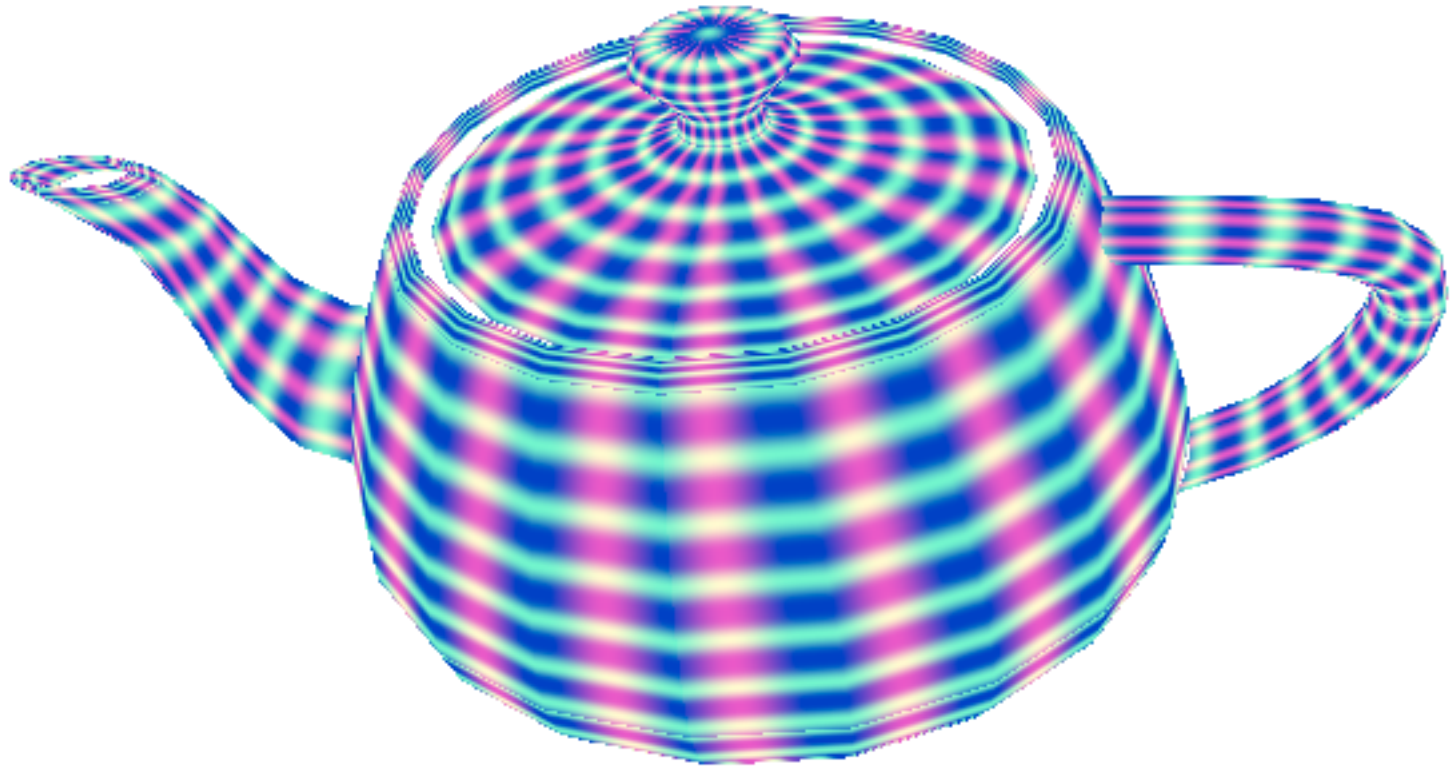
TNM084
Procedural images

Ingemar Ragnemalm, ISY



Lecture 5

A bit more on anti-aliasing

Open Shading Language

Using OSL with Blender

Lab 2

Dugga 1

Varied results.

Harder than expected and intended - which is usual for the first one.

Was I too unclear at some points...?

Rules for the retake

You have a second chance!

You can do all or some.

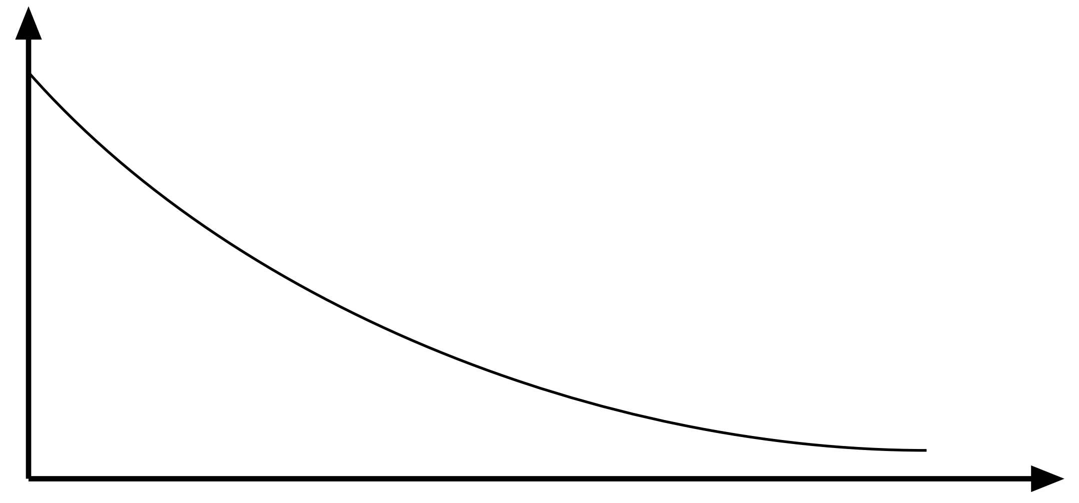
Not same questions but similar.

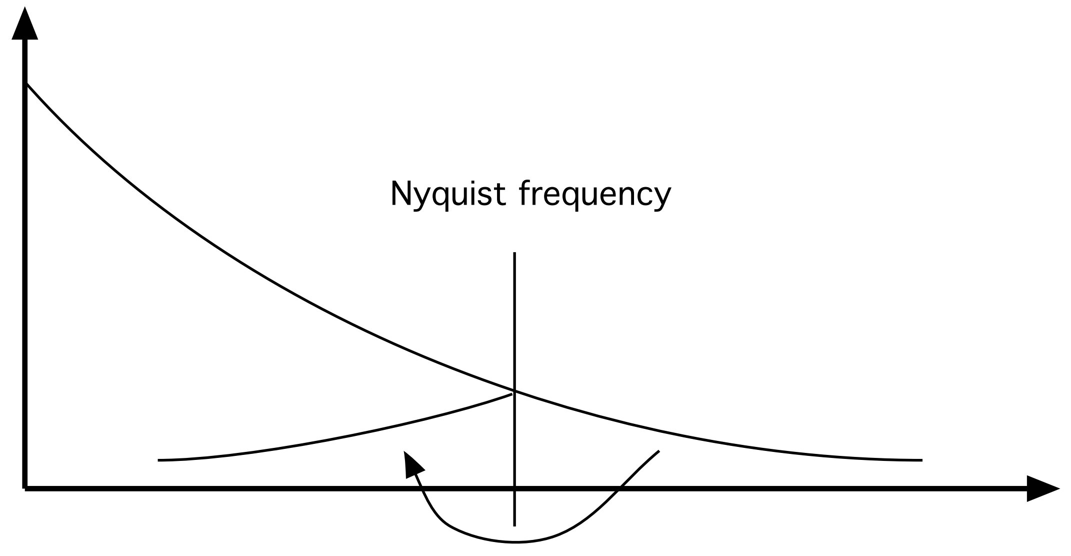
You can *not* lower your points! Best result counts!

Think in frequency space!

Which frequencies are preserved and which cause problems?

Amplitudes usually lower for higher frequencies!



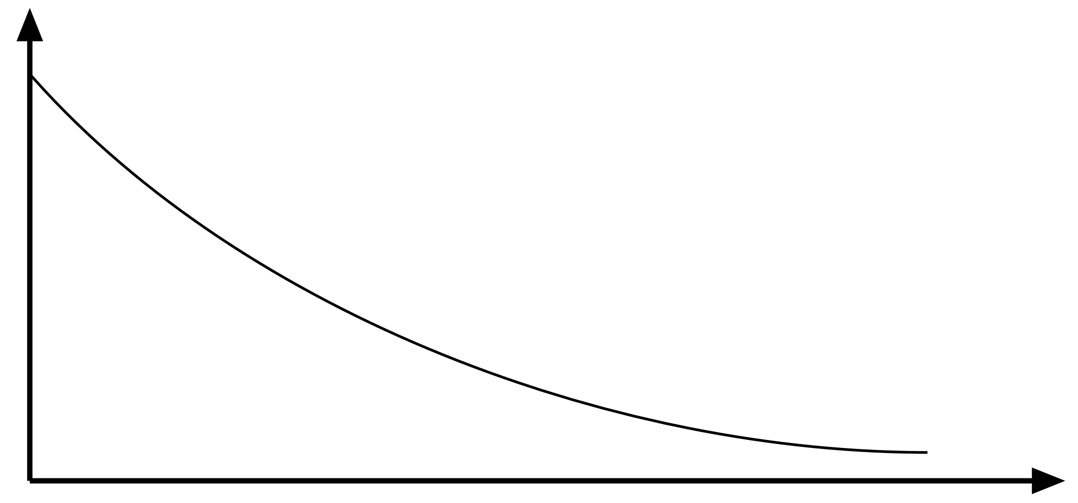


Aliasing caused by frequencies mirrored over the Nyquist frequency!

The 1/f rule

Natural images have, approximately, a frequency contents that vary by 1/f

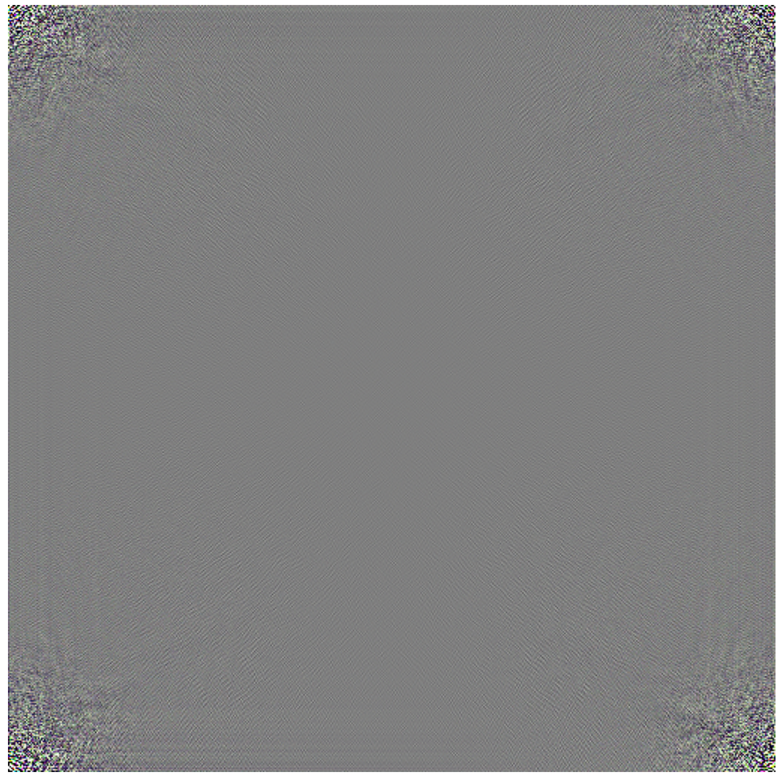
We will see this coming back later in the course.



Example: The "maskros" image

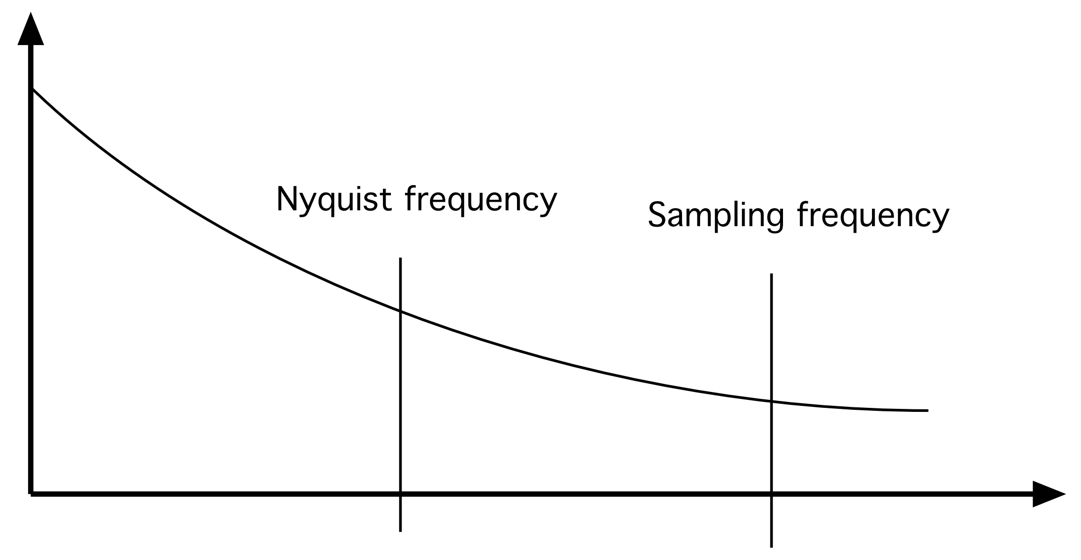
Natural images have, approximately, a frequency contents that vary by 1/f

We will see this coming back later in the course.



The Nyquist frequency

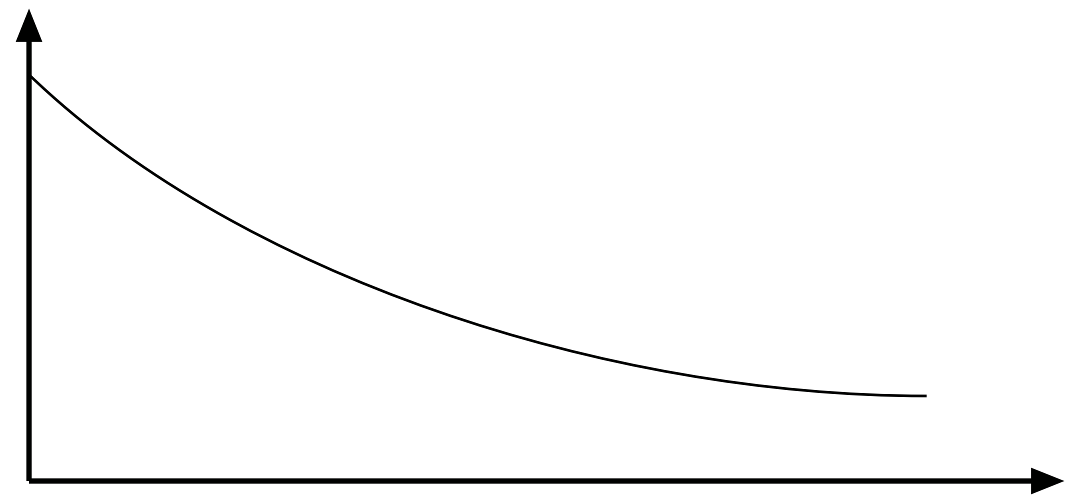
The signal is only correctly represented up to half the sampling frequency, N

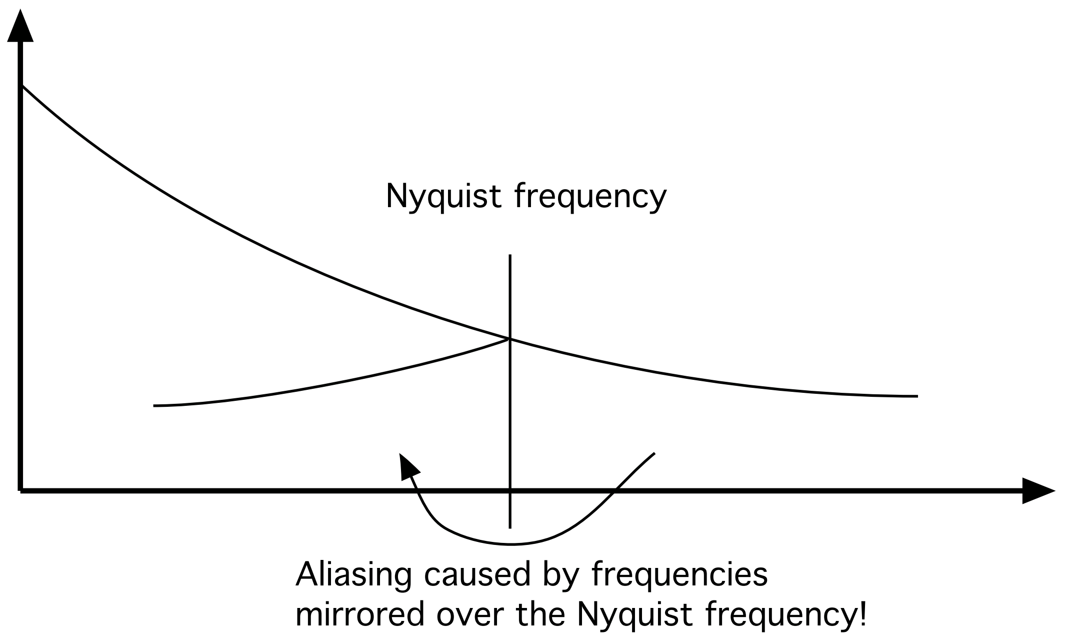


Folding over Nyquist

Errors have highest magnitude right above the Nyquist frequency. Errors are folded down into

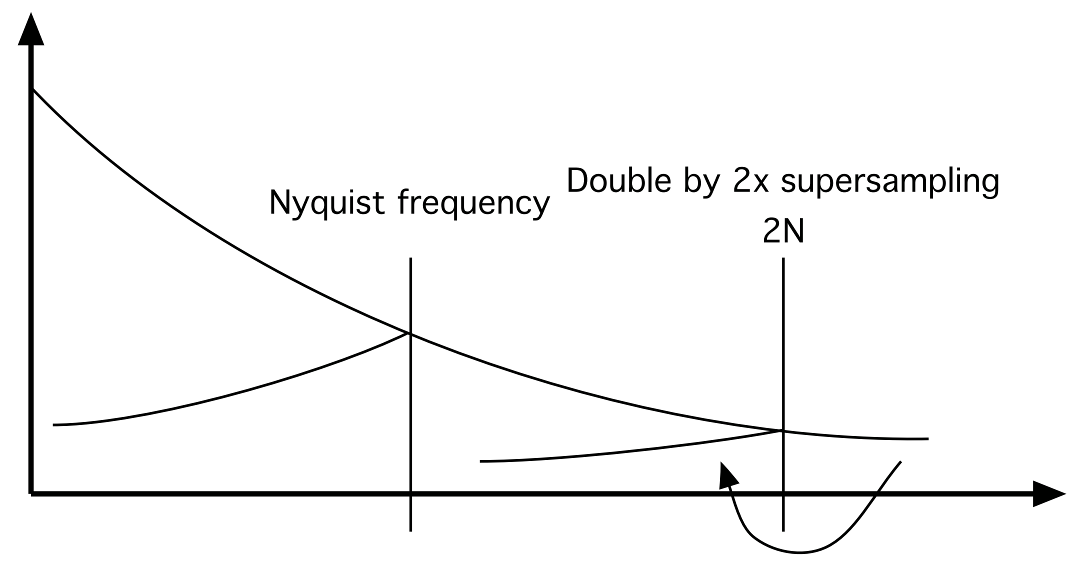
f' = -(f-N) + N = 2N-f





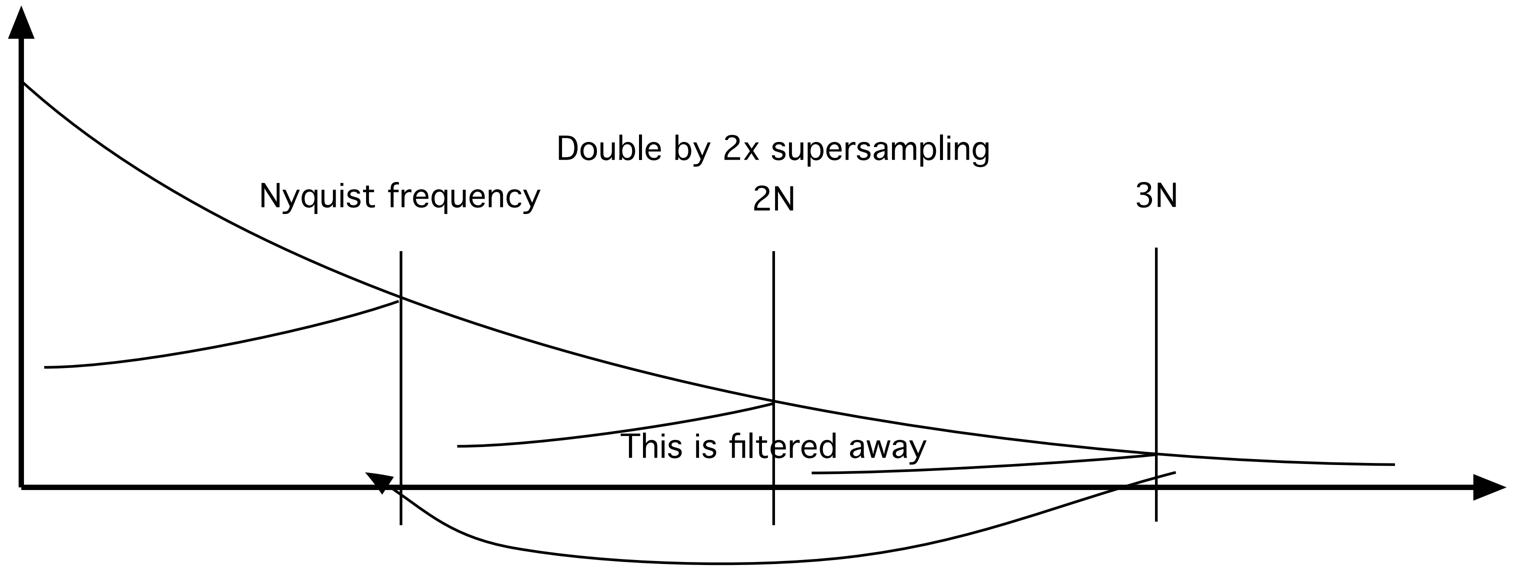
Relevance for supersampling

Double the sampling frequency = Half the error magnitude



Downsampling

But then you downsample! And things get even better! Approximately 1/3 of the magnitude!



FBM

The 1/f rule will become vital for us when doing
Fractal Brownian Motion (later lecture)

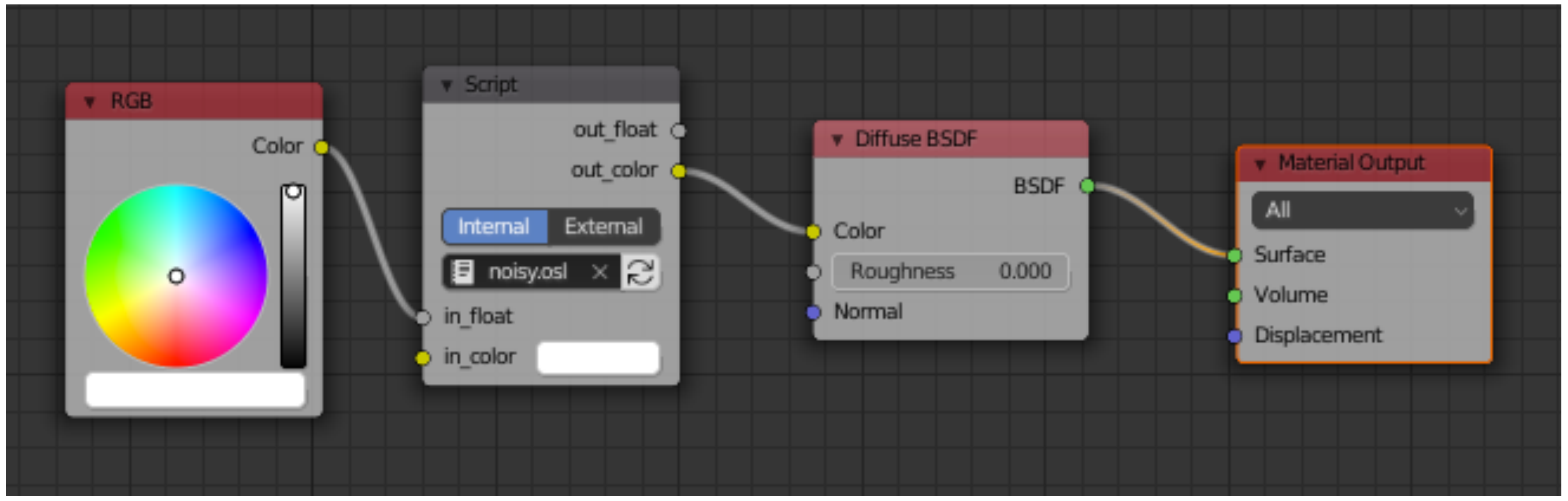
Blender nodes and rendering

Blender has a *node system* and multiple rendering engines.

You will usually use path tracing for high quality rendering.

Network of shader nodes, with a node editor

OSL shaders are part of a more or less complex network of nodes, out of some may be fixed, chosen from a set of pre-made nodes, while other may be OSL shader nodes



Based on "closures"

"OSL's surface and volume shaders compute an explicit symbolic description, called a "closure", of the way a surface or volume scatters light, in units of radiance.

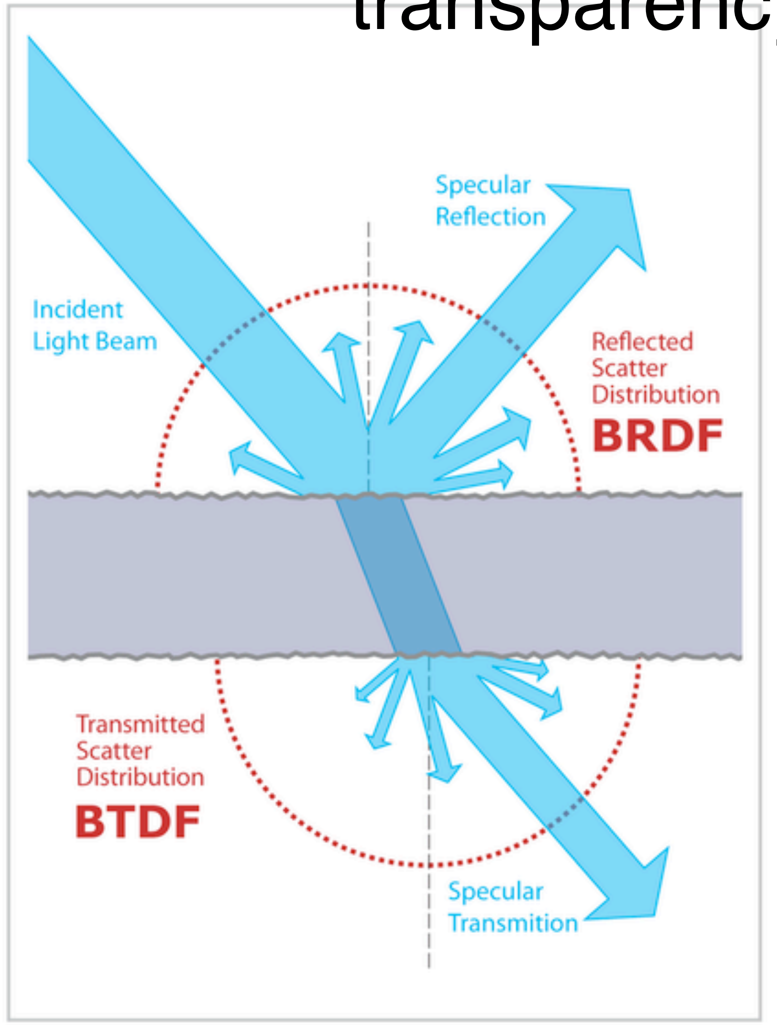
These radiance closures may be evaluated in particular directions, sampled to find important directions, or saved for later evaluation and re-evaluation"

In short: "closures" is about structures being passed between nodes, passing multiple variables for later rendering.

Don't worry to much on this. It is not vital for our work.

BSDF

OSL uses the BSDF model, Bidirectional scattering distribution function. This handles both reflections and transparency. Describes light exchange to any desired detail.



Combines BRDF and BTDF.

BRDF = reflectance

BTDF = transparency/ transmission

From Wikipedia

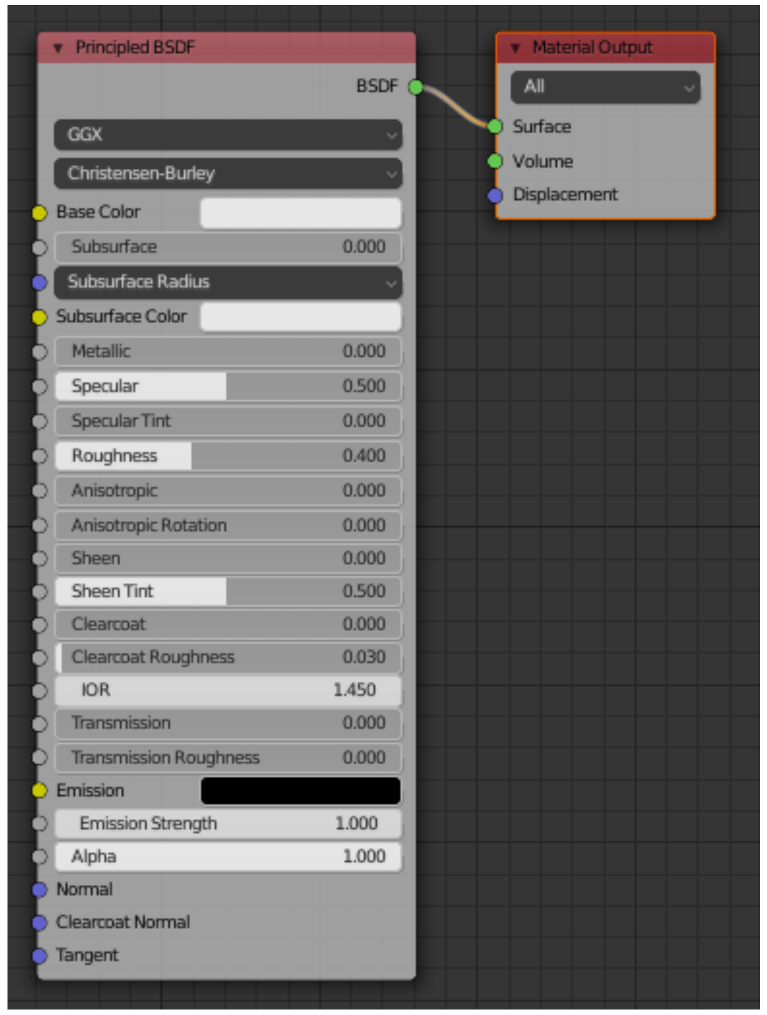
Light sources = emissive surfaces

Like in radiosity (for which we can consider the renderer to be a superset) there are no point light sources, at least not on shader level. You can make surfaces emit light.

The renderer is (generally) a form of *path tracer*, capable of handling global illumination.

Default node setup

When you start a new project, you get this structure for the cube.



Principled BSDF

BSDF

GGX

Christensen-Burley

Base Color

Subsurface 0.000

Subsurface Radius

Subsurface Color

Metallic 0.000

Specular 0.500

Specular Tint 0.000

Roughness 0.400

Anisotropic 0.000

Anisotropic Rotation 0.000

Sheen 0.000

Sheen Tint 0.500

Clearcoat 0.000

Clearcoat Roughness 0.030

IOR 1.450

Transmission 0.000

Transmission Roughness 0.000

Emission

Emission Strength 1.000

Alpha 1.000

Normal

Clearcoat Normal

Tangent

Material Output

All

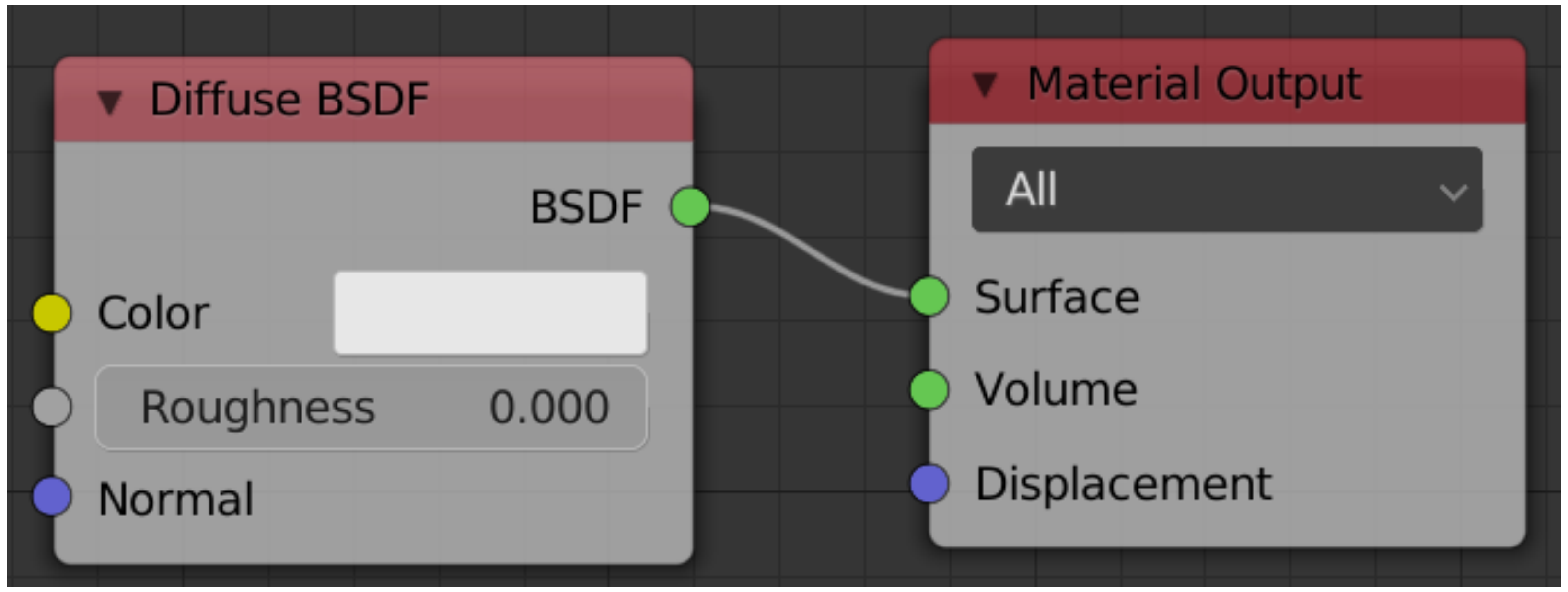
Surface

Volume

Displacement

Light calculations

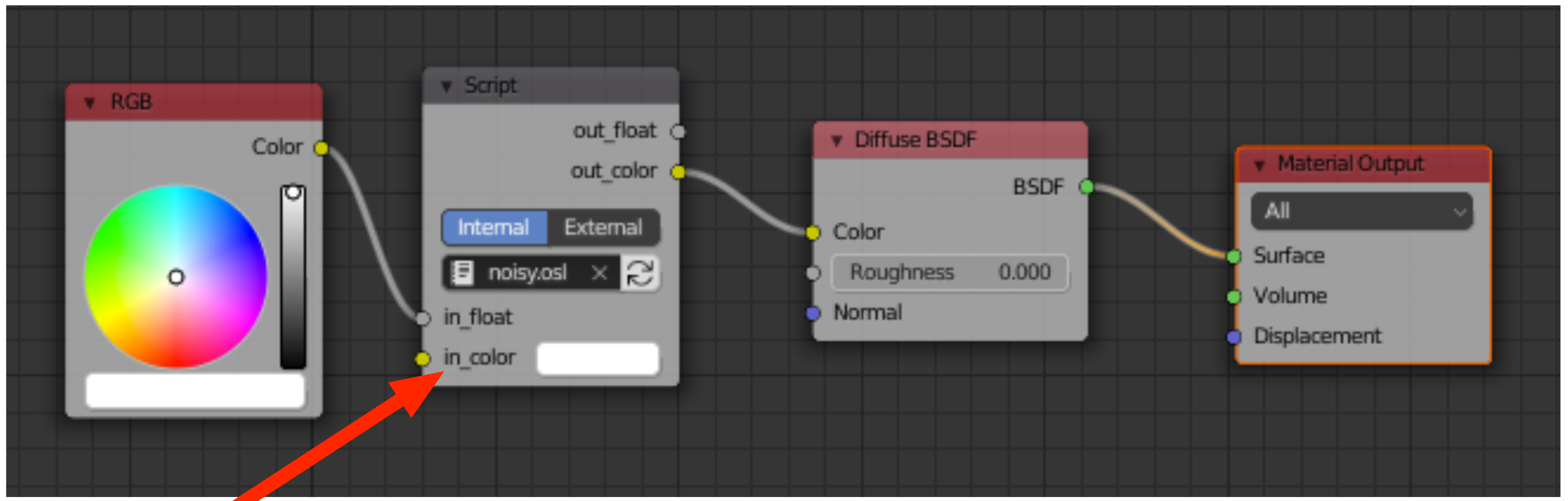
You generally do not need to think about lighting when writing a texture shader. The light can be left for other parts. All you need to do is to create the surface, possibly with normal vectors, and leave the lighting to other parts.



Simple lighting using the Diffuse BSDF shader node (not OSL)

Script nodes

A node can also be a "script node", which means a node that defines its function by an OSL shader



Script node!

Let's see what we can do:

Open Shading Language

Developed by Sony Pictures Imageworks

Supported by several renderers including

Cycles in Blender

Arnold in 3DSMAX

Has been used for CG effects in movies

Open Shading Language (OSL)

No fixed pipeline

Not tightly bound to hardware

Often implemented in CPU = not fast

Based on Renderman Shading Language

Found in off-line rendering packages like Blender and 3DSMAX

Built-in functions

Conveniently for our purposes, there are many built-in functions in OSL including noise functions. We will be using these.

Many functions are also hard-coded in specific node types. We will use these too, but not only these.

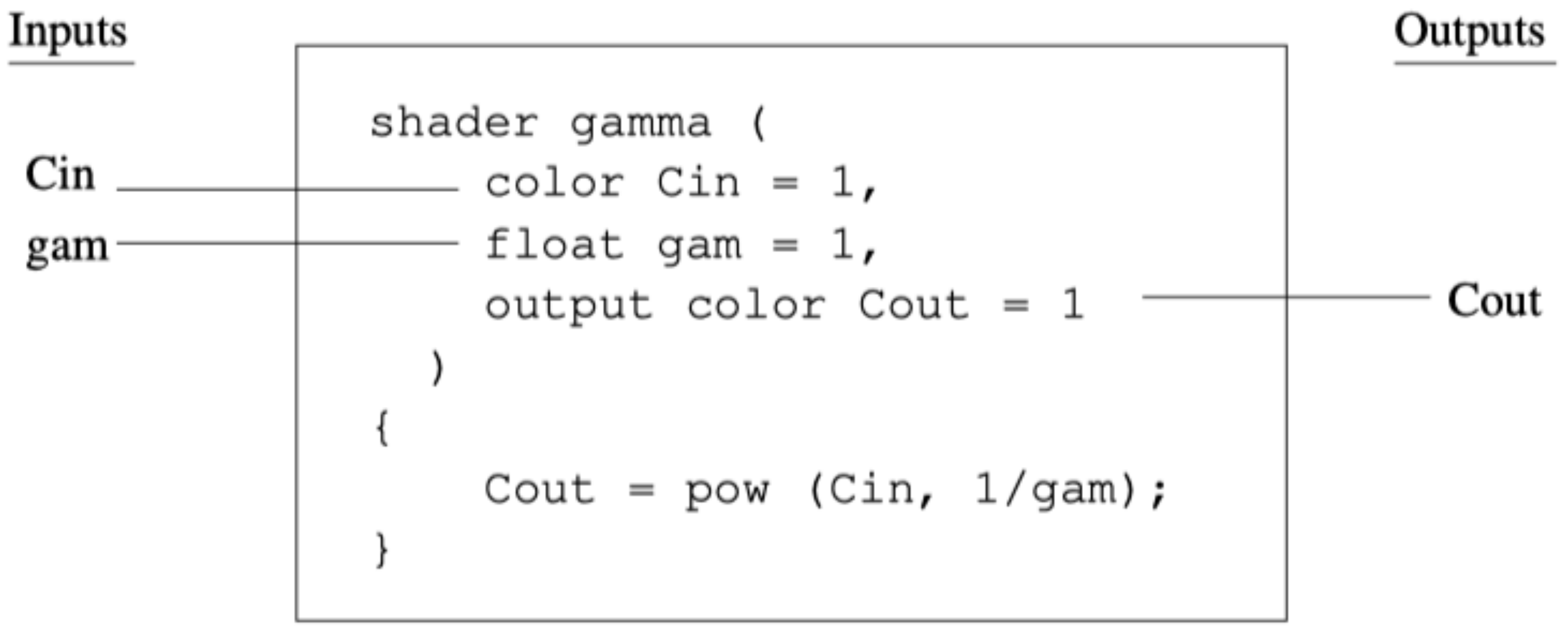
A shader is written for a single task

With the node system, it is easy to separate tasks into separate shaders.

Example shader from the docs

Takes a color and a gamma value in.

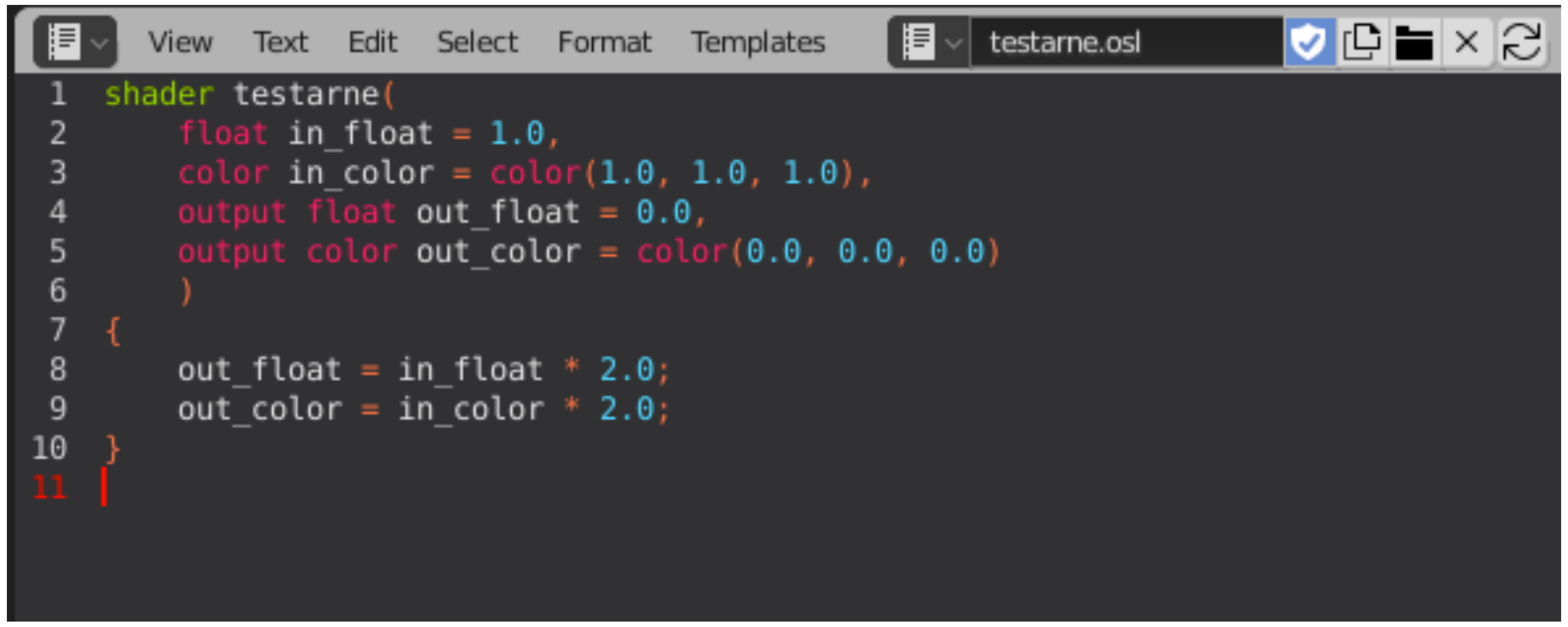
Sends gamma adjusted colors out.



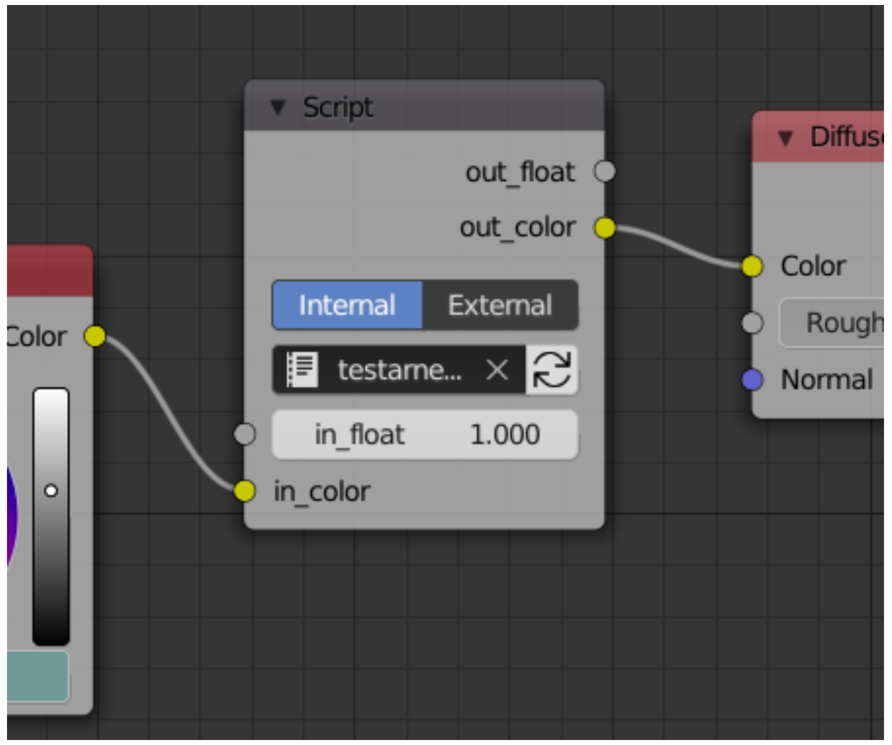
```
shader gamma (
    color Cin = 1,
    float gam = 1,
    output color Cout = 1
  )
{
    Cout = pow (Cin, 1/gam);
}
```

Input and output points in the node "box" is defined by the input and output variables in your code!

Points for point-and-click connecting appear as the code is saved/compiled.



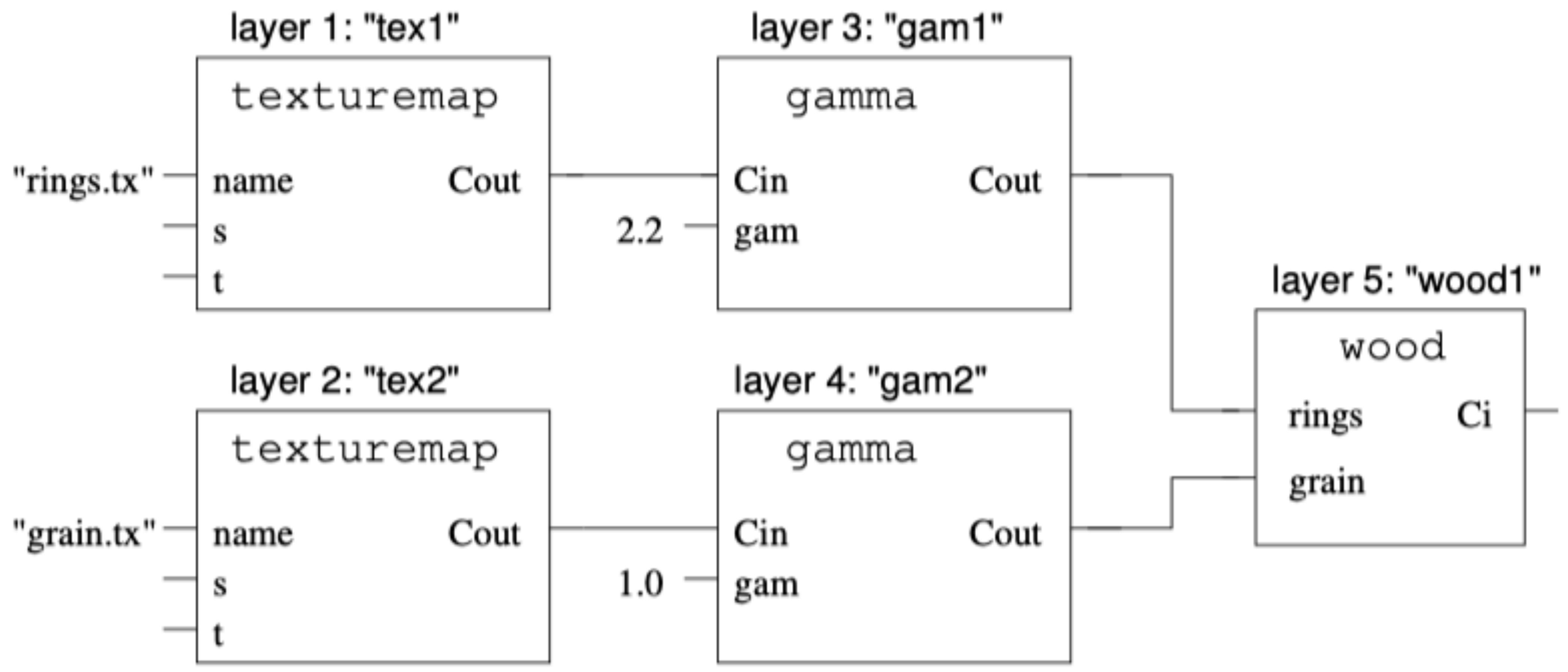
```
shader testarne(
    float in_float = 1.0,
    color in_color = color(1.0, 1.0, 1.0),
    output float out_float = 0.0,
    output color out_color = color(0.0, 0.0, 0.0)
    )
{
    out_float = in_float * 2.0;
    out_color = in_color * 2.0;
}
```



Node groups

Multiple nodes, OSL shader or not, are connected.

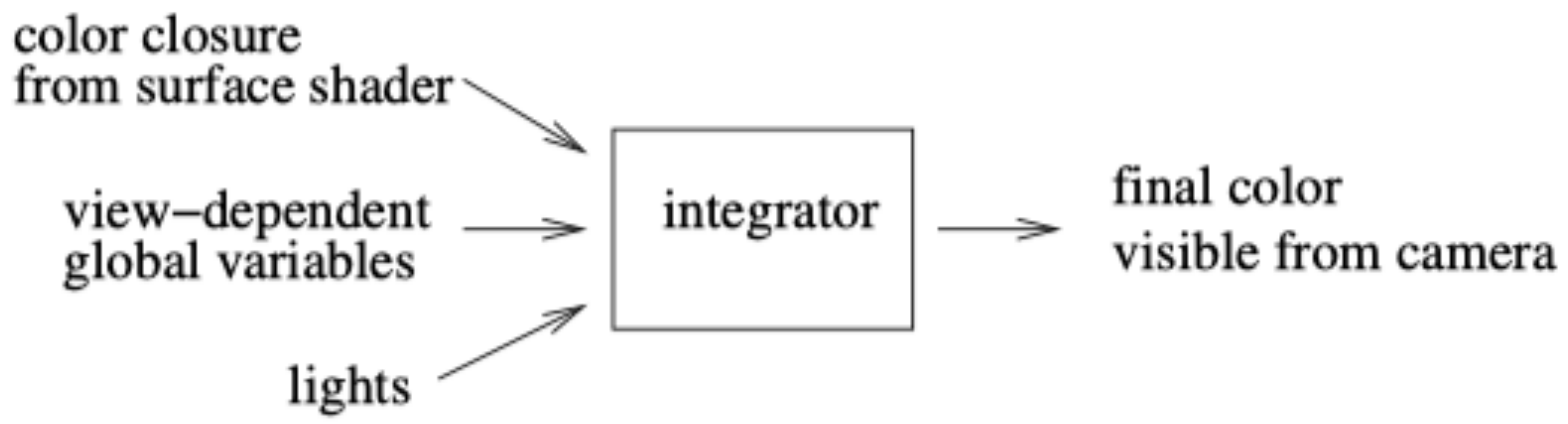
You make connections to define the data paths.



Integrators

The final stage of the rendering is the integrator, where the different parts are merged to a final output. The integrator is not your job, you only feed it your data.

You feed your result to "Material output". Then you let the renderer do the rest.


color closure from surface shader
view–dependent global variables
lights
integrator
final color visible from camera

Language

Much is business as usual.

Alphanumerics for identifiers. Comments as in C and GLSL.

Much more reserved words than in GLSL:

and break closure color continue do else emit float for if illuminance illuminate int matrix normal not or output point public return string struct vector void while

and some that are not yet used but still reserved:

bool case catch char class const delete default double enum extern false friend goto inline long new operator private protected short signed sizeof static switch template this throw true try typedef uniform union unsigned varying virtual volatile

Preprocessor

A lot more than GLSL:

#define #undef #if #ifdef #ifndef #elif #else #endif #include
#pragma once

and version numbers:

OSL_VERSION_MAJOR OSL_VERSION_MINOR OSL_VERSION_PATCH
OSL_VERSION

More like ordinary C than GLSL.

Overall syntax:

```
optional-function-or-struct-declarations

shader-type shader-name ( optional-parameters )
{
   statements
}
```

You may notice how we declare the shader type as part of the code.

Existing types:

surface, displacement, light, volume, shader (means generic shader)

Some operations are only available to specific types.

Surface shaders

Compute the surface behavior, most specifically its color, thereby also other ways it reacts to light.

It can also emit light.

Can not alter the position of the surface.

Similar to a fragment shader in GLSL.

Displacement shaders

Displacement shaders alter the position and shading normal (or, optionally, just the shading normal) to make a piece of geometry appear deformed, wrinkled, or bumpy.

They are the only kind of shader that is allowed to alter a primitive's position.

Similar to vertex shaders, or geometry shaders in GLSL.

Volume shaders

Volume shaders describe how a participating medium (air, smoke, glass, etc.) reacts to light and affects the appearance of objects on the other side of the medium.

They are similar to surface shaders, except that they may be called from positions that do not lie upon (and are not necessarily associated with) any particular primitive.

Generic shaders, "shader"

Generic shaders are generic routines that may be called as individual layers in a shader group.

Generic shaders need not specify a shader type, and therefore may be reused from inside surface, displacement, or volume shader groups.

They may not contain any functionality specific to some other type (for example, they may not alter P, which can only be done from within a displacement shader).

Shader parameters

Like function arguments.

Must have an *initializer*, giving a default value for the parameter

Syntax for a single parameter:

```
type parametername = default-expression
```

Multiple parameters may be defined, separated by commas.

Parameters may include one-dimensional arrays as well as structures.

Parameters get values in several ways

• Connected to an earlier stage value

• If name/type matches a primitive variable of the geometry being shaded, the parameter value will be computed, possibly interpolated, from this value. (varying!)

• There may be an *instance value*, giving a parameter an explicit per-instance value at the time that the renderer referenced the shader

• If none of these are present, the default value given as initializer is used.

Metadata

Shaders can define metadata, information not used for the rendering but for passing information to the user or host program about the shader.

Example: A string telling the UI a name to use for the shader.

This is one case where shaders have a use for text. (Another comes up soon.)

Not a major thing when learning OSL but something that you may find in existing ones.

Data types

int	
float	
point vector normal	Like vec3
color	
matrix	Always 4x4
string	
void	

Creating colors

Colors can be created in various formats. All will be stored as RGB.

```
color (0, 0, 0) // black
color ("rgb", .75, .5, .5) // pinkish
color ("hsv", .2, .5, .63) // specify in HSV space
color (0.5) // Same as color(0.5,0.5,0.5)
```

3-component RGB, no RGBA

Separate colors channels are accessed as an array, e.g. color[2]

Colors can be added, scaled, compared...

Matrices

Matrices are also arrays.

```
matrix zero = 0; // matrix with all 0 components
matrix ident = 1;  // identity matrix

// Construct a matrix from 16 floats
matrix m = matrix (m00, m01, m02, m03,
                   m10, m11, m12, m13,
                   m20, m21, m22, m23,
                   m30, m31, m32, m33);
```

The matrix is accessed in a 2-dimensional way:

```
     matrix M;
float x = M[row][col];
   M[row][col] = 1;
```

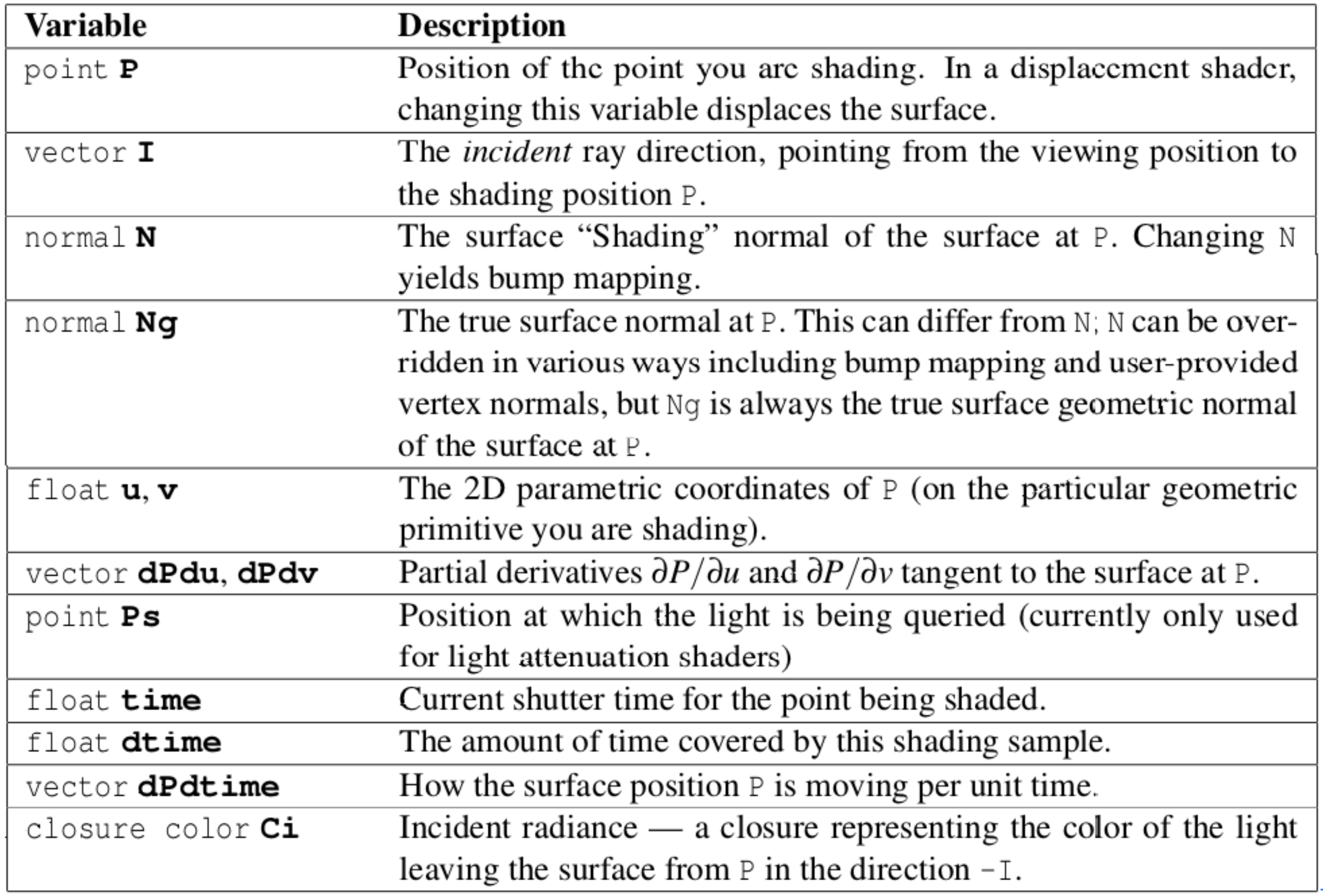
Strings

Strings are, for the purpose of rendering, mainly used for *file names for textures*.

In GLSL, we handle that in the main program, but we don't write our own main program here so we must be able to ask the system to find the files for us.

Inside OSL strings are also used for specifying variants of things like color format and noise type.

Global variables



Variable	Description
`point` **P**	Position of thc point you arc shading. In a displaccment shadcr, changing this variable displaces the surface.
`vector` **I**	The *incident* ray direction, pointing from the viewing position to the shading position `P`.
`normal` **N**	The surface "Shading" normal of the surface at `P`. Changing `N` yields bump mapping.
`normal` **Ng**	The true surface normal at `P`. This can differ from `N`; `N` can be overridden in various ways including bump mapping and user-provided vertex normals, but `Ng` is always the true surface geometric normal of the surface at `P`.
`float` **u, v**	The 2D parametric coordinates of `P` (on the particular geometric primitive you are shading).
`vector` **dPdu, dPdv**	Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ tangent to the surface at `P`.
`point` **Ps**	Position at which the light is being queried (currently only used for light attenuation shaders)
`float` **time**	Current shutter time for the point being shaded.
`float` **dtime**	The amount of time covered by this shading sample.
`vector` **dPdtime**	How the surface position `P` is moving per unit time.
`closure color` **Ci**	Incident radiance — a closure representing the color of the light leaving the surface from `P` in the direction `-I`.

Interesting global variables

P: Position. Can be useful, especially combined with N.

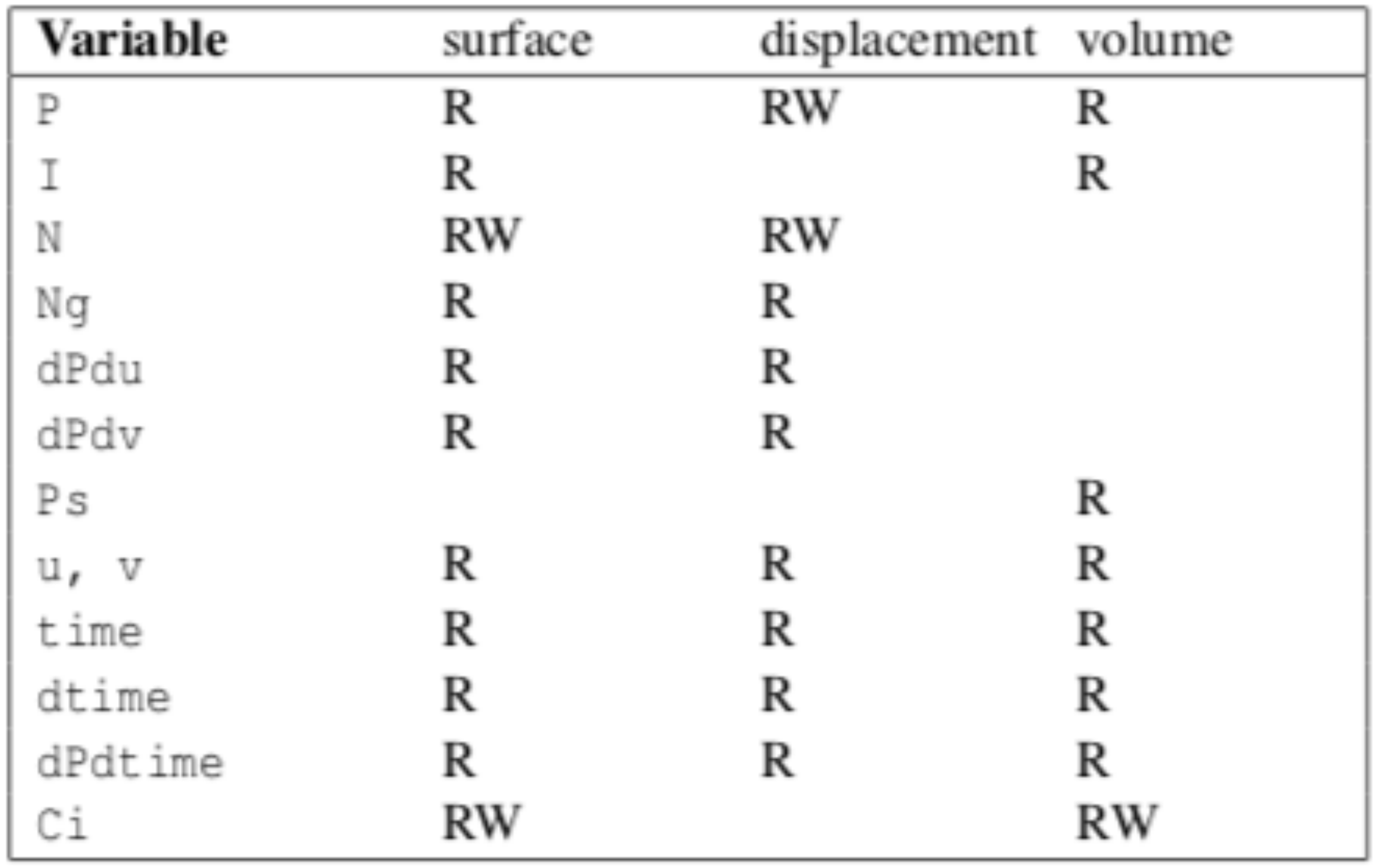
I: Incident ray direction. No...

N, Ng: Normal vector. Valuable!

u, v: Primitive local coordinates (triangle edges?). No.

These should take you pretty far.

Accessibility



Variable	surface	displacement	volume
P	R	RW	R
I	R		R
N	RW	RW	
Ng	R	R	
dPdu	R	R	
dPdv	R	R	
Ps			R
u, v	R	R	R
time	R	R	R
dtime	R	R	R
dPdtime	R	R	R
Ci	RW		RW

Table 6.2: Accessibility of variables by shader type

I believe this table says a lot about what you can do from each shader type! (Except generic shaders.)

Library functions

Much of the standard stuff, sqrt, sin, cos, floor, fract, round, min, max, also the less used clamp, mix...

Geometric functions: Constructors for point etc, dot, cross, length (that is norm), normalize

distance, reflect, refract, rotate

Matrices: transpose, constructors

Library functions for pattern generation

Here we can see that OSL has a lot built-in:

step, linearstep, smoothstep

noise (for several types)

pnoise (for several types), periodic noise

If you skip the type parameter, there is noise, snoise, pnoise, psnoise, cellnoise for specific noise types. These calls look deprecated AFAIK.

aastep is included

Noise functions

Functions get argument for choosing noise algorithm:

type noise (string noisetype, float u, ...)
type noise (string noisetype, float u, float v, ...) type noise (string noisetype, point p, ...)
type noise (string noisetype, point p, float t, ...)

"perlin", "snoise"
"uperlin", "noise"
"cell"
"hash"
"simplex", "usimplex"
"gabor"

Amazing richness in built-in noise functions!

Example shader with noise

P is a good source for basic input.

```
shader basic_shader(
    float in_float = 1.0,
    output color out_color = color(0.0, 0.0, 0.0)
    )
{
    out_color = noise(P * in_float * 10.0);
}
```

More for us

aastep for anti-aliasing

displace

Displace surface along the normal

bump

Adjust the normal by some amount

Textures

You may also want textures. You can get the texture coordinated from a special node, the "texture coordinate" node. From that you take the "UV" vector.

For using an existing texture from file, use a texture->image texture node and input the image from the node.

Not that vital to us but expected

3D textures

Point cloud operations

All in all

A lot of stuff pre-defined

Focus on the basic types, noise generation for surface textures, and, for the later parts of the lab, modifying geometry

Start here

One generic OSL shader

One input, one output.

Simple node chain, feed to a BSF node and output

P, N, texture coordinate node

noise()

GLSL vs OSL

GLSL runs on GPU. OSL only on CPU so far.

GLSL is made for real-time. OSL for offline rendering.

GLSL is controlled from your program, can be integrated in any application.

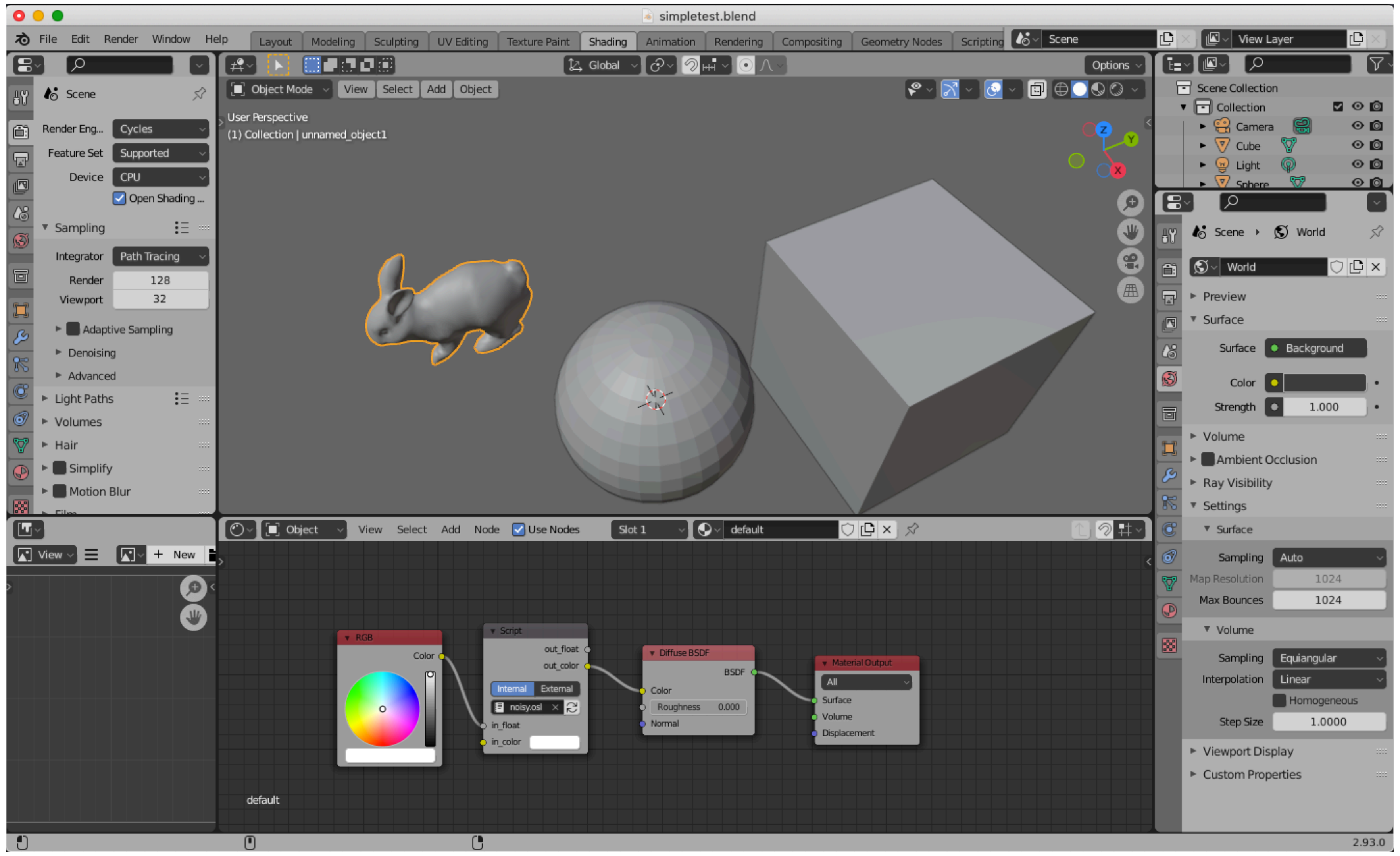
OSL comes integrated in a modeller/renderer software.

GLSL renders specific stages in a pre-defined pipeline. OSL renders stages in a custom node-based structure.

GLSL lighting must be in your fragment shader. In OSL you can separate texturing and light etc.

Live demo

Let's see if I can show how it works...



Scene

Render Engine: Cycles
Feature Set: Supported
Device: CPU
✓ Open Shading ...

Sampling
Integrator: Path Tracing
Render: 128
Viewport: 32
Adaptive Sampling
Denoising
Advanced
Light Paths
Volumes
Hair
Simplify
Motion Blur

Scene › World
World
Preview
Surface
Surface: Background
Color
Strength: 1.000
Volume
Ambient Occlusion
Ray Visibility
Settings
Surface
Sampling: Auto
Map Resolution: 1024
Max Bounces: 1024
Volume
Sampling: Equiangular
Interpolation: Linear
Homogeneous
Step Size: 1.0000
Viewport Display
Custom Properties

Note: I use Blender 2.93.0. Don't use much older versions. Blender had a major GUI overhaul a few versions ago.

A few Blender hints

Every window has a selection in the upper left corner. You will mostly use "3D viewport" and "Properties".

Tabs on the top: You will mainly use Shading and Scripting.

To manipulate models:

G move
R rotate
S scale
X remove
Z select rendering mode

Scrollwheel in the lab!

Move camera with two-finger drag + shift/ctrl

Nothing stops you from "Modelling" and "Sculpting" but not for the lab

Lab material

A few models are provided, two versions of the bunny model (one was not compatible with Blender - now corrected) and the Utah Teapot, plus a simple example shader.

You will do most work directly in Blender or 3DSMAX.

Thank you for your attention!

See you at the lab on monday!