# OpenGL Shading Language Course

# Chapter 3 – Basic Shaders

By
Jacobo Rodriguez Villar
jacobo.Rodriguez@typhoonlabs.com

# CHAPTER 3: Basic Shaders

## INDEX

## INTRODUCTION

In this chapter we will explain some basic shaders, showing the basic operations of the OpenGL Shading Language and how to achieve some simple effects. We'll also cover the access to to OpenGL states from shaders, and we'll show how the state values can be set from the OpenGL Shader Designer IDE. We will present a detailed step-by-step guide on both shader construction and useage of the Shader Designer IDE.

First we will focus on the syntax of GLSL and the operations used. The shaders we will make are mostly implementations of simple algorithms. In the next chapter, "Advanced Shaders," we will focus on more complex algorithms, so knowing the basic GLSL syntax and the use of operations will serve as a pre-requisite for that chapter.

## First Basic Shaders

This section consists of two step-by-step guides to implement two basic shaders using uniforms and varying variables.

### Simple Shader 1

This shader is the simplest shader that we are able to write; that is, a mesh will be drawn with a plain color and we will play with the vertex shader parameters.

First, create an empty project called "**simple1**." This action will create the .gdp file containing all OpenGL states that the shader needs. Also, two empty files will be created and opened by the editor automatically and will have the same name as the project tile, but their extensions will be .vert and .frag. They will hold the source code for the vertex and the fragment shader.

Upon project generation, the Shader Designer automatically generates the main body functions for both the vertex and the fragment shader. They are (as discussed earlier) named main, they don't have parameters, and the return type is **`void`**.

We will compute the homogeneous vertex coordinates and the fragment color in the first shader as follows:

*[Vertex shader]*

```
void main()
{
  gl_Position = ftransform();
}
```

As said before, ftransform() computes the homogenous vertex coordinates in clip-space keeping the invariance intact  (useful for multipass shaders). We could do **gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;** and the result would be the same, but the invariance could not be guaranteed.

**Important note**: **gl_ModelViewProjectionMatrix * gl_Vertex IS NOT EQUAL TO gl_Vertex * gl_ModelViewProjectionMatrix**. Remember this from linear algebra: matrices multiplications are not commutative.

[*Fragment shader*]

```
void main()
{
 gl_FragColor = vec4(1,1,1,1);
}
```

This function is fairly straightforward; it only assigns a pure white color to the fragments. This would be the result after clicking F4 key (compile) with the torus selected as preview mesh. For compilation results, see the build log at the bottom of the IDE.

This is a very simple and boring shader; so let's juice it up a bit.

First, create two uniform variables:

- First variable: call it "**scale**" with type = **vec4** and with values = 1,1,1,1. Assign to it the slider widget and set a minimum value of 0.5 and a maximum value of 3.
- Second variable: call it "**color**" with type = **vec4** and with values = 1,1,1,1. Assign to it the color slider widget.

Go to the vertex shader and modify the shader like this:

```
uniform vec4 scale;
void main()
{
  vec4 pos = gl_Vertex * scale;
  gl_Position = gl_ModelViewProjectionMatrix * pos;
}
```

We have changed the **ftransform()** line for these two because we want scale the vertex in world coordinates **(vec4 pos = gl_Vertex * scale**;) and then compute the position in clip-space (**gl_Position = gl_ModelViewProjectionMatrix * pos;).**

Go to the fragment shader and modify it as well:

```
uniform vec4 color;
void main()
{
  gl_FragColor = color;
}
```

First we will play with the color. Right-click the uniform list window, highlighting the color variable. A pop-up menu will appear with several choices. Select "Open Widget." This should bring up the color widget dialog. As you adjust the sliders you should notice how the mesh color changes in real time. The color value is updated as it is changed in the widget.

Now open the scale widget and try adjusting the values. You can change the scale of each dimension individually. After some modification the mesh can end up looking like this (yes, it is still a teapot):

## Simple Shader 2: Using Varying Variables

As said before, varying variables (sometimes called interpolators) are variables that are written in the vertex shader and read in the fragment shader with a perspective-correct interpolation. The following is a basic example of how to use a varying variable, along with a shader that uses it to get a real effect.

First, create a new project with the Shader Designer and call it “varyings.gdp.” Then code the vertex shader:

[*Vertex shader*]

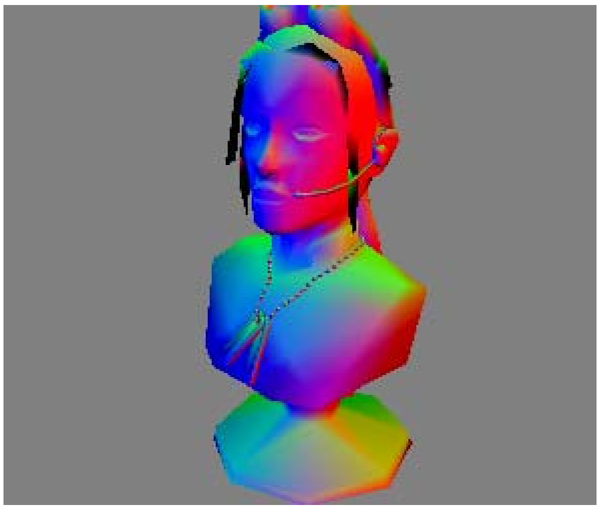
```
varying vec3 vertex_color;
void main()
{
  gl_Position = ftransform();
  vertex_color = gl_Vertex.xyz; // Example of how to use swizzling.
  // This is the same as vertex_color = vec3(gl_Vertex);
}
```

[*Fragment shader*]

```
varying vec3 vertex_color;
void main()
{
  gl_FragColor = vec4(vertex_color,1.0);  // Example of how to use a constructor.
}
```

We are storing the vertex color value from the current vertex in the varying, and reading it back in the fragment shader. The value we read is interpolated across each primitive (usually, at each triangle). As a result of this operation we get the famous RGB cube. Here is a screenshot of the shader applied to the “busto” mesh:

As a variation of this shader, you can also “see the normals” encoded as RGB colors. Just replace **gl_Vertex** with **gl_Normal** in the vertex shader:



This shader teaches a few things. First, do you notice how colors are smoothly interpolated? This is proof that varying variables actually does interpolate the values. Second, since the vertex positions aren't normalized, shouldn't this shader produce some unexpected results? Actually, no. GLSL clamps the **gl_FragColor** values to the range [0,1].

## Using Built-in Uniforms and Attributes

It's now time to learn how to use the built-In attributes and OpenGL states. Here we will use material properties from the property grid, and combine these values to get a material (without lighting for now).

First, go to the property grid (default placement is at the right edge of the IDE main window) and choose colors for the front material fields. For example, here we choose a pure red color (255, 0, 0) for Front Material Ambient and pure green color (0, 255, 0) for the diffuse component.

The following vertex shader source shows how to access the diffuse component of the front material, and uses it as value for **gl_FrontColor**:

[*Vertex shader*]

```
void main()
{
  gl_FrontColor = gl_FrontMaterial.diffuse;
  gl_Position = ftransform();
}
```

We are computing the vertex position, but most important in this example, we are assigning the green color the vertex color. We will perform some modifications to this vertex shader later.

The fragment shader only sets the color of the current fragment to the color of the vertex.

[*Fragment shader*]

```
void main()
{
   gl_FragColor = gl_Color;
}
```

The values in **gl_Color** and **gl_SecondaryColor** will be derived automatically by the system from **gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor**, and **gl_BackSecondaryColor** based on which face is visible.

In this case, it will hold a pure green color because we are assigning the same color to each vertex. If we had assigned a different color to **gl_FrontColor** we would see a smooth interpolation across the primitive (as in the previous shader).

Let's do a simple combination of the two built-in uniforms modified:

[*Vertex shader*]

```
void main()
{
  gl_FrontColor = gl_FrontMaterial.diffuse + gl_FrontMaterial.ambient;
  gl_Position = ftransform();
}
```

This will produce a yellow mesh, because (255,0,0) + (0,255,0) = (255,255,0) (yellow).

This shader isn't really useful, as it does not produce any effects worth talking about, but it does show how to access the built-in states in OpenGL. Setting up light states and properties is very similar to using the states and properties for materials.

As a final modification for this shader, let's add some noise to the final color. At this point, the knowledge of how Perlin Noise works is assumed, but just in case, here is a brief explanation:

Perlin Noise consists of an efficient computation of pseudo-random values. These values are continuous between one value and its neighbors, and for a starting seed, Perlin always generates the same values. There are many ways of getting Perlin Noise in the fragment shader; one of them is by using a texture that holds the noise values, while another is using the built-in **noise**[1|2|3|4] functions. We will be using the latter (**notice**: not all cards supports this kind of noise, through a built-in function).

Now, let's get dirt on our teapot.

Modify the vertex shader by adding this line:

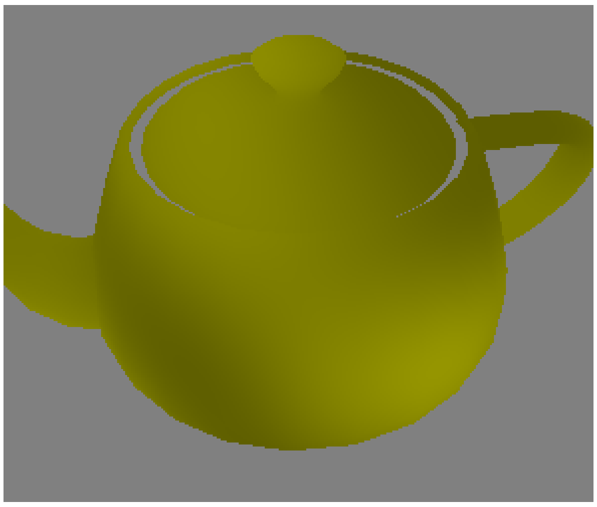
[*Vertex shader*]

```
varying vec3 position;
void main()
{
  position = gl_Vertex.xyz;
  gl_FrontColor = gl_FrontMaterial.diffuse + gl_FrontMaterial.ambient;
  gl_Position = ftransform();
}
```

[*Fragment shader*]

```
varying vec3 position;
void main()
{
  vec2 col = (noise2(position.xy) * 0.5) + 0.5;
  // Noise returns values in the range [-1,1], we need map them to [0,1].
  float val = (col.x+col.y ) / 2.0;
  // Convert the noise values to a ‘fake’ greyscale.
  gl_FragColor = gl_Color * vec4(val, val, val, 1);
  gl_FragColor.a = 1.0; // We do not want a transparent teapot.
}
```

Our mesh color will be modulated by the noise value after we’ve converted it to greyscale. This is the result after modulation:



Many 3D cards does not support noise functions, so if you do not get this shader working, don’t worry, check your compilation log to see if there are any ‘noise’ unknown symbols (or something like that)

## Texturing and Multitexturing (Vertex Attributes)

It’s time to take a peek at some more useful features. We discussed samplers earlier, and now take a closer look at them and how to access a texture trough a sampler. Procedural generated shaders are very interesting, and we will take a closer look at them later, along with some more complex ones.

First is texture access in the fragment shader. The following explains how to load a texture with Shader Designer in order to have it available for the shader.

Go to the textures dialog, and select ‘1’ in the “Texture units” spin button (the topmost of the dialog). This will enable the first texture unit for us. Select GL_TEXTURE_2D in the target field, GL_LINEAR for both Min and Mag filters, and GL_REPEAT for both clamp fields (wrap_S and wrap_T). For an OpenGL programmer, those texture settings are very common, so it should not be a problem to select the appropriate settings. Now we need to choose the texture file. Once you have all texture parameters filled, press the Refresh Textures button. If all goes well, you should see a thumbnail preview of the texture that was loaded. If you don’t see it, it’s probably caused by a unsupported texture format (Shader Designer allows .TGA, .BMP, .JPG, .PNG, and many more file formats, using RGB, RGBA, BGR, BGRA, or LUMINANCE pixel formats).

When done, click Accept, and the texture will be ready to use in the shader. Now, let’s write our first texturing shader.

Add a new uniform variable (you should already know how to do this). Call it tex, and give a value of 0 and a type of **int**. We use zero as value because our texture is placed in the texture unit zero.

We will use two methods to pass the texture coordinates. First we will use a custom interpolator (a varying variable), and then a built-in interpolator.

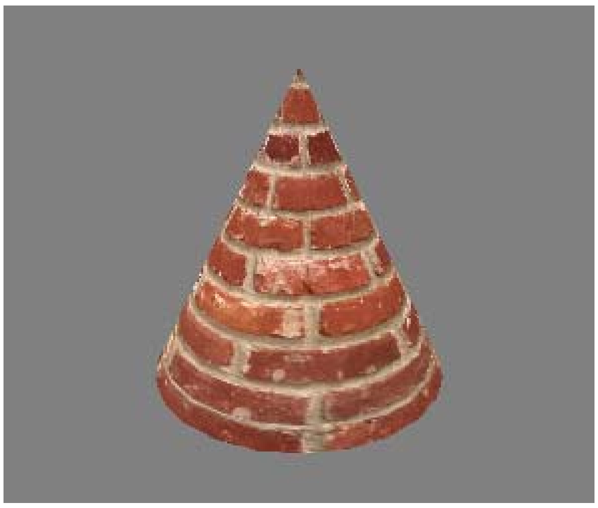
[*Vertex shader*]

```
varying vec2 texCoords;
void main()
{
  gl_Position = ftransform();
  texCoords = gl_MultiTexCoord0.st;
//  This  attribute  holds  the  texture  coordinates  issued  with  OpenGL
glTexCoord2 calls for the texture unit 0.
}
```

[*Fragment shader*]

```
varying vec2 texCoords;
uniform sampler2D tex;
void main()
{
  gl_FragColor = texture2D(tex, texCoords);
}
```

Press F4, and you should see something like this (depending on the texture chosen):



Here are the shaders using the built-in texture coordinates interpolator:

[*Vertex shader*]

```
void main()
{
  gl_Position = ftransform();
  gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

[*Fragment shader*]

```
uniform sampler2D tex;
void main()
{
  gl_FragColor = texture2D(tex, gl_TexCoord[0].st);
}
```

This will produce exactly the same result as the previous method, but using the built-in texture interpolators is considered the standard.

With all of this still fresh in our heads, let’s get started on multi-texturing. As an OpenGL programmer, you should know that the only way to combine textures without shaders is through blending and multiple passes (though limited by the blending modes), or by using the texture combiner. With shaders, you can combine the textures any way you wish. Let’s see some examples.

First, we need more than one texture, so add another one with the textures dialog and add another uniform (name tex2, type = **int**, value = 1).

Because we only need one UV set (and the Shader Designer only provides one UV set per mesh), we can use the same texture coordinates for the two textures. However, let's make the shader a bit more complex. For texture ‘tex,’ use the same UV set as the last example, but for texture ‘tex2,’ modify the UV set. The modification is a simple scale by a uniform value: add another uniform called 'scale' and give it a type of vec2 and a value of 1,1 (give it a custom widget with a min = 0 and max = 4). Then modify the shader like this:

[*Vertex shader*]

```
uniform vec2 scale;
void main()
{
  gl_Position = ftransform();
  gl_TexCoord[0].st = gl_MultiTexCoord0.st;
  gl_TexCoord[1].st = gl_MultiTexCoord0.st * scale;
}
```

[*Fragment shader*]

```
uniform sampler2D tex;
uniform sampler2D tex2;
void main()
{
  vec4 value1 = texture2D(tex, gl_TexCoord[0].st);
  vec4 value2 = texture2D(tex2, gl_TexCoord[1].st);
  gl_FragColor = value1 + value2;
}
```

Open the ‘scale’ widget and play around a bit with it. You should be able to see how the texture is stretched/enlarged in both directions:

As you can see, We used a brick texture and a fire texture, and both are added.

The line **gl_FragColor = value1 + value2**; is the core of the shader; here you can do whatever you want, including addition, subtraction, interpolation, bump mapping, combining only certain texture components, etc.

Before we end this example, let’s throw in another effect. We’ll animate the fire texture based on time from init by using some predefined uniforms that Shader Designer provides. Let’s implement a simple animation in the horizontal axis.

We only need to modify the vertex shader because we only need to modify the texture coordinates.

First of all, delete the ‘scale’ uniform, since we don’t need it anymore. Then go to the uniform dialog and select the TIME_FROM_INIT predefined variable from the list. This variable increases its value in each frame, so it is perfect for our purposes.

[*Vertex shader*]

```
//Uniform vec2 scale:
uniform int TIME_FROM_INIT;
void main()
{
  gl_Position = ftransform();
  gl_TexCoord[0].st = gl_MultiTexCoord0.st;
  float offset = float(TIME_FROM_INIT) * 0.001;
  gl_TexCoord[1].t = gl_MultiTexCoord0.t;
  gl_TexCoord[1].s = gl_MultiTexCoord0.s + offset;
}
```

The reduction of the TIME_FROM_INIT is needed because this variable holds the milliseconds elapsed since the program starts, which is a ,large number to add to the texture coordinates. As an idea, you can set up a uniform to control the speed of the animation, but do not use fixed values like 0.001.

At this point you know all about texturing with GLSL. You can use textures 1D, 2D, 3D, and cubemaps, using the textures dialog and the texture access built-in functions.

## Discard Shader and Subroutines

In certain situations we will have fragments that we want discarded; that is, we don't want them to be updated in the frame buffer. For the color buffer we can use the alpha test or alpha blending, but the other buffers will be updated (stencil, depth, etc.) unless we use discard. Using **discard** we can save some processing power by returning to an early stage of our shader, avoiding later computations, and giving our meshes a "transparency" on many parts.

In this example we will draw a yellowish grid based on the texture coordinates to discard some fragments. We will also introduce the use of subroutines.

Our vertex shader only needs to compute the vertex position and the texture coordinates. In order to be homogeneous and follow the standard, we multiply the texture coordinates by the right texture matrix. By doing this we can be sure that any changes made to the texture matrix in the host application are included in the computations in the shader.

**NOTE:** This step is only to show how to access the texture matrix (for academic purposes) and if you don't need it (because your host doesn't use the texture matrix), this step can be skipped.

[*Vertex shader*]

```
void main()
{
  gl_TexCoord[0]  = gl_TextureMatrix[0] * gl_MultiTexCoord0;
  gl_Position     = ftransform();
}
```

Our fragment shader has more code this time. We need to know when to discard the fragment, so we need to do a test, for which we'll use the "if" statement. Since we will be discarding fragments based on their texture coordinates, we must decide when we will discard them. For this example, this criterion is when the second decimal cipher is greater than a given threshold. To achieve this, we modify the texture coordinates by scaling them by 10.0 and then drop the integer part with the built-in **fract** function.

To introduce the use of subroutines, we'll move the vector comparison into a new function in a separate source file. Click on the 'New Fragment Shader" toolbar button and write this:

[*fp1.frag Fragment shader*]

```
bool greater(in vec2 v1, in vec2 v2)
{
  return all(greaterThan(v1,v2));
}
```

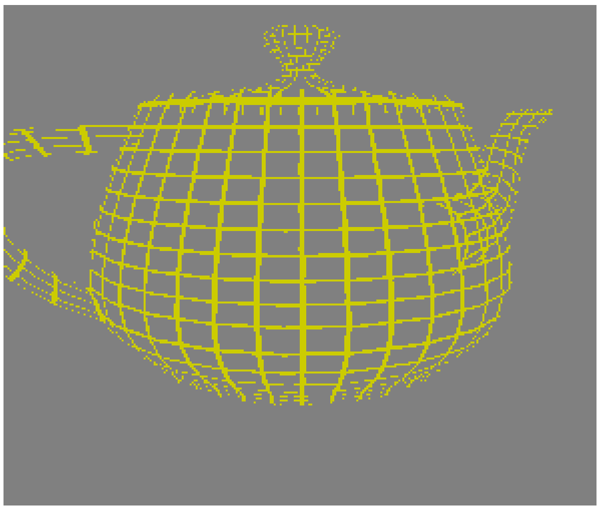
Like in C, a function must be declared before using it, so we have to write its prototype (aka: declare it).

[*Fragment shader*]

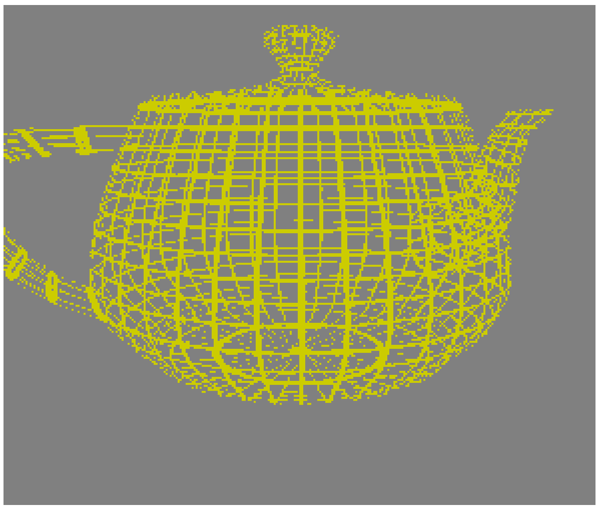
```
uniform vec2 threshold;
uniform vec3 color;
bool greater(in vec2 v1, in vec2 v2);

void main()
{
   vec2 ss = fract(gl_TexCoord[0].st * 10.0);
    if (greater(ss, threshold))
      discard;
    gl_FragColor = vec4 (color, 1.0);
}
```

As you can see, we’ve setup two uniform variables, one for the mesh color and one for the threshold. Give an initial value for last one of (0.13,0.13) and use a custom widget with minimum limit of zero and maximum limit of 1, then, play with it.

This shader should produce an output like this:



Notice that the lines of the back side are not shown. To see them, disable the backface culling in the vertex states panel. It will then look something like this:

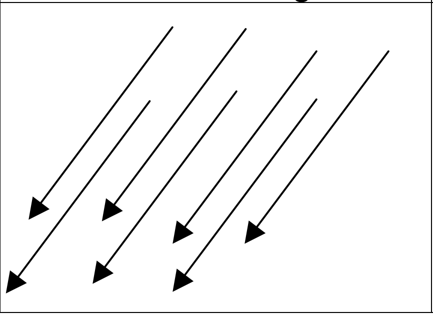
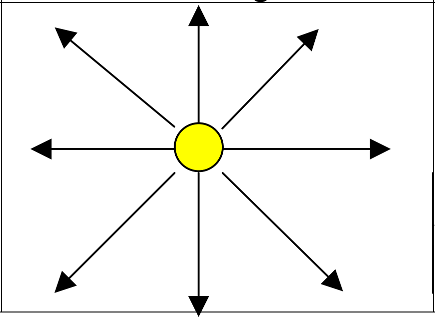
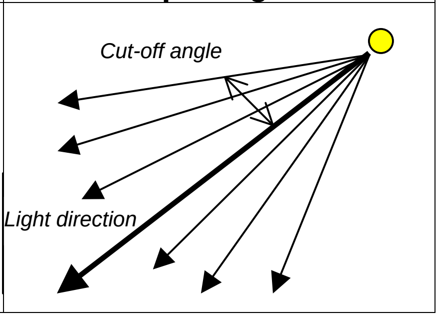


## Simple Illumination Model

It is time to add lights to our shaders. This topic requires a bit of math knowledge, in terms of how to build a fast illumination model as close as possible to its physical behavior. We'll go through the basic models and then build more sophisticated models (bump mapping for example) later on.

At this point we assume that you are familiar with how OpenGL lights works, the usual OpenGL lighting models, and the different lights that OpenGL provides. In this shader we will implement Goraud (per-vertex) lights and later extend it to per-pixel lighting.

There are three types of lights: Directional Light, Point Light, and Spot Light. We will implement only the first two of them for now.

| Directional Light | Point Light | Spot Light |
|---|---|---|
|  |  |  |

**Note**: Because this is a basic example, we are not concerned about surface material or specular components. We only use a texture as a base material.

In this case I will show the fragment shader first, because it will be used along the following lighting examples.

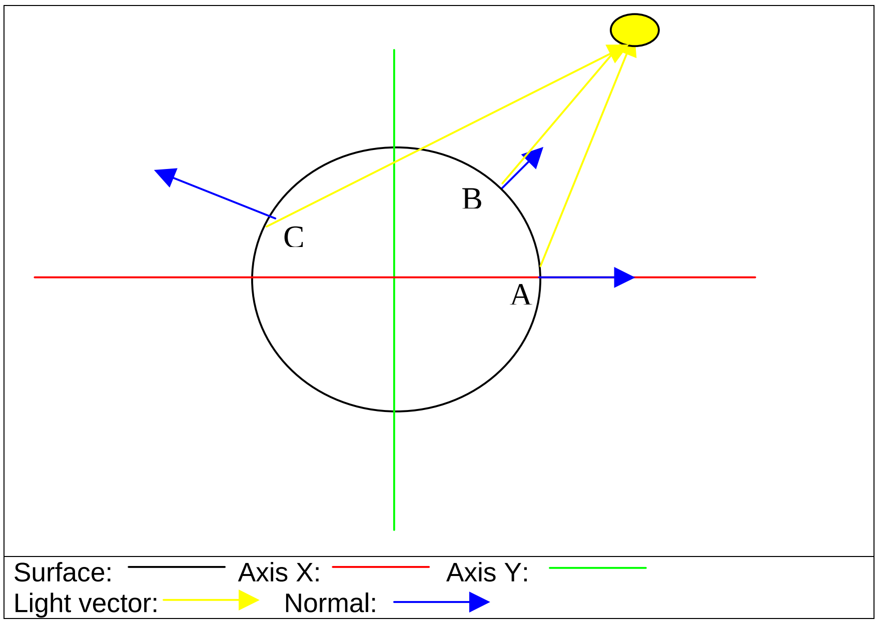
[*Fragment shader*]

```
uniform sampler2D texture;
varying vec4 diffuse;

void main()
{
  vec4 texColor = texture2D(texture, gl_TexCoord[0].st);
  gl_FragColor = (gl_LightSource[0].ambient + diffuse) * texColor;
}
```

This is a very simple method (but not the best method) for implementing lighting, but for our test it is good enough. In the vertex shader we will compute the diffuse light contribution, depending on the type of light and the light parameters, at each vertex. This value will be interpolated and passed to the fragment shader. There, the texture will be modulated by the light contribution.

What is the light's diffuse component? Light can be reflected by a surface in a variety of ways. One of these ways is the diffuse reflection.

The value of the light's diffuse component increases as the light vector and the normal decrease their angle, reaching its maximum when they are parallel and minimum when they are perpendicular.



Looking the graph:

- A: Angle between normal and light vector is less than 90º, so there should be light, but only with a little power, because the angle is close to 90º.
- B: Here the light must be maximum, because this point directly faces the light point (angle ~= 0º).
- C: Here there should not be any diffuse light contribution, because the angle is greater than 90º. This means that this point is not visible from the light point.

Our main goal here is finding a normalized value (range [0,1]) to modulate the diffuse color using the following: diffuse_contribution = diffuse_color * diffuse_power; where if diffuse_power = 0, black color is applied, and if diffuse_power = 1, diffuse color is applied.

The dot product is our friend. To clarify, the dot product is the scalar product between two vectors, and its formula (one of several you can find in any algebra book) is:

$$a \cdot b = |a| * |b| * \cos(\text{angle\_between\_vectors})$$

If a and b vectors are normalized, the dot product means:

$$a \cdot b = 1 * 1 * \cos(\text{angle});$$

The cosine function always return a value in the range [-1,+1].

The nxDir formula**: nxDir = max(0.0, dot(normal, lightVector));** means:

- nxDir will have a value > 0 if the angle between the vector is between 0º and 90º
- If the angle is greater than 90º, the max function will clamp the value to 0.

With this value we know in which way the diffuse contribution affects to a surface point.

Let’s move on to the directional light vertex shader.

```
varying vec4 diffuse;

void main()
{
  gl_Position = ftransform();
  gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;

  //vec3 normal = gl_NormalMatrix * gl_Normal; // Usually your lights won't be in
world space, so you have to multiply the normal by the normal matrix to have it in
world space.
  vec3 normal = gl_Normal; // Normals are expected to be normalized. If they are not,
do it here.
  vec3 lightVector = normalize(gl_LightSource[0].position.xyz);
  float nxDir = max(0.0, dot(normal, lightVector)); // Normal x light direction.

  diffuse = gl_LightSource[0].diffuse * nxDir;
}
```

Brief explanation:

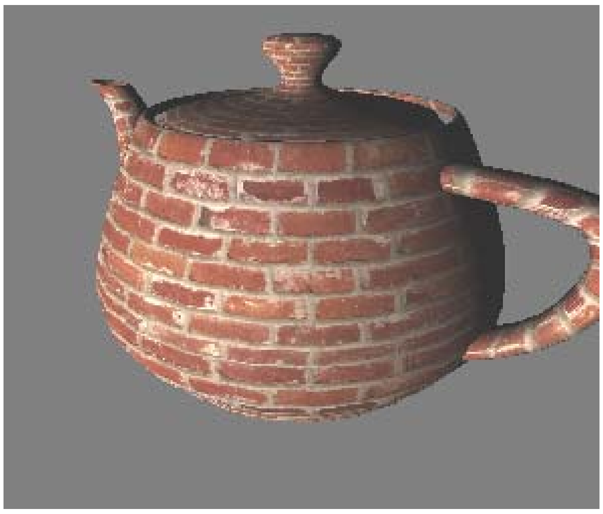
Directional lights rays are defined relative to the origin, so a directional light with a ‘position’ of xyz really means that the director light ray parts from xyz and goes to (0,0,0) (remember that all other rays are parallel to this one), so our light vector is xyz - (0,0,0). In other words, the light vector has the same value as the light position.

After the usual computations of the vertex position and texture coordinates, we need to know the vector and the vertex normal. Shader Designer provides meshes with smooth normals (per vertex, not per face) which are normalized, so renormalization is not necessary. **The light positions are given in world coordinates (that’s how Shader Designer works), so we do not need to do computations on those either. If they weren’t given in world space we would have to transform them by the normal matrix, since the object and the lights must be in the same space before making any computations. REMEMBER MULTIPLY LIGHTS BY THE NORMAL MATRIX BEFORE NORMALIZATION IF YOUR LIGHT POSITIONS DOESN’T COME IN WORLD COORDINATES. Remember this, because all following shaders in this course will receive the light position in world coordinates, but in your real application, it most likely wont happen.**

Ambient colors, diffuse colors, and position are filled with the values from the property grid (lighting states). Insert the value of (51,51,51) to the light0’s ambient component (51 → [0,1] = 0.2) and a yellowish color for the diffuse component (255,255,128). Place the light at (0,0,3) to give the direction for the light rays.

It is now time to compute the two light components that we are going to use (ambient and diffuse) and pass them to the fragment shader by using varying variables. Ambient is quite simple; just use the value filled in by the property grid with **gl_LightSource[0].ambient**. For the diffuse component we need do some computations. First the light direction: this is done with a dot product of the normal and light position in world coordinates.

The shader rendering should look like this:



Now let's see how point lights works, and implement them into a shader.

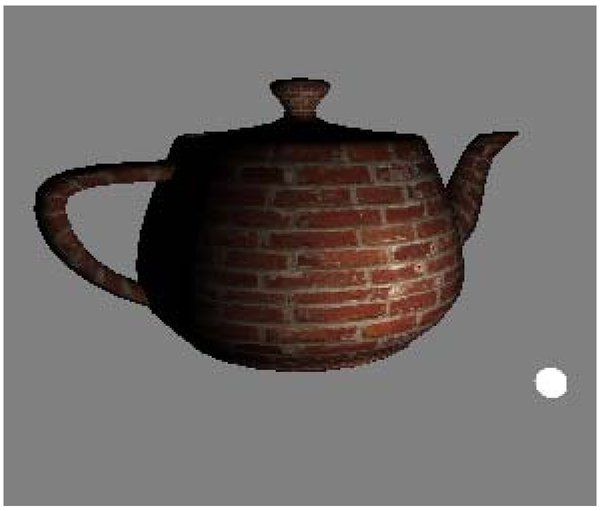
The vertex shader for our point light is this:

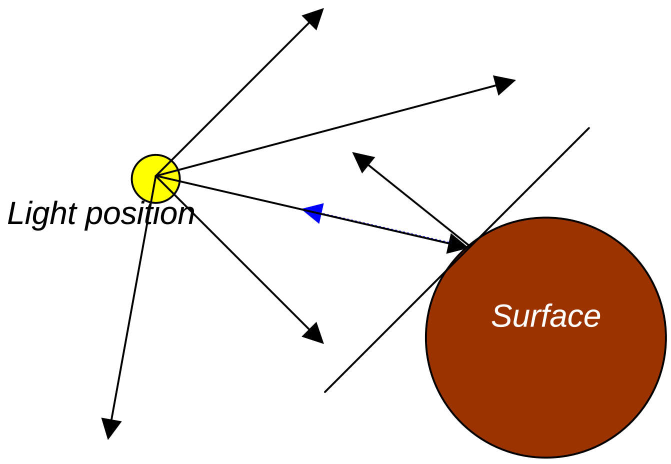
[*Vertex shader*]

```
varying vec4 diffuse;

void main()
{
  gl_Position = ftransform();
  gl_TexCoord[0] = gl_TextureMatrix[0]  * gl_MultiTexCoord0;
  /* vec3 normal = gl_NormalMatrix * gl_Normal; // Usually your lights won't be in world
space, so you have to multiply the normal by the normal matrix to have it in world space */
  vec3 normal = gl_Normal; // Normals are expected to be normalized. If they are not, do it
here.

  vec3 lightVector = gl_LightSource[0].position.xyz - gl_Vertex.xyz;
  float dist = length(lightVector);

  float attenuation = 1.0 / (gl_LightSource[0].constantAttenuation +
                             gl_LightSource[0].linearAttenuation    * dist +
                             gl_LightSource[0].quadraticAttenuation * dist * dist);

  lightVector = normalize(lightVector);
  float nxDir = max(0.0, dot(normal, lightVector)); // Normal x light direction.
  diffuse = gl_LightSource[0].diffuse * nxDir * attenuation;
}
```

The biggest differences between directional lights and points light are the attenuation computation and the light vector. Light vector is computed as the difference between the light position and the incident vertex position (since the light and the vertex are both in the same space, there is no need for any kind of transformation to another system). By doing it this way, we get radial vectors to the light position (remember that directional light rays are parallel, not radial).

This is a render with point light and the following parameters: ambient color = (0,0,0); diffuse color = (255,255,255); light position = (0,0,2); linear attenuation = 1, constant attenuation = 0. The rest of the light states are the default values.





## Simple Illumination Model 2 (Per Vertex Specular & Glossiness)

We’ve just taken a look at a basic illumination shader. It computed light and little more. It did not cover criteria like specularity and glossiness. Specular hightlights are an interesting feature, as they are view-dependent. Take a shiny object (a metal plate, for example) and hold it close to a lamp. Hold the plate still, but move your head while looking at it. Notice how the light highlight moves according to your head position. This highlight is called the “specular term” of a light, and depends on three factors: the light, the surface material, and the viewer position. This part was left out of the previous shader because it is a bit harder than the previous concepts. However, with ambient and diffuse understood, specular is easier to explain.

Before we start coding let's initialize a few things. First, add the texture “./textures/parallaxgloss.tga.” It has the base colors in RGB components, as well as a gloss map in the alpha channel. A gloss map is a map that holds only 0 or 1 (0, 255), and will help us decide what parts of the surface are dirty and won’t be affected by specular component. Remember to setup the uniform with the texture unit too. We’ll use the following light parameters: FrontMatShininess = 64, Ambient = (0,0,0), Diffuse = specular = (255,255,255).

Let’s start with the fragment shader.

*[Fragment shader*]

```
uniform sampler2D texture;
varying vec4 diffuse;
varying vec4 specular;

void main()
{
  vec4 texColor = texture2D(texture, gl_TexCoord[0].st);
  gl_FragColor =  gl_LightSource[0].ambient +
                  (diffuse * vec4(texColor.rgb,1.0)) +
                  (specular * texColor.a);
}
```

This is almost the same shader as in the last examples, but it receives the interpolated specular component, and the texture is divided in two parts: the RGB one and the gloss one (alpha component). This gloss might be multiplied by the specular factor in order to avoid it if the gloss map is 0 (we do not want specular contribution over dirty surfaces: gloss = 0 → dirty).

Now let’s start with the topic of this example: computing the specular contribution in the vertex shader.

Specular contribution works very similarly to diffuse, but with three differences: a) the specular strength depends on the surface material, not only on the light; b) specular is view–dependent, that is, it changes if the relative observer position to the observed object changes; and c) the specular reflection is very gathered at a reduced area of the surface.

Let's see how to factor in those three points:

> a: We will use material properties to compute the specular contribution (**gl_FrontMaterial.shininess**).
>
> b and c: Instead of using the light vector and the normal to see if there is contribution, we will use a new vector, formed by the camera vector and the light vector (half vector).

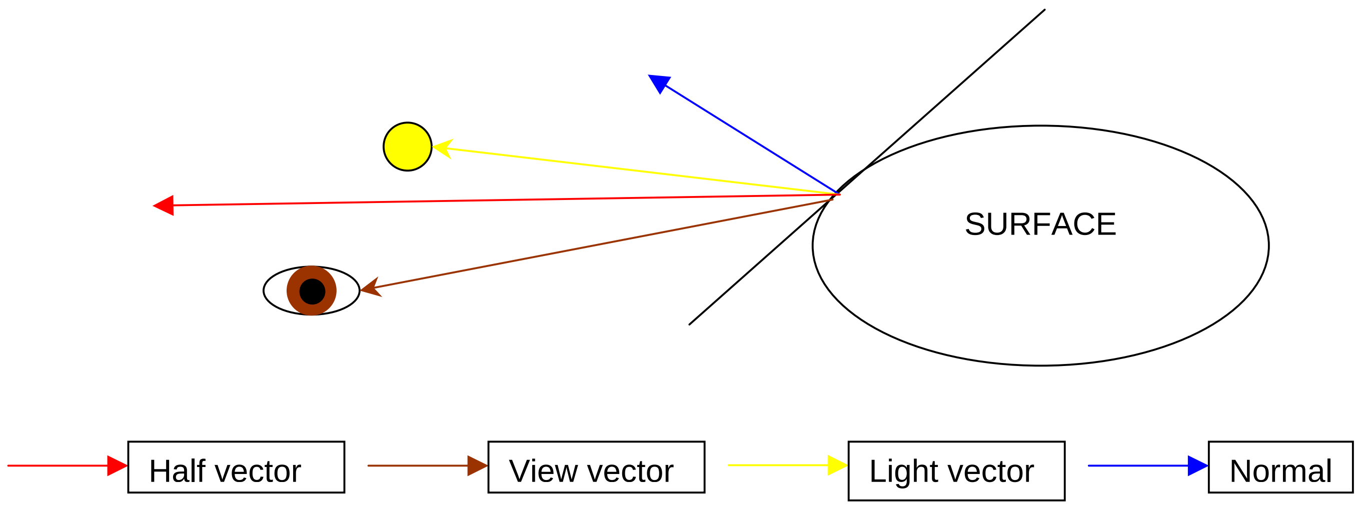
Because the specular contribution is small and gathered, we can’t use the measure given by the normal and the light vector. Instead, we will compute a vector that is in the middle of those two vectors: halfVector = normalize(cameravector + lightvector);. Using it in the max/dot formula,  we get our modulation value, but another step is needed. We also need to control how powerfull and gathered the specular reflection is, so we raise this value with a material property: shininess (the higher the shininess, the less the specular power). At the end we do not want specular contribution if the point is not viewable from the light point, so we will enclose all into a conditional

[*Vertex shader*]

```
varying vec4 diffuse;
varying vec4 specular;
uniform vec3 CAMERA_POSITION;

void main()
{
  gl_Position = ftransform();
  gl_TexCoord[0] = gl_TextureMatrix[0]  * gl_MultiTexCoord0;

  vec3 normal = gl_Normal;
  vec3 lightVector = gl_LightSource[0].position.xyz - gl_Vertex.xyz;
  float dist = length(lightVector);
  float attenuation = 1.0 / (gl_LightSource[0].constantAttenuation +
                            gl_LightSource[0].linearAttenuation       * dist +
                            gl_LightSource[0].quadraticAttenuation * dist *
dist);

  lightVector = normalize(lightVector);
  float nxDir = max(0.0, dot(normal, lightVector));
  diffuse = gl_LightSource[0].diffuse * nxDir * attenuation;

  //-----------------NEW CODE TO COMPUTE SPECULAR TERM----------------
  //-----------------NEW CODE TO COMPUTE SPECULAR TERM----------------
  float specularPower = 0.0;
  if(nxDir != 0.0)
  {
    // Programatic way
    //vec3 cameraPosition = vec3(gl_ModelViewMatrixInverse * vec4(0,0,0,1.0));
    //cameraPosition = normalize(cameraPosition - gl_Vertex.xyz);
    vec3 cameraVector = normalize(CAMERA_POSITION - gl_Vertex.xyz);
    vec3 halfVector = normalize(lightVector + cameraVector);
    float nxHalf = max(0.0,dot(normal, halfVector));
    specularPower = pow(nxHalf, gl_FrontMaterial.shininess);
  }
  specular = gl_LightSource[0].specular* specularPower * attenuation;
}
```

This shader uses the basecode from the last shader, and adds some new lines at the end to compute the specular contribution. The first noticeable change is the addition of a predefined uniform variable, the camera position (in world coordinates). We need it to compute the camera vector and the half vector, which are both needed for the specular reflection. We can also compute the camera position programmatically. Any point in view space can be multiplied by the inverse of the modelview matrix to obtain its position in world space, and the camera position in view space is (0,0,0), so if we multiply (0,0,0) by the inverse of modelview, we get the camera position in world space as well.

The specular illumination model needs three vectors: one that goes from the light to the surface, one that goes from the camera to the vertex, and a vector called halfVector (the normalized sum of the other two). Let's see a figure:



We need to know all those vectors. LightVector have been computed in earlier shaders and it is there already there for us, but we don't know the others:

```
vec3 cameraVector = normalize(CAMERA_POSITION - gl_Vertex.xyz);
vec3 halfVector = normalize(lightVector + cameraVector);
```

The half vector is crucial because it provides us with useful information. For example, it gives us the size of the highlight according to the camera and light positions. We obtain this information this way:

```
float nxHalf = max(0.0,dot(normal, halfVector));
float specularPower = 0.0;
if(nxDir != 0.0)
   specularPower = pow(nxHalf, gl_FrontMaterial.shininess);
specular = gl_LightSource[0].specular * specularPower * attenuation;
```

The nxHalf variable has the same meaning as nxDir: it informs us of the direction and power of the contribution (more than 90º with the normal = no specular at all), depending on the surface (the normal) and the 'lightVector' (for specular we use the halfVector, not the real lightVector).

To clarify, let's say the dot product is the scalar product between two vectors, and its formula (one of them) is this:

$$a \cdot b = |a| * |b| * \cos(\text{angle\_between\_vectors})$$

If a and b vectors are normalized, the dot product means:

$$a \cdot b = 1 * 1 * \cos(\text{angle});$$

The cosine function always returns a value in the range [-1,+1].

The nxDir formula: **nxDir = max(0.0, dot(normal, lightVector));** means:

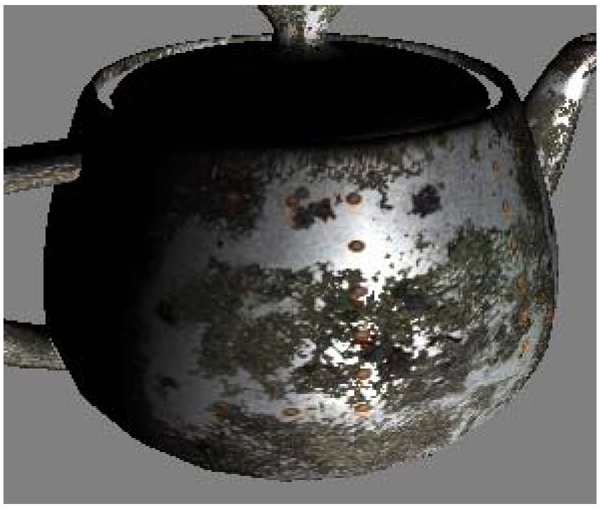
- nxDir will have a value > 0 if the angle between the vector is between 0º and 90º.
- If the angle is greater than 90º the max function will clamp the value to 0.

This gives us a measurable way of computing whether a point is lit or not (an angle > 90º between the normal and the lightvector means that the point doesn't receive light).

With this value we choose if there is specular contribution and, if so, how much. We also use this value to compute the strength of the specular contribution (according to the surface properties; in this case, we only take care of the shininess exponent).

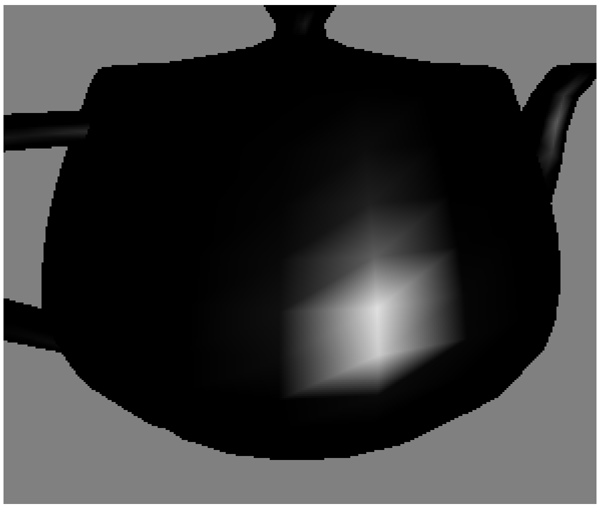
Let's go over some interesting screenshots.

Completed shader: Point light with ambient, diffuse, specular, and gloss mapping. You can see the specular highlight obscured by the gloss map:



Shader without gloss map; full specularity is applied:

You may be asking why you should exert so much effort to do the same thing that OpenGL does by itself with the function fixed pipeline, and you are right. OpenGL can do this without problems with just a few calls to glLightfv. However, that method has a large problem. It is per-vertex lighting, which means that the final quality depends on how many triangles have our mesh. With poor tessellation, it would look something like this (without textures, for clarity):



This is a terrible illumination model! But don’t worry; we’ve done all the work up to this point and know how light contributions can be computed. The process is the same, no matter if we do it in the vertex shader (per vertex lighting) or in the fragment shader (per pixel lighting).

Per vertex lighting consist of computing the light contributions on a per vertex basis, and passing those contributions, interpolated through the primitive, to the fragment shader. Per pixel lighting is almost the same, but with one major difference: the value passed from the vertex shader, via an interpolator, to the fragment shader, is the vertex normal, and all lighting computations are done in the fragment shader with this interpolated normal. With per pixel lighting, we only need to do a few small changes to get a great-looking illumination. The following chapter will cover this.