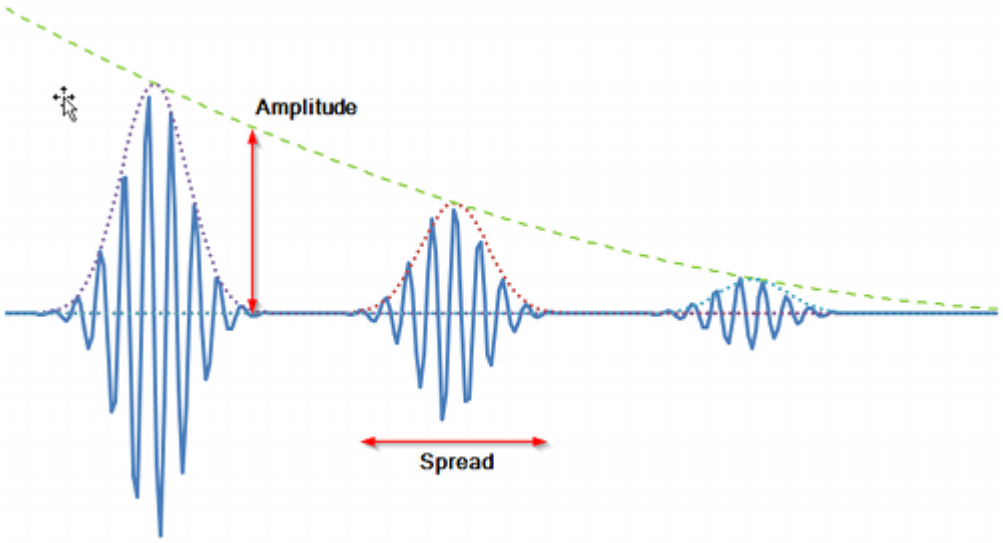
It is possible to simulate falling droplets on a body of water (for example with Blender's fluid simulator) but physically accurate simulations take a lot of time to compute and to look good they need a lot of detail which may take a lot of time to render. The solution we seek should therefore be both quick to render and somewhat better to control than a simulation.

A droplet

In the droplet shader we create here we assume that the input position is in the range [0,1] and the start and end times limit the period wherein droplets are generated. The units are arbitrary and not linked to frames. The `Time` parameter may be animated and determines the point in time that we are rendering, that is how far the ripples have expanded. It is

perfectly acceptable for `Time` to be larger than `EndTime` but for large values all the contributions of the various drops will have died away.

The remaining parameters control the shape of the wave as it expands outward as illustrated in the diagram.



The `Damping` parameter determines how well the wave keeps its form. The smaller this value the quicker the wave dies away.

Code available in Shaders/droplet.osl

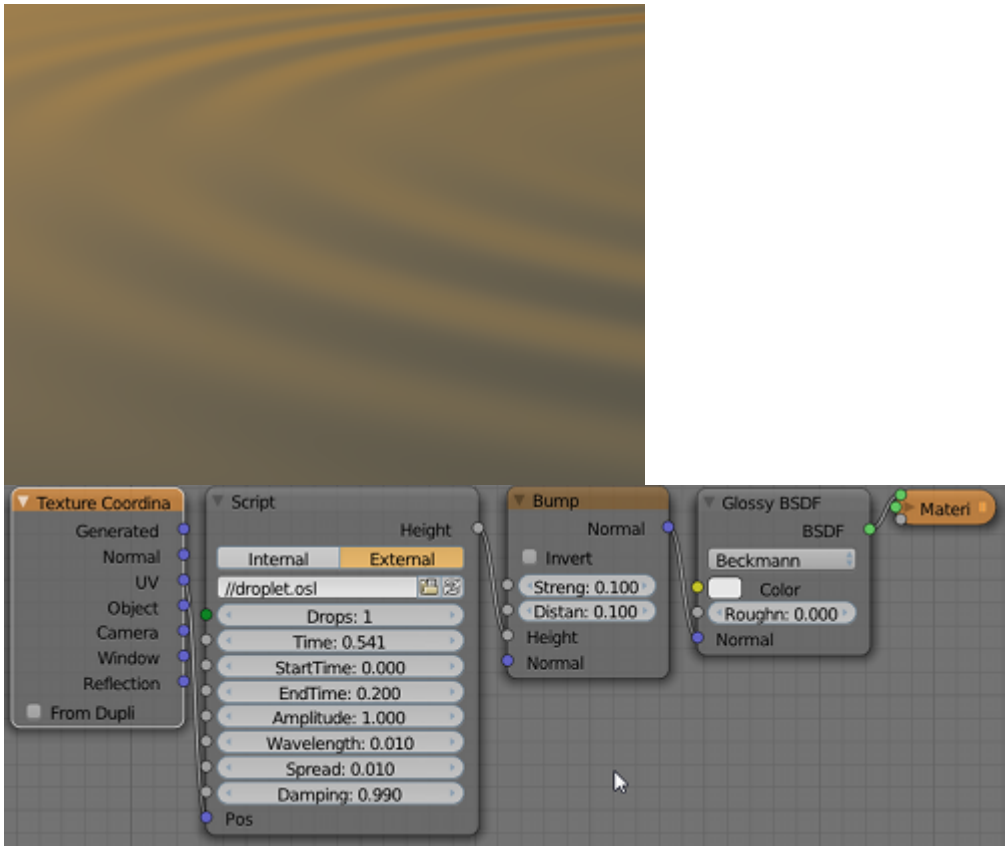
```
01.  shader droplet(
02.    point Pos = P,
03.
04.    int Drops = 1,
05.    float Time = 1,
06.    float StartTime = 0,
07.    float EndTime = 1,
08.    float Amplitude = 1,
09.    float Wavelength = 1,
10.    float Spread = 1,
11.    float Damping = 0.99,
12.
```

```
13.    output float Height = 0
14.  ){
15.    for(int i=0; i < Drops; i++){
16.      vector center = noise("cell",i,1);
17.      float start = (EndTime-StartTime)
18.                    * noise("cell",i,2);
19.      float peak = Time - start;
20.      float dc =hypot(Pos[0]-center[0],
21.                      Pos[1]-center[1]);
22.      float t = dc/Wavelength;
23.      float a = Amplitude * cos(t);
24.      a *= pow(Damping,start);
25.      a *= exp(-(dc-peak)*(dc-peak)
26.                           /Spread);
27.      Height += a;
28.    }
29.  }
```

For each drop specified we generate a unique position and a unique start time and from this start time we calculate the current position of the peak of the ripples (line 17).

The next step is to calculate the distance between the position that is being shaded and the center of the ripples. Dividing this distance by the wavelength and taking the cosine allows us to calculate the amplitude. We moderate this amplitude by damping it (line 24) and by taking into account the distance to the center peak before adding this contribution to the `Height` output parameter.

In the image shown below we used the height of the ripples to calculate the normal and used this normal to drive a simple glossy shader. Combined with a suitable sky shader thus gives a rather nice impression.



Rain

There are a couple of things we can improve upon and the most important one is performance. If we want to display the ripples from a large number of droplets we currently check all the droplets to see if their ripples contribute to the position that is being shaded. If instead of increasing the number of droplets we scale the input position and impose a grid, we can check beforehand if the current position is affected by checking just the grid cell we are in together with an appropriate number of neighboring cells.

In the previous implementation the ripples faded out increasingly slowly as they expanded without ever really dying out completely. If we want to limit the number of cells we want to check we'd better reduce the range to something less than infinite so we have to rethink our damping function. An implementation is shown below.

Code available in Shaders/droplets.osl

```
01.  shader droplets(
02.    vector Position = P,
03.    float Scale = 5,
04.
05.    int Drops = 3,
06.    float Time = 1.7,
07.    float StartTime = 0,
08.    float EndTime = 10,
09.    float Amplitude = 1,
10.    float Wavelength = .01,
11.    float Spread = .01,
12.    float Limit = 2,
13.    float Damping = 0.5,
14.
15.    output float Height = 0
16.  ){
17.    float L =max(ceil(abs(Limit)),1);
18.
19.    vector Pos = Position * Scale;
20.    vector ip = floor(Pos);
21.
22.    float h = 0;
23.    for(float dx= -L; dx<=L; dx++){
24.      for(float dy= -L; dy<=L; dy++){
25.        vector p=ip+vector(dx,dy,0);
26.        for(int i=0; i < Drops; i++){
27.          vector center =
28.               noise("cell",p,i) + p;
29.          float start=(EndTime
30.                       -StartTime)
31.               *noise("cell",p,-i-1);
32.          float peak = Time - start;
33.          float dc = hypot(
34.                 Pos[0]-center[0],Pos[1]-center[1]);
```

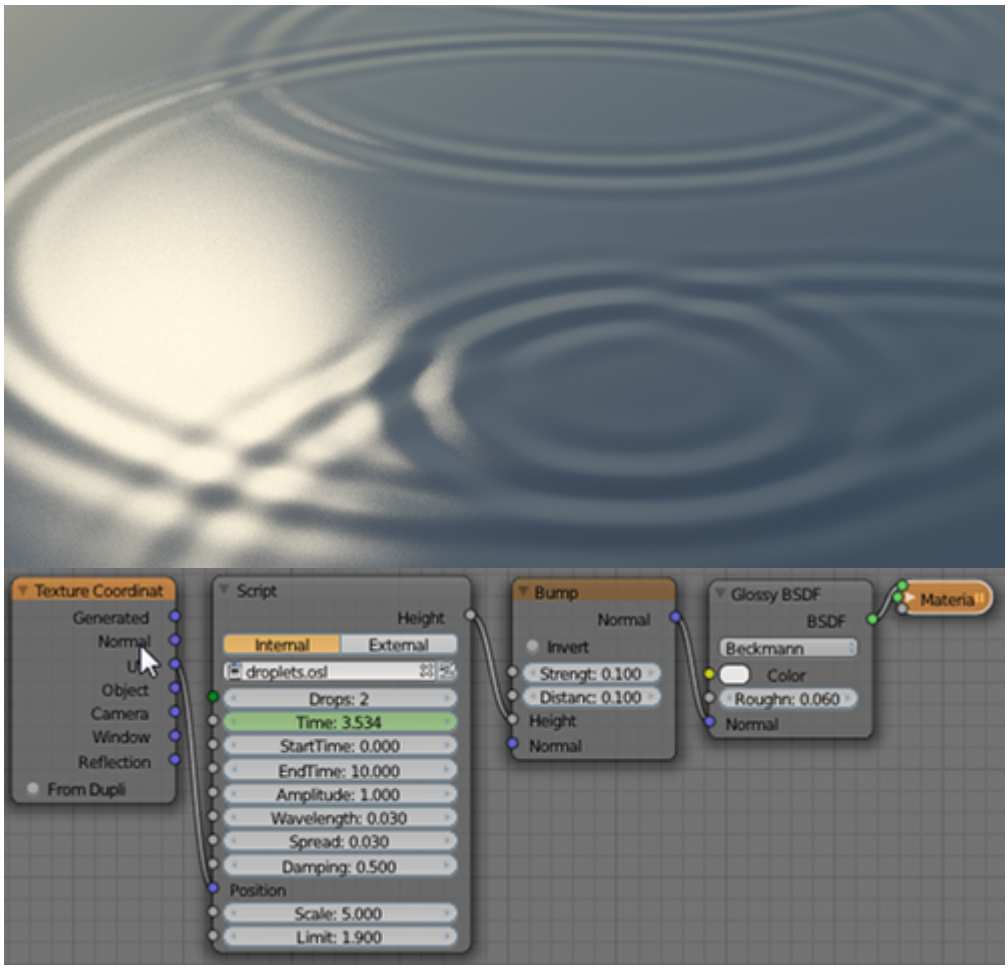
```
35.           float t =dc/Wavelength;
36.           float a =Amplitude*cos(t);
37.           float dr=(Limit-peak)
38.                    /Limit;
39.           float dm=pow(dr,Damping);
40.           dm *= exp(-(dc-peak)
41.                    *(dc-peak)
42.                    /Spread);
43.           if( dm > 0 ){
44.             h += a*dm;
45.           }
46.         }
47.       }
48.     }
49.     Height = h;
50.   }
```

The most important difference is that we now loop over the neighboring cells as well. How many cells we take into account is determined by the `Limit` parameter which we round up to the nearest positive integer. We make sure that we check at least the directly adjacent neighbors by ensuring this value is at least 1 (line 17).

The random centers and start times of the droplets are calculated for each cell and the distance is determined in the same manner as before. Because our ripple contribution will be zero if the distance is larger than the value of `Limit` we only calculate the contribution for these ripples if we are within range.

The damping of the amplitude is now dependent on the distance from the center and is reduced to zero when this distance reaches zero (line 35). The shape of this damping function is controlled by the `Damping` parameter. In fact our damping may result in a negative value for distance beyond `Limit` but we check for that and disregard any contributions if this happens (line 37).

The image below is a still from an animated sequence of droplets:



The video itself can be watched on Youtube,
http://youtu.be/Zejw9ESvfxM

Concrete shader

Focus

- Building a complex material by layering textures
- using different kinds of noise for different effects

Concrete is a versatile material used since roman times. In modern times it is ubiquitous so a good shader would be an invaluable tool for any modeler who wants to create buildings or other architectural structures.

Concrete is also a rather complex material consisting of a cement binder and one or more types of aggregate material like sand, crushed rocks or pebbles. To implement the concrete shader we take a layered approach and combine the color contribution of the cement with the color of three types of aggregate: sand, split and pebbles. Each of these aggregates may attribute to the surface normal and displacement as well.

Besides the different sizes, these aggregate components differ in shape as well: sand and split are angular while pebbles are more or less rounded. We will use different types of noise to approximate those shapes. A final touch is added by allowing the user to specify some stain colors that might be added on top of the basic concrete texture.

If we look at the way concrete is used we see that often the concrete is poured inside a mold made of wood. When this mold is finally removed some of the aggregate particles at the surface may dislodge, resulting in gaps. (It is also possible that some of the gaps are caused by air bubbles). In this shader we allow for this situation by providing gap parameters for the split and the pebbles. The gap parameters specify the percentage

of aggregate particles that falls off and leave a hole. The hole is then created by inverting the normal and the displacement, creating in fact a negative aggregate particle, but we leave the color of the cement unchanged.

To allow for additional control, the outputs are not just color, surface normal and displacement but also a set of integer values that indicate whether the surface at the position being shaded is a certain type of aggregate, for example `IsSplit`. These outputs may be used for example to select different glossy shaders for each aggregate type or to select different special shaders that you may already have, for example a quartz shader for the pebbles and a granite shader for the split. By providing this information we allow for maximum versatility without complicating the shader too much.

Code

The code for this shader is available in Shaders/concrete.osl

```
01.  point voronoi3d(point p, float t)
02.  {
03.    int xx, yy, zz, xi, yi, zi;
04.
05.    xi = (int)floor(p[0]);
06.    yi = (int)floor(p[1]);
07.    zi = (int)floor(p[2]);
08.
09.    float dbest = 1e10;
10.    point pbest = 1e10;
11.    for(xx=xi-1; xx<=xi+1; xx++){
12.      for(yy=yi-1; yy<=yi+1; yy++){
13.        for(zz=zi-1; zz<=zi+1;zz++)
14.        {
15.          vector ip =
16.              vector(xx, yy, zz);
```

```
17.           point vp = ip
18.             + noise("cell", ip, t);
19.           vector dp = p - vp;
20.           float  d  = dot(dp,dp);
21.           if (d < dbest) {
22.              dbest = d;
23.              pbest = vp;
24.           }
25.         }
26.       }
27.     }
28.     return pbest;
29.   }
30.
31.   float voronoi(point p, float t,
32.     output point pos,
33.     output float v1,
34.     output float v2)
35.   {
36.     pos = voronoi3d(p, t);
37.     v1 = noise("cell",pos,t-713);
38.     v2 = noise("cell",pos,t+317);
39.     return noise("cell",pos,t+173);
40.   }
```

Besides the shader itself we provide two auxiliary functions: `voronoi3d()` which returns the closest point from a random collection of nearby points and `voronoi()` which returns a float value in the range [0:1] unique for each voronoi cell, together with the center position of the cell and two other random, uncorrelated float values that are the same throughout the cell. These additional float values will be used to define the amount of split that is dislodged from the matrix and the amount the splits sticks out, i.e. the height of the displacement.

You can read more on voronoi functions in the chapter on noise.

The code for this shader is available in Shaders/concrete.osl

```
01.  shader concrete(
02.    point Position   = P,
03.    float Scale      = 1,
04.
05.    float Sand       = 0.9,
06.    float SandSize   = 0.001,
07.    float SandDensity= 10,
08.    float GrainHigh  = 0.5,
09.    float GrainLow   = 0.0,
10.
11.    float Split      = 0.01,
12.    float SplitSize  = 0.02,
13.    float SplitGaps  = 0.001,
14.
15.    float Pebbles    = 0.005,
16.    float PebbleSize = 0.1,
17.    float PebbleGaps = 0.001,
18.
19.    float Stains     = 0.02,
20.    float StainSize  = 1,
21.    float StainMix   = 0.2,
22.
23.    color Cement1 = 0.9,
24.    color Cement2 = 0.75,
25.    color Sand1   = 0.35,
26.    color Sand2   = 0.42,
27.    color Split1  = 0.7,
28.    color Split2  =
29.                 color(0.7,0.6,0.5),
30.    color Pebble1 = 0.8,
31.    color Pebble2 =
32.                 color(0.8,0.7,0.6),
33.    color Stain1  = 0.3,
34.    color Stain2  =
```

```
35.                color(0.3,0.4,0.3),
36.
37.    output color Col     = 0.5,
38.    output normal Normal = 0,
39.    output float Disp    = 0.0,
40.    output int IsCement  = 1,
41.    output int IsPebble  = 0,
42.    output int IsSplit   = 0,
43.    output int IsSand    = 0
44.  ){
45.    point p = Position * Scale;
46.    Col = mix(Cement1,Cement2,
47.              noise("uperlin",p,0));
48.
49.    float pebblenoise = noise(
50.                   "uperlin",
51.                   p/PebbleSize,1);
52.    if(Pebbles > pebblenoise){
53.      Disp = sqrt(
54.               pebblenoise-Pebbles);
55.      if(PebbleGaps < noise(
56.            "cell",p/PebbleSize,2))
57.      {
58.        IsCement = 0;
59.        IsPebble = 1;
60.        Col = mix(Pebble1,Pebble2,
61.                pow(noise(
62.                     "uperlin",
63.                     p/PebbleSize,
64.                     3),    2));
65.      }else{
66.        Disp = -Disp;
67.      }
68.    }else{
69.        // angular split
70.        float peakheight, gapvalue;
```

```
        point splitpos;
        float splitnoise =
           voronoi(p/SplitSize,4,
           splitpos, peakheight,
           gapvalue);
        if(Split > splitnoise){
          Disp = peakheight
            *(1-distance(p,
                        splitpos));
          if(SplitGaps < gapvalue){
              IsCement = 0;
              IsSplit = 1;
              Col = mix(
                    Split1,Split2,
                    pow(
                       noise(
                       "uperlin",
                       p/SplitSize,
                       6),
                    2));
          }else{
              Disp = -Disp;
          }
        }else{
          // sandgrains
          int sandvisible =
               Sand > noise(
                        "uperlin",
                        p/SandSize,
                         7);
           if(sandvisible){
             float grain=noise(
               "gabor",p,8,
               "bandwidth",4,
               "anisotropic",2,
               "direction",
```

```
              vector(
                SandDensity,0,0));
            if(grain>GrainLow
            && grain < GrainHigh){    IsCement = 0;
                IsSand = 1;
                Col = mix(
                   Sand1,Sand2,
                   noise(
                   "uperlin",p,9));
                Disp  = noise(
                   "cell",
                   (p*SandDensity)
                   *0.001*grain,
                   10);
                Normal= noise(
                   "cell",
                   (p*SandDensity)
                   *0.001*grain,
                   11);
                Normal[2]=1;
                Normal =
                 normalize(Normal);
            }
          }
       }
    }
    // overall color stain
    float stainnoise =
       noise("uperlin",
       p/StainSize,12);
    if(Stains > stainnoise){
      color StainCol =
              mix(Stain1,Stain2,
              noise("uperlin",p,
                    13));
      Col = mix(Col, StainCol,
```

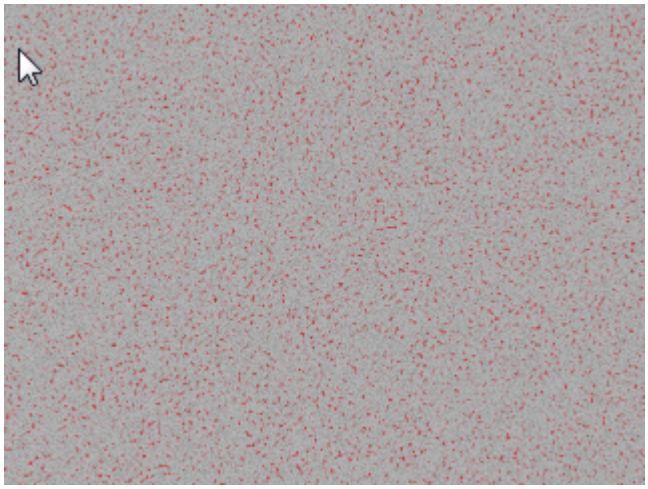
```
143.                 StainMix);
144.     }
145.   }
```

The code for the shader itself is dominated by the long list of input and output parameters but the body is straightforward. The base color for the cement is calculated and then we check from large to small (pebble, split, sand) whether we should shade something special. Each type of aggregate uses a different noise function to model the particles. Pebbles are modeled with Perlin noise for a rounded outline while split is sharp, something we can emulate with voronoi noise.

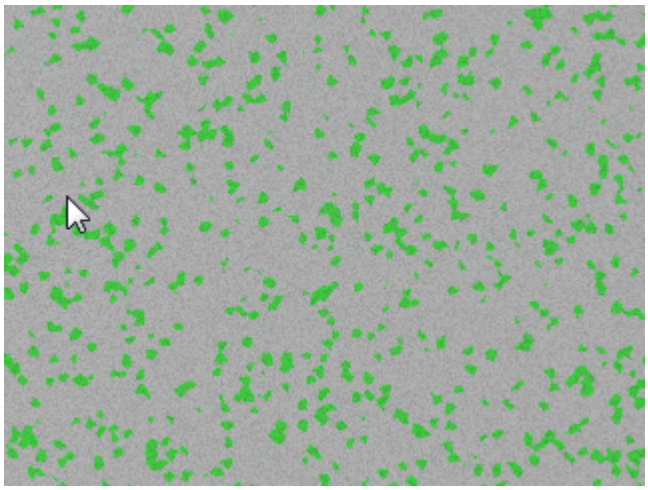
Sand uses two noise sources, Perlin noise to determine the distribution of the patches where the sand grains are visible and Gabor noise to create the grains. The final step is adding some stains on top of the various components. Note that all calls to the noise function are passed an extra value besides the position that is different for each call. This ensures that the various noise values are not correlated.

Examples

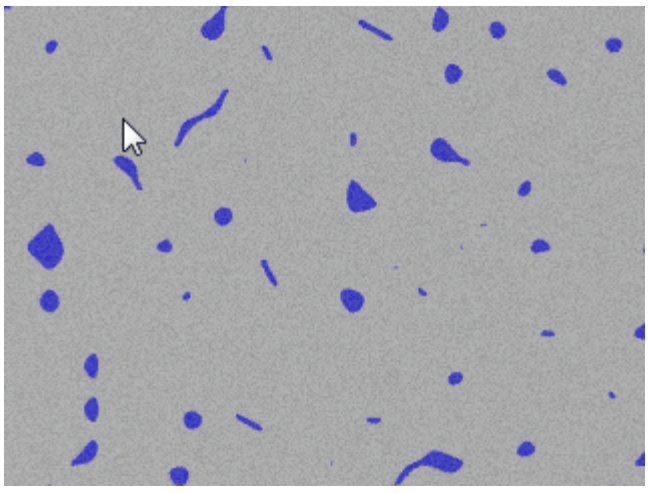
To show the shapes and distributions of the different components I rendered a few examples with fully saturated colors for the different components and without any bumps. The first shows the distribution of the sand grains (in red):



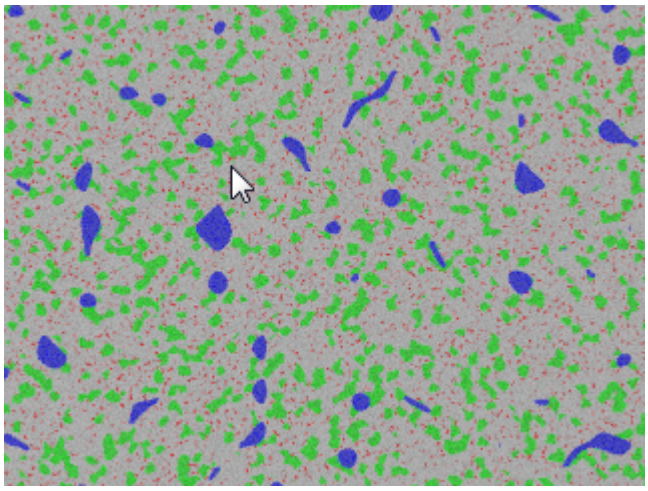
The split (in green) might look like this:



And pebbles (in blue) are more rounded

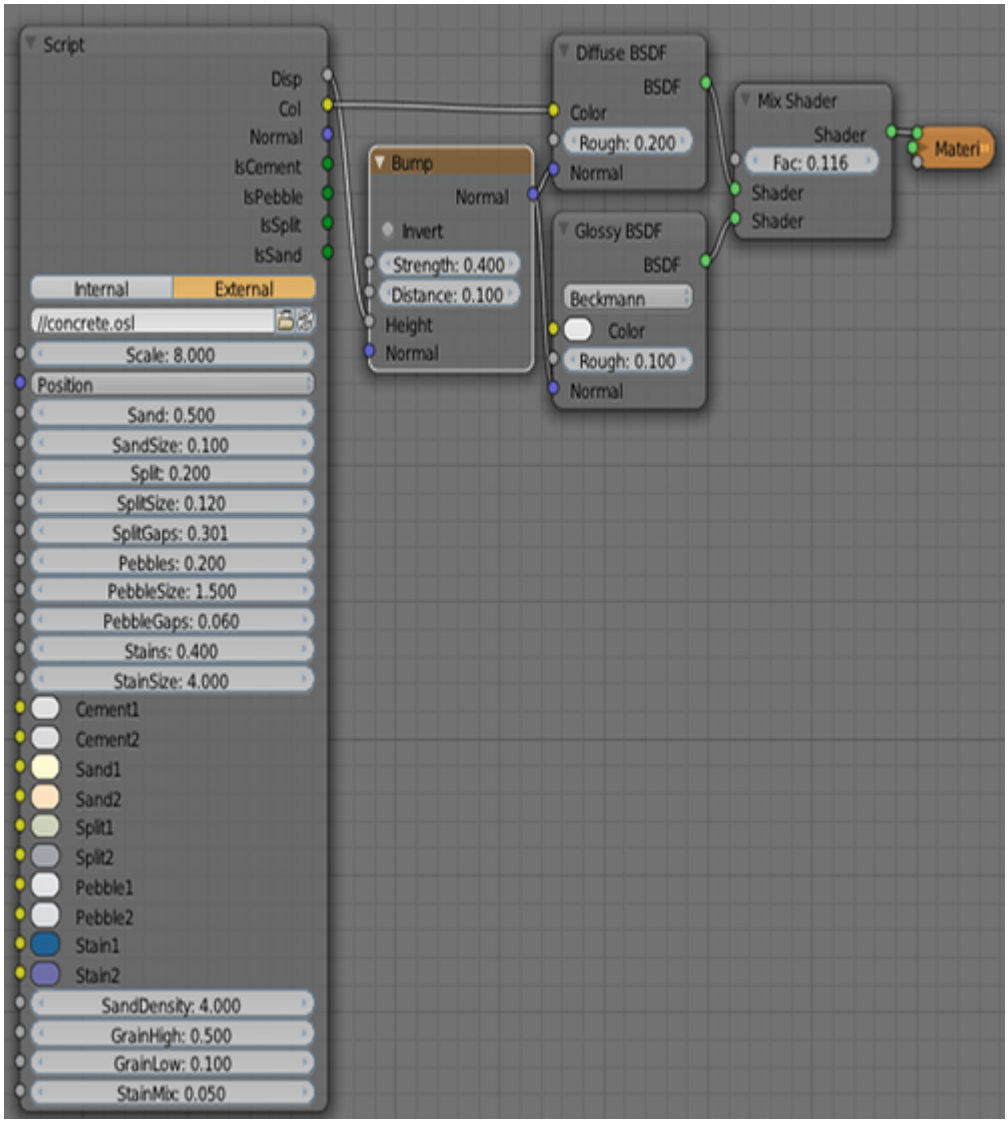


The combined distribution is shown below:



The final example with realistic colors might look something like this, although most color variations are probably to subtle to show up clearly in a low resolution illustration:

The node setup for the previous example looks like this:



Additional reading

Gabor noise is a rather new type of noise that is not yet seen much in computer graphics. There is no node for it yet in Blender's node editor but as you could see, it is available from OSL. You might want to read some more about it on my blog. The first article implements a simple noise texture that exposes most parameters of Gabor noise to the end user, while the second article explores the effects of the parameters. The original article on Gabor noise by Ares Lagae et al. might be a good read as well although it is quite technical.

Citrus peel

Focus areas

- function overloading
- seamless textures

Citrus fruit like orange, lemon, lime and grapefruit all share a similar outer skin or 'zest'. This zest is fairly uniform in color and rather smooth and shiny once the fruit has ripened but it also has a large number of small vesicles embedded in it that contain the fragrant oils that give the citrus it characteristic smell.

On closer inspection we can see that these vesicles appear slightly darker and are to a certain degree transparent. They are also slightly raised above the surrounding peel.

The distribution of these vesicles appears random yet fairly uniform. There seem to be no regions where these vesicles are bunched up or very sparse except maybe near the point where the stem was attached.

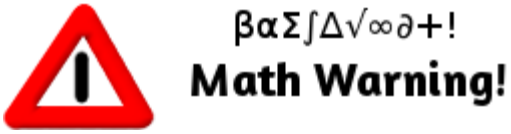
Not so random numbers

Our first objective is to find a way to distribute points across a surface in a slightly more regular fashion than the so called uniform distribution we get when calling `noise()`.

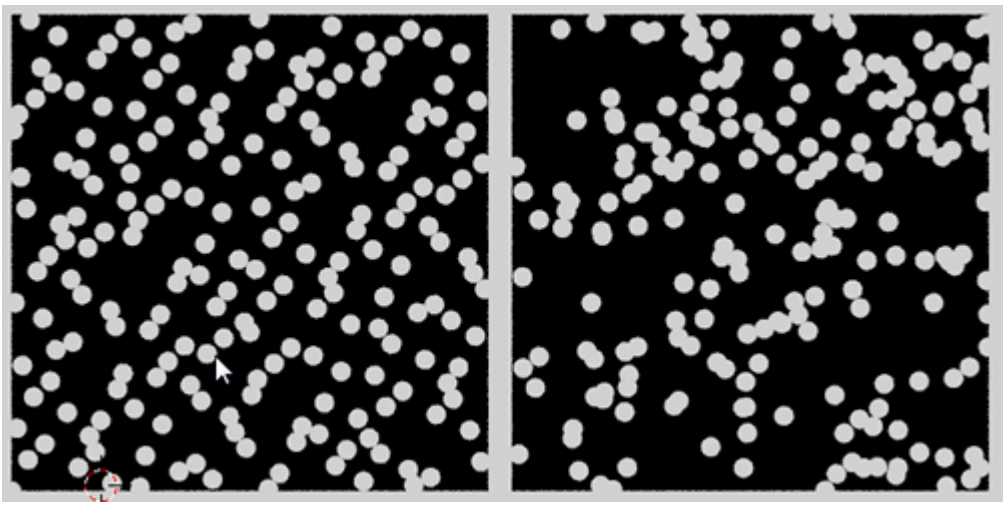
One way to approach this would be to simply check if a generated point is too close to the ones already generated and generate a new one if necessary. This would be incredibly expensive however, especially if the density of the points was high. You have to keep in mind that we might be doing this for thousands or even millions of samples so we rather

would like a function that generated a suitable distribution without the need to check and reject generated points.

Fortunately there do exist distributions with exactly those characteristics. The one we will be using is called the Halton sequence and the code is shown below.



The Halton sequence is an example of a family of distributions that is sometimes referred to as *quasi* random numbers. Unlike the *pseudo* random numbers often used in programming, these numbers are random enough for many purposes yet more evenly distributed, which is one reason why renderers (including Blender) use these kind of distributions when calculating multiple samples per pixel for example. In this context such a collection of points is often referred to as being 'jittered'. The difference is shown in the image below, which has the Halton distribution on the left.



The following code is available in Shaders/halton.h

```
01.  float haltonsequence(
02.            int index,
03.            int base){
04.      float result=0.0;
05.      float f=1.0/base;
06.      int ri=index;
07.      while(ri>0){
08.          result += f*(ri%base);
09.          ri=int(ri/base);
10.          f/=base;
11.      }
12.      return result;
13.  }
14.
15.  vector haltonsequence(int index){
16.      return vector(
17.              haltonsequence(index,2),
18.              haltonsequence(index,3),
19.              haltonsequence(index,5)
20.             );
21.  }
```

The code to generate a Halton sequence is simple enough and implemented in the code as the function `haltonsequence()` which takes an index and a base. The index is the position in the Halton sequence so an index of 9 gives the 9th number of a Halton sequence. The base selects which Halton sequence. The are infinitely many of them and details may be found in the Wikipedia article mentioned at the end. Most bases that are prime numbers will do.

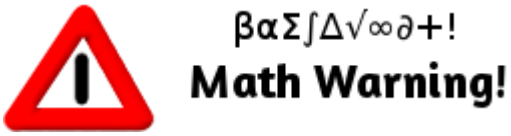
The `haltonsequence()` function returns a floating point number in the range [0,1] but it would be nice if we could design a function that could return a point-like type as well depending on the context. As we have seen earlier when calling built-in functions OSL allows overloading of functions so here we have defined two variants of the function

`haltonsequence()`, one returning a float and another one returning a vector. The OSL compiler will take care of calling the correct one depending on which type of parameters we call it with and to what type of variable we assign the result. Note that the code is available as an include file so it is possible to reuse it in other shaders.

Peeling an orange

When we want to apply a texture to a surface we often opt for creating a uv-map but a uv-map always has seams. It sometimes is possible to stitch those seams and if the texture we're applying was seamless itself these seams wouldn't be visible. However, applying a texture in a seamless manner often results in stretching, especially on more or less spherical objects like our citrus fruit.

Fortunately when working with programmable shaders we can make use of the fact that we know something about the shape. In this case we know that our citrus fruit is (approximately) shaped like a sphere. So if we want to distribute points over this surface without having to worry about seams or regions with an uneven distribution we can use known methods to create such a uniform distribution.



Generating points on a sphere in a uniform manner is not as simple as it looks. Picking a random angle around the polar axis (think of the longitude on a map of the earth) and another random angle relative to the equator (the latitude) would result in more points around the polar regions. (Think of the 'squares' between latitude and longitude lines, these sides span the same angle but toward the poles the areas of these squares are smaller. This means that a uniform sampling of angles leads to more samples per unit area near the poles.

Several clever methods have been devised to correct this and quite a few are illustrated in the article on Sphere Point Picking listed at the end of this section. The method we implement here is the projection of a distribution on a circumscribed cylinder back onto the sphere. It is not necessarily the fastest method but the code is simple.

code available as Shaders/sphericaldots.osl

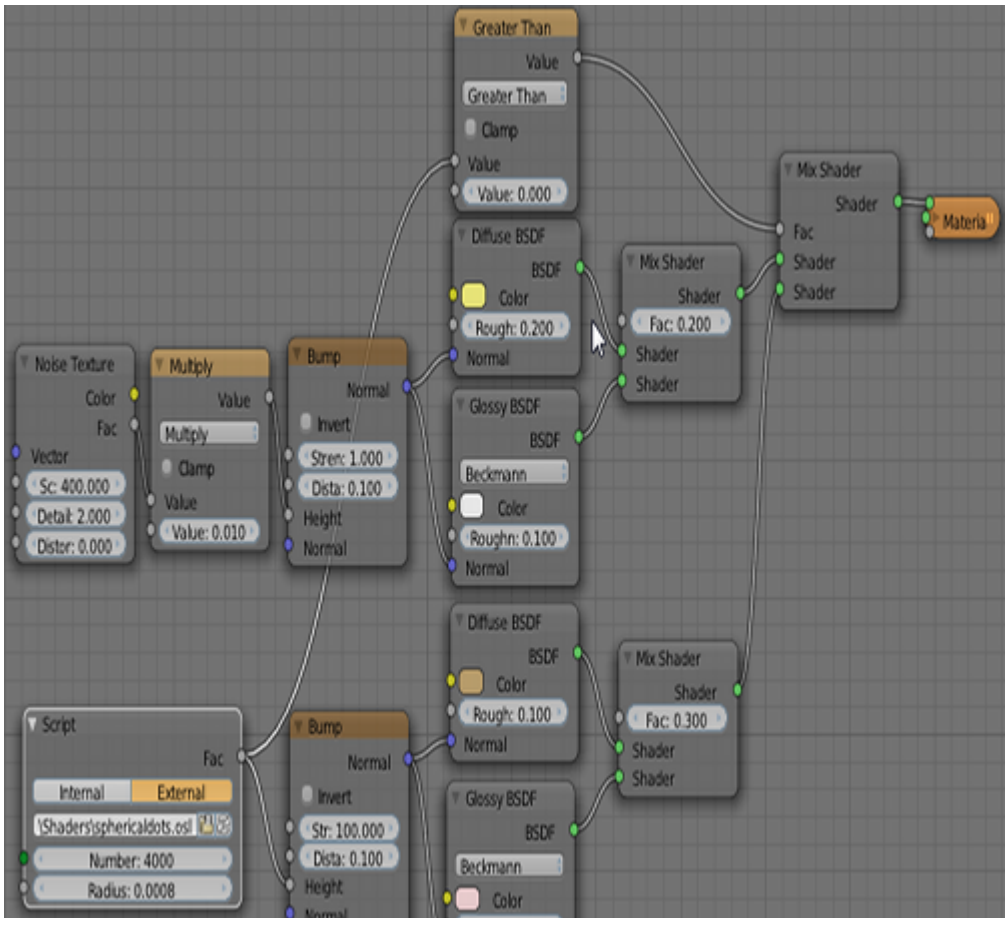
```
01.  #include "halton.h"
02.
03.  vector pointonsphere(int index){
04.    float z = haltonsequence(
05.                  index,2)*2-1;
06.    float t = haltonsequence(
07.                  index,3)*M_2PI;
08.    float r = sqrt(1-z*z);
09.    float s,c;
10.    sincos(t,s,c);
11.    return vector(s*r,c*r,z);
12.  }
13.
14.  shader sphericaldots(
15.    int Number = 100,
16.    float Radius = 0.03,
17.
18.    output float Fac = 0
19.  ){
20.    point p = transform("object",P);
21.    float r = length(p);
22.
23.    for(int n=1; n<=Number; n++){
24.       vector v = r*pointonsphere(n);
25.       float d = distance(v,p);
26.       if( d < Radius ){
27.         Fac = Radius - d;
```

```
28.          break;
29.        }
30.    }
31.  }
```

In the code above we transform the point being shaded to object space. That means that all coordinates are relative to the object center. We assume that the object being shaded is perfectly spherical so we can determine its radius by calculating the length of p, which will (because it is relative to the object center) be the length of the vector from the center to p. This will not be to far of even for roughly spherical objects like a grapefruit but will give strange results for non-spherical objects.

To determine the distance from the point being shaded to the randomly generated points we calculate the distance between these points in a straight line. This is accurate enough for points that are close together but will fail if the two points are at opposite sides of the sphere because then the straight lines distance would be much shorter then the distance along the surface. Because we have a large number of points the points nearest the point being shaded will be close and the straight line distance will be good enough to determine which is closest.

As you might expect this code is simple but gets slower as the number of dots increases because although there is no need to check each generated dot against all others, it is still necessary to compare all dots against the current position. For the grapefruit in the picture below I used 5000 dots based on my measurements of a real grapefruit (its diameter was about 16cm which gives a surface area of about 800 square centimeters. With 5000 dots this results in about 6 dots per square centimeter which seems about right for the grapefruit I studied) and the render time was still acceptable.


Greater Than
Value
Greater Than
Clamp
Value
Value: 0.000
Mix Shader
Shader
Fac
Shader
Shader
Materia
Diffuse BSDF
BSDF
Color
Rough: 0.200
Normal
Mix Shader
Shader
Fac: 0.200
Shader
Shader
Noise Texture
Color
Fac
Vector
Sc: 400.000
Detail: 2.000
Distr: 0.000
Multiply
Value
Multiply
Clamp
Value
Value: 0.010
Bump
Normal
Invert
Stren: 1.000
Dista: 0.100
Height
Normal
Glossy BSDF
BSDF
Beckmann
Color
Roughn: 0.100
Normal
Diffuse BSDF
BSDF
Color
Rough: 0.100
Normal
Mix Shader
Shader
Fac: 0.300
Shader
Shader
Script
Fac
Internal
External
\Shaders\sphericaldots.osl
Number: 4000
Radius: 0.0008
Bump
Normal
Invert
Str: 100.000
Dista: 0.100
Height
Glossy BSDF
BSDF
Beckmann
Color

Additional reading material

All you want to know about citrus fruit can be found in this citrus article.

More on Halton sequences can be found in this Wikipedia article

Various methods of generating uniform spherical distributions are documented on Math World

Ceiling plaster

Focus

- creating anisotropic reflections
- using BSDFs aka Closures

Anisotropy

Anisotropy, that is some property that is not the same in all directions is not limited to glossy reflections of brushed metals alone. An example is the final brushwork applied to the finish of some plastered ceilings or walls.

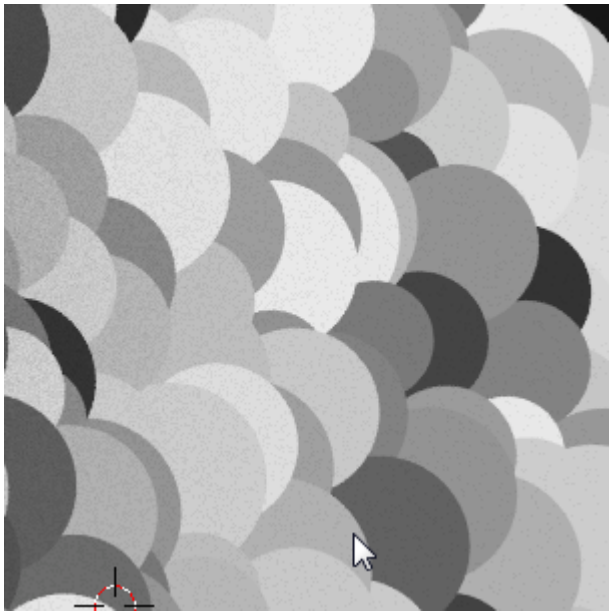
To get an even result the plasterer applies a brush or sponge in broad semicircular strokes when the plaster has almost dried. The effect is to break the uniform and rather boring off white expanse of the ceiling in a rather subtle way.

If we want to mimic the appearance of the plaster finish we may use the anisotropic shader that is available in Cycles and provide a suitable tangent vector but we still have to find a way to create a a pattern of semi circular areas that is more or less random.

In the shader presented below we consider a grid with a randomly positioned point inside each grid cell. We then check the neighboring cells and the cell where the point currently being shaded is in to see if we are in range of a circle centered at such a random point.

This is just like many of the shaders we developed so far but instead of checking all neighboring cells to find the closest random point we scan those cells from left to right and top to bottom and stop as soon as we

have found an overlap. The first point we find is then used to calculate the anisotropic glossy reflections relative to this center. Because we don't look for the closest point but stop at the first one found the resulting areas of influence look like this:

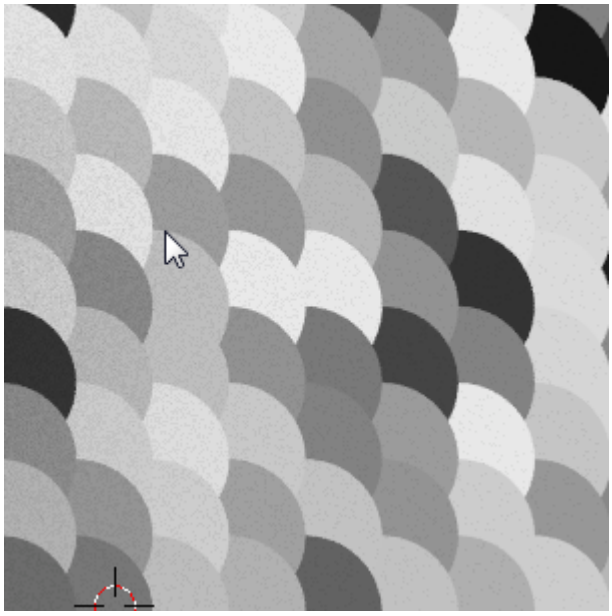


Shaders/semicircles.osl

```
01.  shader semicircles(
02.    color Color = 0.8,
03.    point Pos = P,
04.    float Scale = 8,
05.    normal Normal = N,
06.    float Pvar = 1,
07.    float R = 1,
08.    float Rvar = 0.3,
09.    float Roughness = 0.2,
10.    float Anisotropy = 0.5,
11.
12.    output float Fac = 0,
13.    output closure color BSDF = 0
14.  ){
15.    vector p=Pos * Scale;
```

```
16.    vector pi=floor(p);
17.    int range=(int)ceil(R+Rvar);
18.    for(float x=-range; x<=range; x++){
19.      for(float y=-range; y<=range;y++){
20.        vector pc=pi+vector(x,y,0);
21.        vector pq=pc+Pvar
22.                    *noise("cell",pc,1);
23.        float r =R+Rvar
24.                  *noise("cell",pc,2);
25.        vector d=pq-p;
26.        if(length(d)<r){
27.          vector Tangent = cross(
28.                          normalize(d),
29.                          vector(0,0,1));
30.          BSDF = Color
31.            * ward(Normal,
32.              Tangent,
33.              Roughness*(1-Anisotropy),
34.              Roughness/(1-Anisotropy));
35.          Fac = noise("cell",pc,3);
36.          return;
37.        }
38.      }
39.    }
40.  }
```

The code for the semicircles shader has a number of parameters besides the position and the scale that are specific to this shader. The first is `Pvar` which controls the randomness of the point distribution. Setting this to zero will result in a completely regular placement reminiscent of fish scales.



We also provide an input for the normal that defaults to the calculated normal of the mesh we are shading but might be replaced by a normal that simulates the basic fine grained structure of the ceiling plaster as is demonstrated in the sample node setup at the end of this section.

Next is the color of the anisotropic reflection. The default is white, suitable for most non-metallic materials. `R` and `Rvar` are the radius and the amount of randomness added to the radius of the semi circular areas. The `Roughness` determines how sharp a reflection is and the final input parameter `Anisotropy` shapes the reflection: a value of zero will give a circular reflection while non-zero values will elongate the reflection along the tangent for negative values and perpendicular to the tangent otherwise. Note that if the `Roughness` is zero the value for `Anisotropy` will have no effect.

The shader produces two output values. `Fac` is a random gray value that is uniform within the circle of influence. The output value `BSDF` is of a type not encountered before; a closure color.

Closures

Closures or bidirectional scattering distribution functions (BSDF) are descriptions of the way incoming light is reflected depending on the angle of the incident light and the viewing angle. They cannot be programmed from scratch in OSL but Blender Cycles has a fair number of these closures available and they are what we normally call shader nodes (accessible with `Add -> Shader` in the node editor).

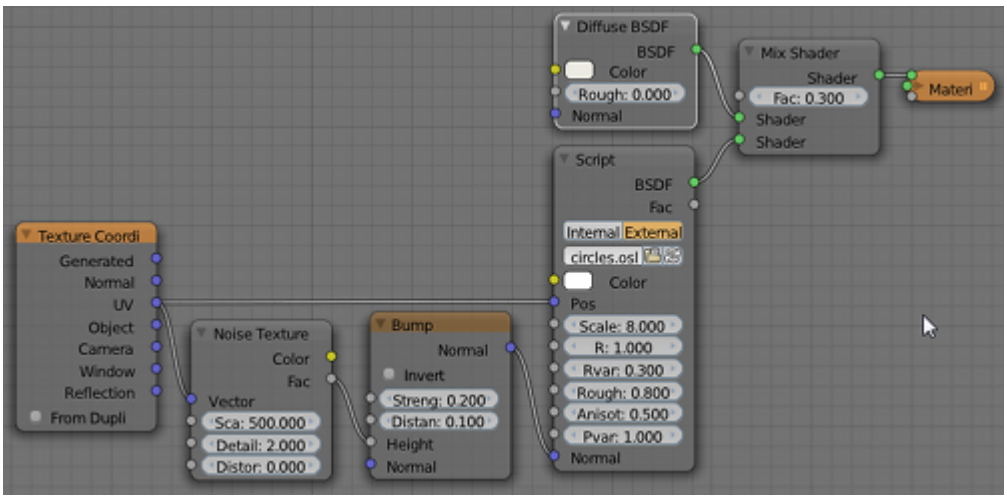
The specific BSDF we use here is called `ward()` and we could have simply calculated the tangent vector as output and plugged that into a Anisotropic node but by calling this ourselves we save a few calculations and we have a nice demonstration of their use within OSL code. Note that the `ward()` closure and hence our shader deals with reflection only so in a final shader setup you would probably mix it with a diffuse shader, for example to provide a base color.

The body of the semicircles shader consists mainly of two nested for loops used to determine to which area of influence the current shading position belongs. We do not restrict the search for center points to just the nearest neighbors but calculate the range based on the radius plus the variance in radius because the resulting range could be larger than one and result in visible artifacts.

In the innermost loop we calculate the center of the area and a random radius before calculating the direction vector `d` which points from the point currently being shaded to the center `pq` of the area of influence. We are inside this area if the length of the vector `d` is smaller than the radius. If this true we calculate the `Tangent`, a vector perpendicular to both the surface normal and the vector pointing to the center of influence. This is calculated by taking the cross product of these two vectors were we have the normal pointing in the z-direction because we expect to be dealing with uv coordinates.

The closure that is returned is the color of the reflection multiplied by the `ward()` closure and the for loops are all terminated by the return statement.

An example image with a sample node setup is shown below. The lighting and roughness is exaggerated to ensure this rather subtle effect shows in book reproductions.



Additional reading

An excellent tutorial on how to use the regular anisotropic glossy shader in Cycles is this one from Andrew Price.

More on BSDFs can be found in this Wikipedia article.

A mountain shader

Focus areas

- arrays
- string operations

Large mountains can have a varied texture due to their dramatic geometry. At the top they might be covered in snow while at lower heights plants may grow if the slope is not too steep.

The key ingredient here is being able to determine how high we are and what the orientation angle is of the point that is being shaded.

Obtaining the information so that we can select different materials in a Cycles node setup is doable by combining nodes but having all necessary information grouped together is much more convenient.

Determining height and slope

The shader presented below offers a number of input options. Because the positive z-axis is not always the relevant up-axis of a mesh we provide the user with the option the specify the axis, either x, y or z is a valid choice as are the -x, -y and -z axes.

Normally (pun intended) the normals of a mesh will point outward but if this is not the case we offer the option the invert them when we calculate the angle (the popular ANT landscape generator for example has all the normals pointing down and if actually inverting them on the mesh is not an option is is nice to have that choice here).

The `MirrorHeight` and `MirrorAngle` options will treat negative heights and angles just like positive ones when set to non zero. This offers interesting possibilities when shading sphere like objects: you could add ice on both poles when creating a planet shader for example.

The next four input values define what we regard as high or steep. Any value larger than `UpperLimit` will be seen as high. To get a more natural divide it is possible to add some noise to this value, the amount of which is defined by the `LimitVar` input.

Likewise `SteepestAngle` is the smallest angle (in degrees) that we regard as steep, lower angles are considered to be flat. Some noise can be added to this angle by choosing a nonzero `AngleVar`.

All these inputs result in two output values, `High` and `Steep`, which will be nonzero if conditions apply. They can be used to drive a mix shader for example.

Code is available in Shaders/mountain.osl

```
01.  shader terrain(
02.    point p = P,
03.
04.    string UpAxis = "z",
05.    int InvertNormal=0,
06.    int MirrorHeight=0,
07.    int MirrorAngle=0,
08.
09.    float UpperLimit = 0.5,
10.    float LimitVar = 0.1,
11.    float SteepestAngle = 45,
12.    float AngleVar = 5,
13.    float NoiseScale = 1,
14.
15.    output int High=0,
```

```
16.    output int Steep=0
17.  ){
18.    vector up[6]={
19.      vector(1,0,0),  // x-axis
20.      vector(-1,0,0), // neg. x-axis
21.      vector(0,1,0),  // y-axis
22.      vector(0,-1,0), // neg. y-axis
23.      vector(0,0,1),  // z-axis
24.      vector(0,0,-1)  // neg. z-axis
25.    };
26.
27.    int i=0;
28.    if(startswith(UpAxis,"-")){
29.      i++;
30.    }
31.    string last=substr(UpAxis,-1);
32.    if(last=="z"){
33.      i+=4;
34.    }else{
35.      if(last=="y"){
36.        i+=2;
37.      }else{
38.        if(last!="x"){
39.          warning(
40.            "Unknown axis [%s]",
41.            UpAxis);
42.        }
43.      }
44.    }
45.    vector axis=up[i];
46.
47.    normal n=N;
48.    if(InvertNormal){
49.      n *= -1;
50.    }
51.
```

```
52.    float d=length(p*axis);
53.    if(MirrorHeight){
54.      d=abs(d);
55.    }
56.    float angle=degrees(
57.                   acos(dot(n,axis)));
58.    if(MirrorAngle){
59.      angle=abs(angle);
60.    }
61.
62.    High = d>(UpperLimit
63.             +LimitVar
64.             *noise("perlin",
65.                    p*NoiseScale,
66.                    1));
67.
68.    Steep = angle>(SteepestAngle
69.                  +AngleVar
70.                  *noise("perlin",
71.                         p*NoiseScale,
72.                         2));
73.  }
```

We start of by defining an array of vectors, each defining a principal axis. The even slots hold the positive axes, the odd slots contain the negative axes.

OSL is not a string processing language but has enough standard functions available to let us easily determine what axis the user has selected. We first check if a chosen axis starts with a minus sign with the `startswith()` function and if so we add one to our axis index.

Next we use the `substr()` function to slice of the last character. `substr()` allows a negative offset to specify where to start from the end. We compare this sliced off character against valid axis names and add

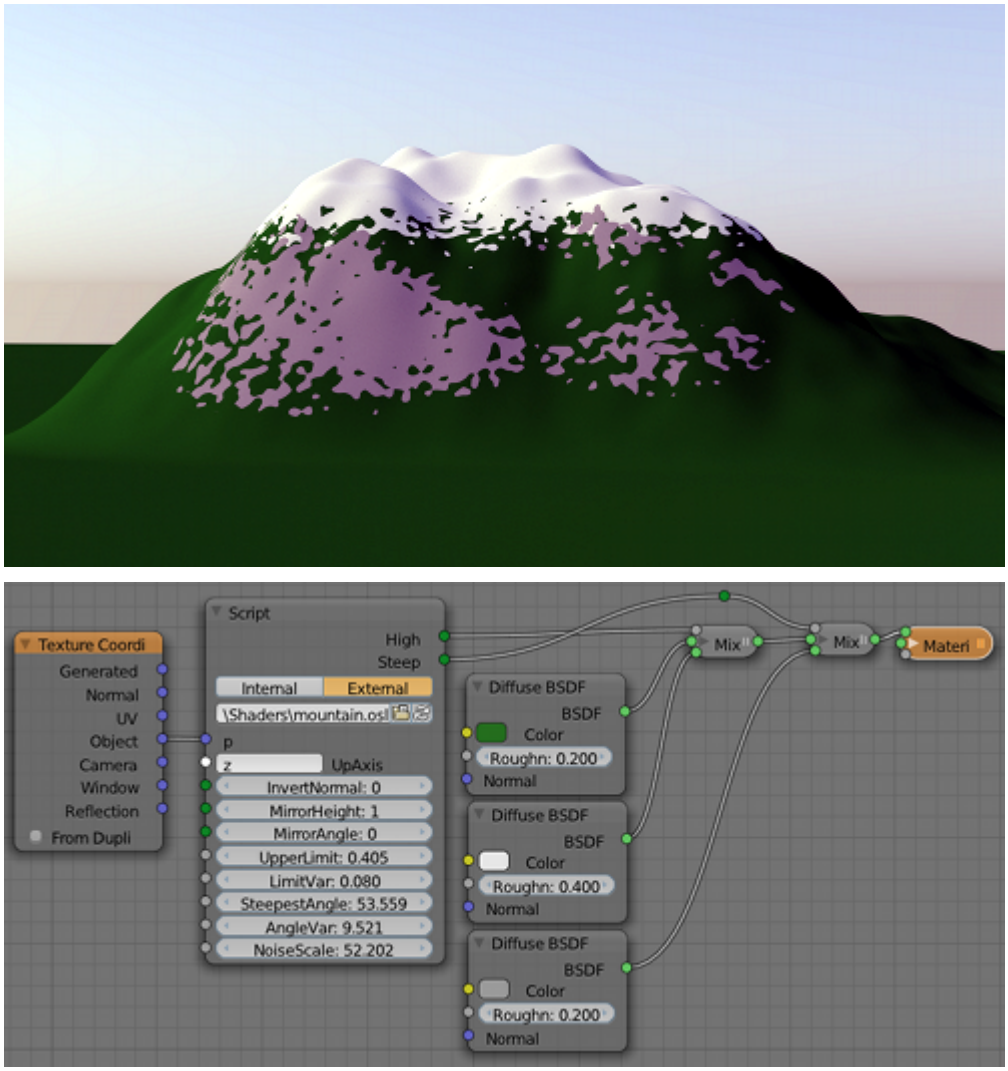
an appropriate offset to the axis index. If the axis specified was not valid we issue a warning that will show up on the console. These checks can be made better but will at least catch the most common errors.

We now select the chosen axis from the array and proceed to check if we need to invert the normal.

The next step is to calculate the height by projecting the position being shaded onto the chosen up axis (a component by component multiplication). If requested we treat negative values just like positive one by converting the result with the `abs()` function.

The slope is calculated by taking the dot product of the normal and the up-axis. There is no need to normalize the arguments because the normal already is normalized and all possible up axes were defined as unit vectors. The dot product is the cosine of the angle between the two vectors so we use the `acos()` function to convert it to an angle.

Then we take the absolute value of the angle if requested and finish with the actual work of comparing the calculated height and angle against their limit values (plus some noise) and set the output values accordingly. Correlation between the noise values is prevented by adding an extra parameter besides the position which is made different for each noise call.



Additional reading

The [ANT landscape generator] is a popular and easy way to create landscape meshes. It is distributed as an add-on bundled with blender.

Lots of pretty pictures

Focus areas

- reading texture files
- working with texture meta-data

Most of the time we use OSL to generate textures that do not rely on images but sometimes textures are convenient to store precomputed vales or some imagery that we want to reuse. The latter is what we do in this section where we develop a shader that will 'splat' or 'rubber-stamp' small copies of an image at random locations across an object.

To access information stored in an image file OSL offers several functions. One is `gettextureinfo()` which provides information like resolution and the number of color channels in an image and another is `texture()` which can be used to retrieve the color and alpha values at a given position. The `splat` shader presented in the code below uses the `gettextureinfo()` function to retrieve the resolution of the image to calculate an aspect ratio from these values (line 39 and 47).

After scaling the position the next step is to generate nine randomly positioned points. This is done by the `voronoi9z()` function in a manner that might be familiar to you from previous examples but this one has a twist: besides a random position each point is also assigned a additional random value and the list of points is sorted according to this additional value. This will assure that when we draw overlapping images this will happen in a reproducible order, independent from the position being shaded. (The `sort()` is identical to the one presented in the leopard spot shader and not shown here)

We treat the list of points that is returned as two dimensional points, that is we use just the x and y values to position the image and use the z value as a rotation. To position the image we calculate the relative distance q, rotate these coordinates around the z-axis and scale them to get a differently sized image if we want. We also add a value of 0.5 to ensure that the coordinates are centered correctly: the `texture()` function assumes that the coordinates passed to it are in the [0,1] range.

The `texture()` function requires a full pathname to an image file and two coordinates. It will return the color at those coordinates. Those coordinates are centered at the upper left corner of the image and assumed to be in the range [0,1]. That implies that if you want to use a non square image you have to use the `gettextureinfo()` function to get the dimensions of the image and calculate the aspect ratio which can then be used to correct the distortion, which is exactly what we did at the start of the shader. If you are only working with square images you might omit this step, but allowing for different aspect ratios makes the shader much more versatile.

The `texture()` function accepts a number of optional named parameters and in this shader we use one: the "alpha" parameter will provide the alpha value if available in the output variable provided. Here we use this alpha value to blend the image with the background which may consist of copies of the image drawn earlier. The final result is stored in the `Color` variable.

Code available in Shaders/splat.osl

```
01.  void voronoip9z(point q,
02.            output point vp[9],
03.            output float z[9])
04.  {
05.    point p = point(q[0],q[1],0);
06.
```

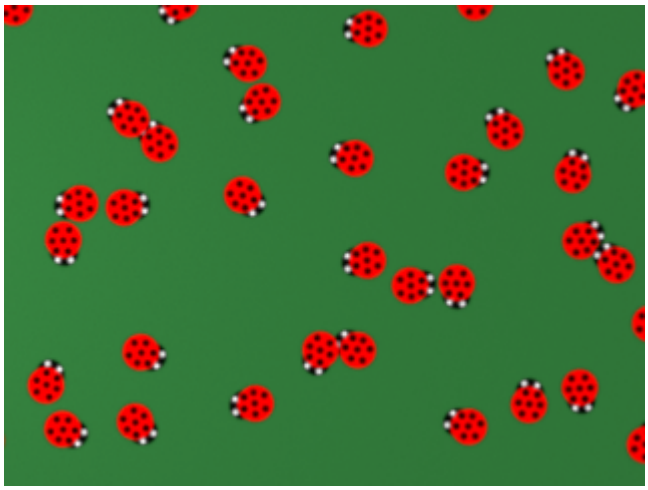
```
07.    float xx, yy, xi, yi;
08.
09.    xi = floor(p[0]);
10.    yi = floor(p[1]);
11.
12.    int i=-1;
13.    for (xx=xi-1; xx<=xi+1; xx++){
14.      for (yy=yi-1; yy<=yi+1; yy++){
15.        i++;
16.        vector ip =vector(xx, yy, 0);
17.        vector np =cellnoise(ip);
18.        vp[i] = ip + np;
19.        z[i] = cellnoise(ip,42);
20.      }
21.    }
22.    sort(9,vp,z);
23.  }
24.
25.  shader splat(
26.      point Pos = P,
27.      float Scale = 1,
28.      string Texture = "",
29.      color Background = 1,
30.      float TextureScale = 1,
31.
32.      output color Color = color(1,0,1)
33.  ){
34.      point p = Pos * Scale;
35.
36.      color splat, result;
37.      float alpha;
38.      int resolution[2];
39.      if( !gettextureinfo(
40.             Texture,
41.             "resolution",
42.             resolution
```

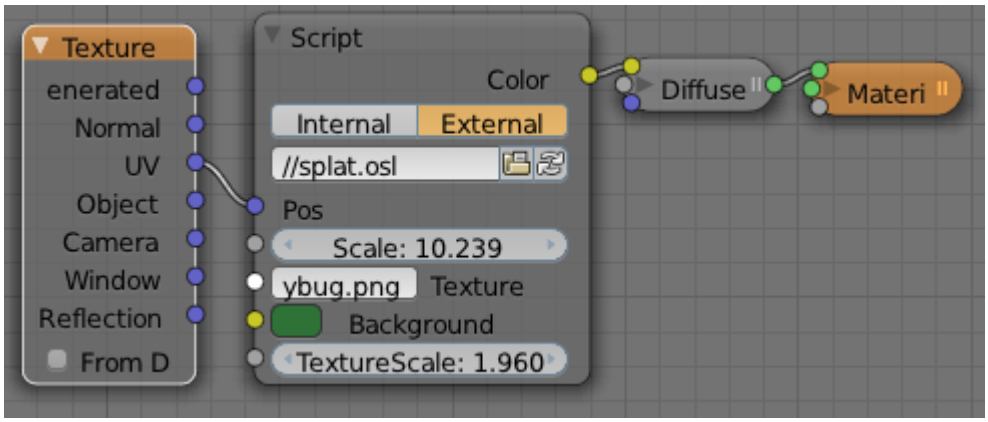
```
          )
     ){
       return;
     }
     float aspectratio =
            (float)resolution[1]
                  /resolution[0];

     point vp[9];
     float z[9];
     voronoip9z(p,vp,z);

     result = Background;
     for(int i=0; i<9; i++){
       vector xy =
                 vector(vp[i][0],
                 vp[i][1],
                 0);
         vector q = vp[i] - p;
         q = rotate(q,
                vp[i][2]*M_2PI ,
                point(0,0,0),
                point(0,0,1));
         q = q + 0.5;
         q = q * TextureScale;
         splat = texture(
            Texture,
            q[0]*aspectratio,
            q[1],
            "alpha", alpha);
         result = mix(
            result, splat, alpha);
     }
     Color = result;
 }
```

The example image shown below (with the node setup used to produce it) was created with the image `ladybug.png` that's available in the Shaders directory as well.





There are a few things to note when working with textures; as seen in the code example we refer to the texture by specifying its full filename including its path in a string. The node will display a text entry field for a string input parameter but don't expect a file browsing dialog because Blender's node drawing routines have no way of knowing this string will

hold a file name. (OSL does support the concept of meta-data for shader parameters but currently Blender does not take this information into account so we do not cover it in this book).

Another thing is that you might wonder if opening an image file and reading the data isn't slowing things down tremendously if this is done for every sample. This would indeed be slow but the OSL compiler arranges for the texture to be read once before the shader is executed and stores its data in memory for rapid access.

Hedge shader

Focus areas

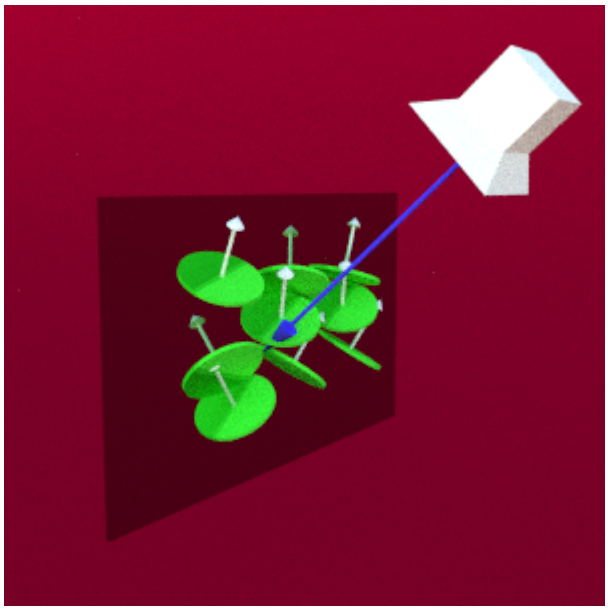
- working with the incidence vector
- using textures
- faking geometry

A well developed and often trimmed hedge shows almost no branches on the outside but consists of a seemingly innumerable amount of leaves that would be very expensive to model as it consist of many faces.

Such a hedge is a good candidate for a shader based approach where we simulate the leaves in the hedge rather than model them. This won't work if the hedge is too close to the camera but for intermediate distances this tends to work very well. A lot of detail can be produced without complicated geometry and the amount of work the shader has to do per pixel stays the same whatever the distance. That means that if our hedge is further away it will render faster if it appears smaller without us having to to prune unnecessary hard to see leaves.

The material we develop here is typically applied to a simple plane yet has to convey the impression of three dimensional leaves. This means it need not only define a shape with proper transparent parts that change its silhouette when the viewing angle changes, but these shapes need to cast proper shadows and have correct surface normals to drive glossiness in a realistic manner.

The way we approach this here is to envision the collection of leaves as circles that have their surface normal roughly oriented upward. Now if we render a pixel the ray from the camera to the point on the plane that we are rendering might pass through one of these circles and if that happens we color the pixel and if not we leave it transparent.



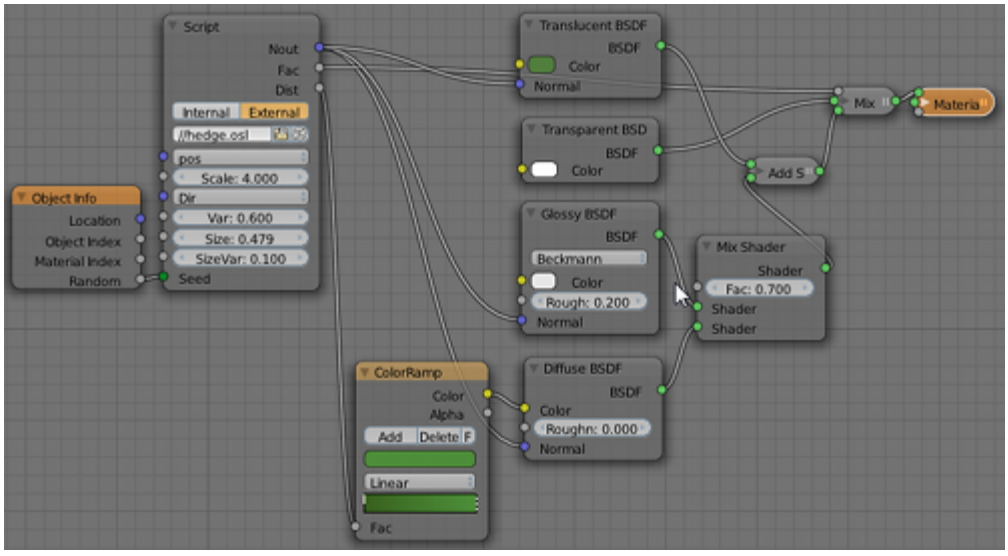
The ray from the camera to the point being shaded is available in OSL as the global vector variable `I`. The `I` is short for incidence vector and in fact it is available not only for the camera ray but for all rays bouncing around in the scene, a feature that will give us correct shadows without any additional effort.

In the shader presented below we randomly distribute the leaves in a manner familiar from previous shaders. Each of the leaves is considered

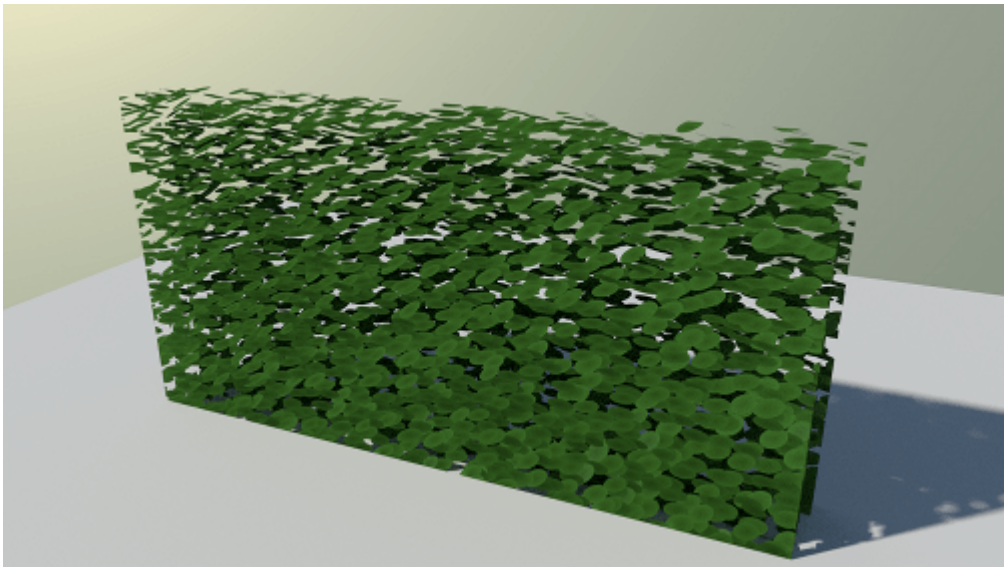
to be a circle that lies in a plane. The first thing to do is to determine where the ray along the incidence vector `I` toward the point being shaded `p` intersects this plane. The plane is defined by the randomly chosen position of a leaf and its normal and determining the intersection point is straightforward algebra (see for example [this Wikipedia article]).

Now that we have the point of intersection on the plane we need to determine if it is within radius of the point that defines our circular leaf. If this is the case we set the `Fac` output parameter to 1 (to indicate we are in a leaf but also to terminate any loop we're in because once we detect we are seeing a leaf there is no need to check if we are seeing other leaves), set the `Nout` parameter to the normal of the plane and return the distance (or actually the squared distance) to the center of the circle in `Dist`.

These outputs can be used in a node setup like the one shown below:



When applied to a few parallel planes it produces the impression of a three dimensional hedge even when seen from an angle.



The code for the shader is presented below:

Code available in Shaders/hedge.osl

```
01.  shader hedge(
02.    vector pos=P,
03.    float Scale=1,
04.
05.    vector Dir=vector(0,0,1),
06.    float  Var=0.1,
07.
08.    float Size=0.3,
09.    float SizeVar=0.1,
10.
11.    int Seed = 42,
12.
13.    output normal Nout=N,
14.    output float Fac=0,
15.    output float Dist=0
16.  ){
17.    vector p=pos*Scale;
18.
19.    for(int xx=-1;
```

```
      xx<=1 && Fac == 0;
      xx++)
  {
    for(int yy=-1;
        yy<=1 && Fac == 0;
        yy++)
    {
      for(int zz=-1;
          zz<=1 && Fac == 0;
          zz++)
      {
        vector pp = p
             + vector(xx,yy,zz);
        vector ip=floor(pp);

        vector leafp = ip
          +noise("cell",pp,Seed);
        vector leafn =
           normalize(
             Dir+(noise("cell",
                    pp,
                    Seed+1)
              -vector(0.5,0.5,0.5)
              )*Var
           );
          float  leafs = Size
            +SizeVar*noise(
                      "cell",
                      pp,Seed+3);

          vector in =
                    normalize(I);
          float d =
            dot(leafp - p, leafn)
           /dot(in, leafn);
          vector dp =
```

```
56.              leafp - (d * in + p);
57.
58.            float r = length(dp);
59.            if( r < leafs ){
60.              Fac=1;
61.              Nout=leafn;
62.              Dist = r;
63.            }
64.         }
65.       }
66.     }
67.   }
```

The only extra trick is the way we generate a random normal. The general direction of the normal is an input parameter but we randomize it somewhat by adding a random vector. This vector will have positive values only for its components so we subtract 0.5 so the range will be [-0.5, 0.5] before we multiply it with the `Var` parameter. Setting the `Var` parameter to zero will make sure all normals point in the same direction but choosing larger values will generate more varied normals. We do normalize the resulting normals to make sure they all have a length of one.

Using a leaf image

Hardly any leaf is a plain green circle so it would probably be a good idea to use an image texture instead of a just a circle to enhance the realism.

In a previous shader we saw how we could apply an image texture to a plane so here we need to devise a way to apply this texture to the part of the plane that represents the leaf and because we need to define a square here rather than just a circle the math is a little bit more involved.

To define a square that is randomly oriented around the chosen surface normal we need to pick a random vector that lies in the plane. A vector that lies in the plane is perpendicular to the normal that defines that plane and one trick to generate such a vector is to create a completely random one and then calculate the cross product of this random vector with the surface normal. The resulting vector is then perpendicular to both that random vector and to our normal vector as well. Normalizing it will give us a usable unit vector `leafk`. (More on cross products in this Wikipedia article.)

To define a square we need not one but two perpendicular vectors. This second vector `leafj` cannot be chosen independently but should be both perpendicular to `leafk` and to the surface normal `leafn`, so we calculate it by taking the cross product of `leafk` and `leafn`.

The next step is to determine if the point where the incidence vector intersects the plane is within the square defined by the two vectors. This is accomplished by projecting the intersection point onto those vectors and checking if the length of this projection is in the range [0,1]. Because `leafj` and `leafk` are unit vectors, the length of the projection of the intersection point with any of them can be calculated using the dot product. (Again, Wikipedia might help you to see why, especially the part on scalar projection.)

If these projections are within range they are used as offsets to retrieve a color value from a texture. We also retrieve the alpha value and only set `Fac` to 1 when the alpha value is 1. This way we don't stop looking for intersections with other leaves if we are within a transparent section of an image, a necessary action to ensure we can see the shape of leaves that are further below.

The code presented below largely follows the outline of the previous shader except for the additional work needed to calculate `leafj` and

`leafk` and the final mapping to the image texture.

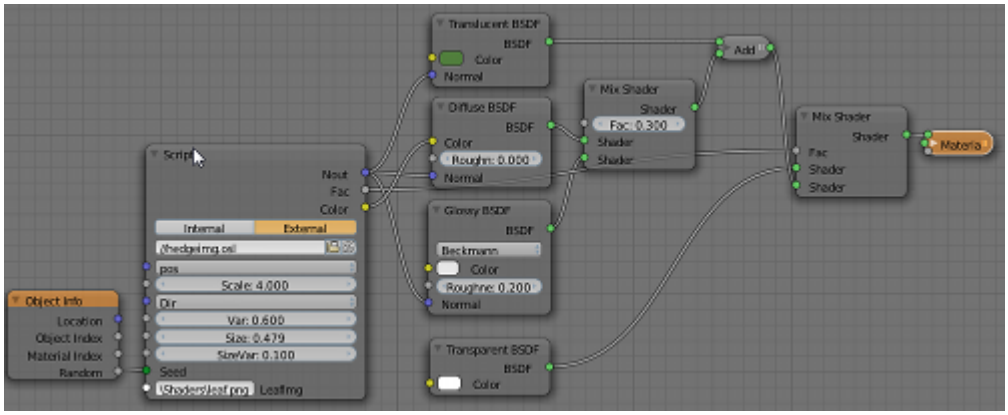
Code available in Shaders/hedgeimg.osl

```
01.  shader hedge(
02.    vector pos=P,
03.    float Scale=1,
04.
05.    vector Dir=vector(0,0,1),
06.    float  Var=0.1,
07.
08.    string LeafImg="",
09.    float Size=0.3,
10.    float SizeVar=0.1,
11.
12.    int Seed = 42,
13.
14.    output normal Nout=N,
15.    output float Fac=0,
16.    output color Color=0
17.  ){
18.    vector p=pos*Scale;
19.
20.    for(int xx=-1;
21.      xx<=1 && Fac == 0;
22.      xx++){
23.      for(int yy=-1;
24.        yy<=1 && Fac == 0;
25.        yy++){
26.        for(int zz=-1;
27.          zz<=1 && Fac == 0;
28.          zz++){
29.          vector pp =
30.                p + vector(xx,yy,zz);
31.          vector ip=floor(pp);
32.
```

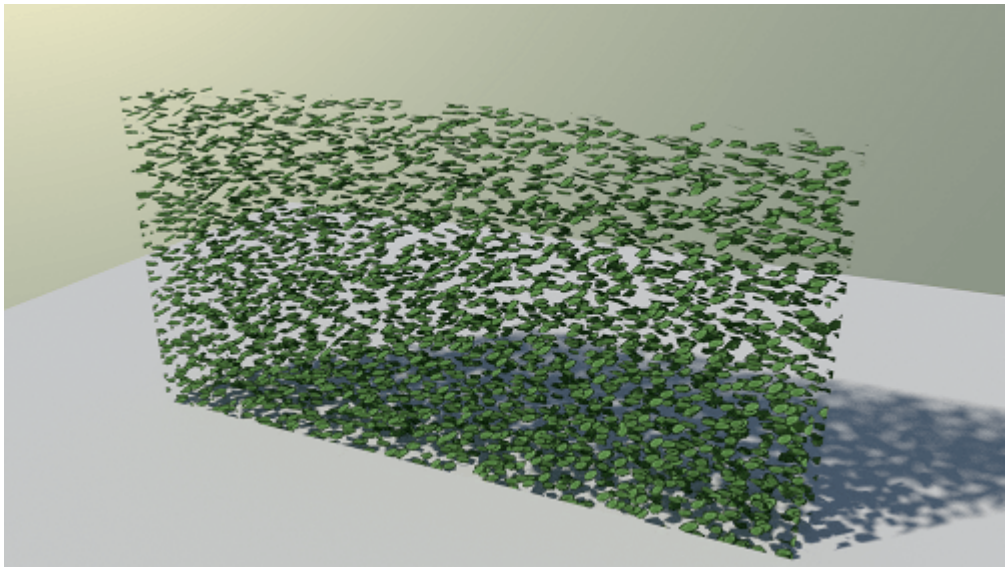
```
33.          vector leafp =
34.          ip+noise("cell",pp,Seed);
35.
36.          vector leafn = normalize
37.               ( Dir
38.                +(noise("cell",pp,
39.                         Seed+1)
40.                -vector(0.5,0.5,0.5)
41.                 ) * Var
42.               );
43.          vector leafk =
44.             noise("cell",pp,Seed+2);
45.          vector leafj =
46.               normalize(
47.                 cross(leafn,leafk));
48.          leafk = cross(leafn,leafj);
49.          float  leafs =
50.            Size + SizeVar
51.            *noise("cell",pp,Seed+3);
52.
53.          vector in = normalize(I);
54.          float d =
55.                dot(leafp - p, leafn)
56.               /dot(in, leafn);
57.          vector dp =
58.                leafp - (d * in + p);
59.
60.          float dpx = dot(dp,leafk);
61.          float dpy = dot(dp,leafj);
62.
63.          if( dpx>0 && dpy>0){
64.            float u = dpx/leafs;
65.            float v = 1-dpy/leafs;
66.            float alpha=0;
67.            color c=texture(
68.                  LeafImg,
```

```
69.                     u,v,"alpha",alpha);
70.             if(alpha){
71.               Fac=1;
72.               Nout=leafn;
73.               Color=c;
74.             }
75.           }
76.         }
77.       }
78.     }
79.   }
```

The node setup to use this shader isn't all that different from the previous one:



The result is dramatically different even though we did use a fairly crude image for the leaf texture:



Grid displays

Focus areas

- dictionaries
- XML files

In this section we will develop a shader that mimics displays based on a matrix of small LED lamps. These displays are for example used as a ticker on buildings or in shop windows and the color of the lights is often red.

To implement such a shader we need a way to convert a string to a pattern of lights. OSL can deal quite well with strings but has no facilities for handling font files. We therefore have to devise a way to define and read a font that can be used for our matrix display.

A very convenient feature available in OSL is the possibility to read data from XML files. From such a file XML elements can be accessed with so called Xpath expressions and OSL takes care of reading and parsing the XML once and storing the contents in memory to prevent shaders from becoming very slow.

Before we use these features in a matrix display shader, lets have a look at how to use these features in a simple way.

Access to XML data

Say we want to be able to convert color names to colors and want to keep this information in an XML file. That file might look like this:

Code available in Shaders/colors.xml

```
<colors>
<color name="aliceblue">240, 248, 255</color>
<color name="antiquewhite">250, 235, 215</color>
<color name="aqua">0, 255, 255</color>
```

```
<color name="aquamarine">127, 255, 212</color>
</colors>
```

It consists of a single root element `colors` which has a list of `color` child elements. Each `color` element has a `name` attribute which will be used as a key. The contents of the `color` hold the rgb values as a comma separated list.

The way to access this information is by using the built-in `dict_find()` and `dict_value()` functions as shown in the code below:

Code available in Shaders/colorname.osl

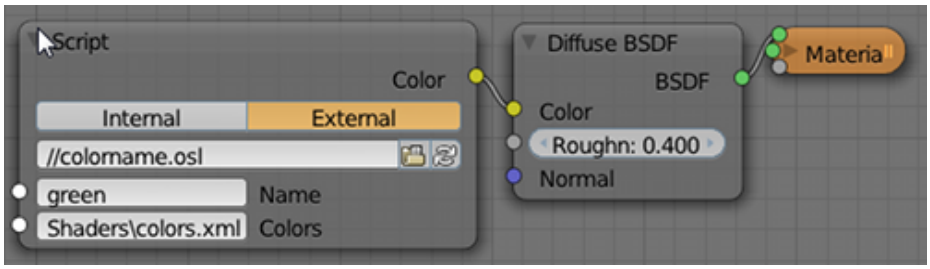
```
01.  shader colorname(
02.    string Name="black",
03.    string Colors="colors.xml",
04.
05.    output color Color = 0
06.  ){
07.    string xpath = concat(
08.                       "//color[@name='"
09.                       ,Name,
10.                       "']/text()"
11.                     );
12.    int nodeid =
13.                dict_find(Colors, xpath);
14.    if( nodeid>0 ){
15.      dict_value(nodeid,"",Color);
16.      Color = Color/255;
17.    }
18.  }
```

The input parameters of the shader node are both strings, one to specify the color name and a second one to specify the filename of the color dictionary. This filename must be a fully qualified path otherwise it won't be found (so there is no possibility to specify a path relative to the .blend file or to use an internal text buffer).

The first task is to construct an Xpath query to retrieve the color element that we need. (See additional reading for more on Xpath). If we entered red as the color name, the Xpath expression will look like `//color[@name='red']` which says in plain English "select all color elements with a name attribute equal to red".

If we pass this Xpath expression along with the filename to the `dict_find()` it will return a positive integer identifying the first matching `color` element or 0 if it didn't find any (and -1 if it couldn't read the file).

This positive integer is then passed to the `dict_value()` function to retrieve the value of the element. The result is placed in the Color output. Because Color is of the type color, the contents we retrieved are interpreted as a list of values. They should be separated by commas and/or whitespace. An example of a node setup is shown below. (No sample image of the result is shown to spare the reader the boredom of looking at a plain green square)



A matrix display shader

Now that we have a way to get information from an XML file we can concentrate on defining a font. Our simulated matrix display has five rows and seven columns for each character so our font definition looks like this:

Code available in Shaders/font5x7.xml

```
<font>
<char name="A" pattern="0111111 1000100 1000100 1000100 0111111" />
<char name="B" pattern="1111111 1001001 1001001 1001001 0110110" />
</font>
```

For each character we want to encode we have a `char` element with a `name` attribute that acts as the key and a `pattern` attribute that holds a space separated list of flags. A one means 'on' a zero 'off'. Each chunk of 7 flags represents a column so there are five chunks, one for each row.

Encoding a complete font this way is doable yet tedious but many people have already gone through the trouble of creating such a font for real hardware devices (see link in additional reading section) so we can reuse that.

With the font available in xml the shader can be implemented as follows:

Code available in Shaders/matrixdisplay.osl

```
01.  shader MatrixDisplay(
02.    point Pos = P,
03.    int Offset = 0,
04.    string Text = "Blender",
05.    string Font = "font5x7.xml",
06.
07.    output float Fac = 0
08.  ){
09.    float x = Pos[0]-Offset/5.0;
10.    int nchar = (int)floor(x);
11.    if( nchar >= 0
12.     && nchar < strlen(Text) )
13.    {
14.      string xpath = format(
15.                   "//char[@name='%s']",
16.                   substr(Text,nchar,1)
17.                   );
18.      int nodeid =
19.                  dict_find(Font,xpath);
20.      if( nodeid > 0 ){
21.        string pattern;
22.        dict_value(nodeid,
23.                     "pattern",pattern);
24.        int row= 6-(int)floor(Pos[1]*7);
25.        int col= (int)floor(mod(x,1)*5);
```

```
26.         if( row>=0 && row<7 && col<5 ){
27.           string bits[5];
28.           split(pattern,bits);
29.           if(substr(bits[col],row,1)
30.              == "1")
31.           {
32.             float xcell =
33.                   mod(mod(x,1.0)*5,1.0);
34.             float ycell =
35.              mod(mod(Pos[1],1.0)*7,1.0);
36.             Fac = max(
37.             (0.4-
38.              hypot(xcell-0.5,ycell-0.5)
39.             )/0.4,
40.             0.0);
41.           }
42.         }
43.       }
44.     }
45.   }
```

Besides the position of the text and text itself the shader also has an additional input parameter `Offset`. This will be used to move the pattern of lights that we generate to the left or to the right. The offset is an integer because we want to move the pattern by one column of lights at the time. If we animate this value it is possible to create a moving ticker.

The first step is to calculate which character we are rendering st this point. After adding the offset we round down to the nearest integer value with the floor() function as we have one character per unit distance.

We the check if this character index actually lies within the string we are rendering. If so we get this single character with the `substr()` function and use it to construct an Xpath expression in much the same way as we did for the color names shader.

We use `dict_find()` to search the XML file and if there was a match for the character we use `dict_value()` to retrieve the contents of the pattern

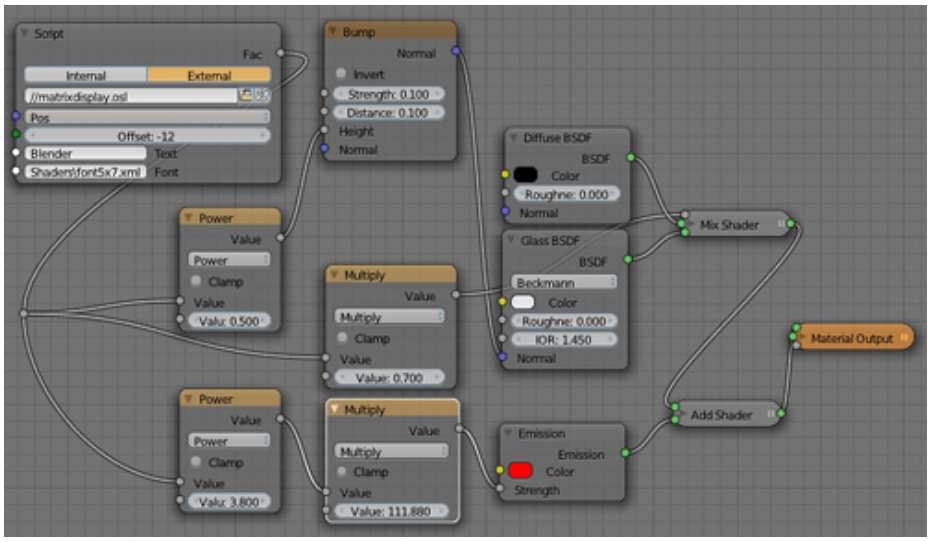
attribute.

The next two lines determine what row and column index to use to retrieve the relevant flag from the pattern. Note that because the top row is the sixth row from the bottom (we number starting at zero) but the first row of flags in the pattern, we have to subtract the row index from six to prevent the character from being displayed upside down.

After some sanity checks on the calculated indices we use the `split()` function to separate the pattern we retrieved in five separate strings. Because we did not supply a separation string the `split()` function will use any whitespace as a separator.

The final check is to see if the relevant flag is a 1. If so we calculate where our position is relative to the single LED lamp we are rendering. To the Fac output parameter we assign a value that starts at one and decreases linearly with the distance to the center of this LED lamp.

By not simply assigning 1 or 0 to Fac but a value that is one at the center and decreases when we are further away we give the end user more options in rendering the lamps. For example we can make the intensity dependent on this value to create the tiny spots of light that real LEDs have in their center. In the example shown below we did that and added some glare in the compositor.



Additional reading

XML and XPath are huge subjects but a lot of information to get you started can be found on w3schools.com.

The xml font file was converted from the font information used for Arduino hardware and can be found on
http://code.google.com/p/ht1632c/source/browse/font.h

Debugging OSL code

While syntax errors are often simple enough to fix because the compiler more or less exactly points them out in console messages, runtime errors are more difficult to spot and fix.

OSL as implemented in Blender does not come equipped with a debugger so we have to use somewhat old fashioned methods.

The first thing to understand is that many runtime errors are simply ignored: a division by zero error for example will not show up on the console but will probably result in unexpected output from your shader.

Equally annoying is that it is possible to create an endless loop that will stall Blender completely and cannot be interrupted from within Blender, forcing you to terminate Blender altogether.

It isn't all that bad of course and the following tactics might help you to spot a misbehaving line of code:

Build your shader step by step

It is easier to find and fix an error in ten lines of code than it is in two hundred. If you inherit a hefty chunk of code from someone else or return to a project you yourself wrote some time ago it might help to comment out large chunks of code. Commenting out large blocks of code with OSL comment tokens (the double slashes //) is tedious so an easier way is to use preprocessor directives:

```
#if 0
  ... this line will not be compiled ...
#endif
```

Inspect intermediate values

This is in a sense related to the first advice: if you suspect something goes wrong somewhere in the middle of your code it doesn't hurt to create an extra output socket and assign an intermediate result that you are interested in to this socket. This way you may inspect this value by connecting it to for example to a diffuse shader. This not only works for colors but float values will produce shades of gray when plugged into a color input.

It might be necessary to use some math nodes (`Add->Convert->Math`) to scale such value to lie within the [0,1] range. You can use the multiply mode of the math node for this. If you suspect that some values are out of range if might be useful to use the less than or greater than modes of the math node to identify those values. All these techniques are more or less non invasive, they let you inspect values without altering the code in your shader.

Send output to the console

OSL has several functions available that let you print information on the console (`warning(), error()` and `printf()`). Before you start using these it might be a good idea to prepare your test scene first by reducing the number of preview samples to 1 and restrict the area that is rendered in the 3dview to a tiny square with ctrl-b.Otherwise rendering will take forever because printing to the console is very slow and will happen for each sample.

When printing things to the console there is one peculiar feature you have to be aware of: printf calls (and error and warning) might be optimized away by the compiler as the result of other optimizations. Why that is is quite tricky to explain but might be very annoying if you are trying to track a subtle error in your logic. If you expect your printf

calls to print something but nothing at all shows up on the console it might be a good idea to disable optimizations. How to do that on various platforms is documented in this blog article.

With these simple techniques debugging shaders is in general not too difficult but it can be a time consuming and feel a bit awkward if you are used to more integrated development environments.

The OSL preprocessor

Before any of the OSL code in your shader is compiled, the source is first fed through a preprocessor. This preprocessor is the full preprocessor that is part of the C language which means it is quite powerful. In fact you can define macros and do all sorts of weird and wonderful stuff but in day to day use you will probably need just the basic features.

Defining constants

Using descriptive names for constants makes your code a bit more readable and that is what the `#define` preprocessor directive is for. For example:

```
#define EPSILON 1e-7
```

(Note that there is no = sign in this definition). Defining constants this way at the beginning of your file allows you to use them anywhere in your code, for example:

```
while(deltax > EPSILON){
  ... do something ...
}
```

It is possible to use a regular variable for the same purpose of course, but defining a constant with the preprocessor will prevent you from accidentally assigning a value to it.

If you define a constant with an expression, remember to put parentheses around this expression to prevent issues with operator precedence:

```
#define A (3+7)

float f = A*7;
```

Without parentheses `f` would be initialized to a value of 28 instead of 70.

Including files

If you want to reuse a collection of useful functions it makes sense to put them in a separate file. With the preprocessor `#include` directive it is possible to include such a file. Say you have a file called `functions.h` containing for example a definition for a function `square()`, you could use this function in the following way:

```
#include "functions.h"

...
a = square(b);
...
```

The include directive looks for the specified file in several places. If the file is a completely specified path, like `/blender/osl/includes/functions.h` or `E:/osl/includes/functions.h` it will only look at that specific file. If, like in the example, just a filename is given, it will try to locate this file first in the directory where the .osl file that is being compiled resides, and if not found it will try a default location (the directory `scripts/addons/osl/shaders` in Blender's installation directory).

If you use an internal text buffer for your OSL script node this means you can only include files from the default location because the `#include` directive cannot refer to other internal text buffers.

The default directory contains one very important file `stdosl.h` which contains a lot of constants and function definitions that are standard features of the OSL language. We do not have to include this file explicitly however as it is included automatically.

A good start to learn about the capabilities of the C preprocessor can be found on Wikipedia. The preprocessor in OSL for Blender is implemented with the Boost::Wave library, information about its specifics can be found on the Boost website but this is probably overkill for shader programmers.

Additional information

About the book

When I started writing blog articles on Open Shading Language for Blender back in November 2012, the response was great, both in number of visitors as in the comments on the blog and elsewhere. So when after a couple of months the attention did not diminish, I started thinking about writing a book.

The publishers I contacted did find the subject interesting and relevant but didn't see enough market potential, i.e. they didn't expect enough people wanted to buy a book on Open Shading Language to cover their production costs. That left me basically with three options: abandon the project, publish the articles on my blog or self publish. I opted for the latter.

I am not in it for the money (and believe me, even with a decent publishing deal you won't get rich writing technical books) but I already spent quite some time on the book so some financial reward was in order I felt. Now self publishing has the advantage of very low overhead costs and a large margin, which means you can keep the price really low and still earn a few euros, a win-win scenario for readers and author alike.

But the low overhead costs translate to doing everything yourself and if you want to do it well, it really takes some effort. Having to do the editing, indexing and not to forget the marketing yourself really makes you appreciate the work of a publisher!

Hopefully you will enjoy reading this book as much as I enjoyed writing it.

About the author

Although a Blender user for over ten years, I have to admit that I am an enthusiastic but (very) mediocre artist at best. I discovered however that I really enjoyed helping people out with programming related questions and a couple of years ago when Packt Publishing was looking for authors on the BlenderArtists/Python forum I stepped in.

So far this has resulted in two books, one on Blender 2.49 Scripting and one on Python 3 Web Development

I really enjoyed cooperating with a publisher and as this book on Open Shading Language proves I'm not done writing yet. I probably will continue writing as long as I enjoy doing it, even if the number of copies I expect to sell cannot convince a publisher to step in.

I maintain a blog on Blender related things, blenderthings.blogspot.com and I keep an eye on the coding forums at BlenderArtists where you can also contact me via private message if you like, my nickname there is *varkenvarken*

Any future books will likely show up my Smashwords author page: https://www.smashwords.com/profile/view/varkenvarken

I live in a small converted farm in the southeast of the Netherlands where we raise goats for a hobby. We also keep a few chickens and the general management of the farm is left to our cats. This arrangement leaves me with with enough time to write the occasional book.

Acknowledgments

Without the support of the people around me, this book would not have been written. I would especially like to thank my partner Clementine for proofreading and criticizing the manuscript and for her support in

general. Also the Blender developers and the community deserve thanks for creating such an enjoyable product and extra kudos to Brecht van Lommel and Thomas Dinges for making OSL available in Blender.

The text of this book was typeset with the following fonts (links are to Google Fonts if you want to see more of them):

Amaranth by Gesine Todt
Gentium Book Basic by Victor Gaultney
Ubuntu Mono by Dalton Maag

Unfortunately many e-book readers strip most style information by default so to get the benefit of viewing the text as intended you will have to disable this override.

Additional OSL information

Besides the GitHub repository of OSL

https://github.com/imageworks/OpenShadingLanguage

and especially the Language specification, the site maintained by Thomas Dinges (one of the people involved in implementing OSL in Blender) is a good source of information:

http://www.openshading.com/

Another go to for all Blender related discussions is of course BlenderArtists and especially the Coding forums which have dedicated areas devoted to OSL scripts.

And of course I keep on adding stuff to my blog.

###