Computer Graphics – The Vertex and Fragment Shaders

Three.js and GLSL Handout

In the example *html* files the shader codes are in the bottom of the file. They are indicated by <script> tags, which have type="x-shader/x-vertex" or type="x-shader/x-fragment".

```
<script id="vertexShader" type="x-shader/x-vertex">
    void main() {
        gl_Position = projectionMatrix * modelViewMatrix * vec4(position,1.0);
    }
</script>
<script id="fragmentShader" type="x-shader/x-fragment">
    void main() {
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    }
</script>
```

Code inside the void main() function will be executed

Main purpose of the **vertex shader** is to assign a correct position in the **gl_Position** built-in variable.

Main purpose of the **fragment shader** is to assign a correct color in the **gl_FragColor** built-in variable.

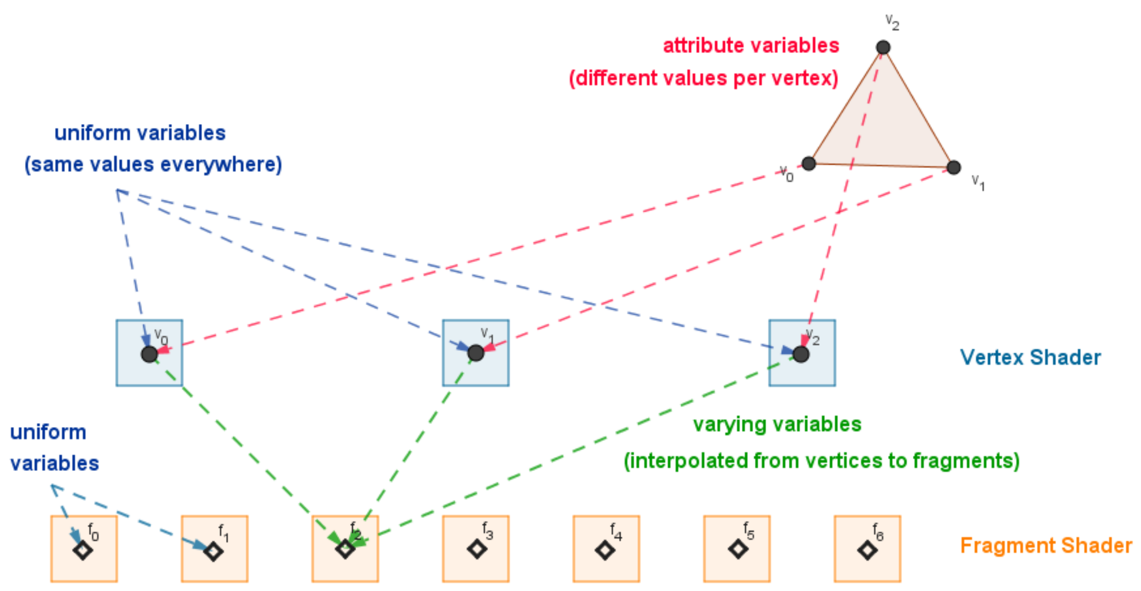
GLSL is very similar to C++ in syntax. However, it is meant to do mainly arithmetics (math ops). It has a very convenient syntax for matrices, vectors, trigonometric functions etc. Eg some tricks with vectors:

Syntax	**Meaning**		**Syntax**	**Meaning**
vec3(1.0, 2.0, 3.0);	Creates a vector [1, 2, 3]		vec3 v = vec3(1.0, 2.0, 3.0); vec2 u = **v.xx**;	Vector *u* will be [1, 1]
vec3(1.0);	Creates a vector [1, 1, 1]		float d = **length**(v);	Variable *d* will have the length of *v*

vec3 v = vec3(0.0);
vec4 u = vec4(**v**, 1.0);
	Vector *u* will be [0, 0, 0, 1]		float x = **dot**(u, v);	Variable *x* will have the dot product of *u* and *v*.

vec3 v = vec3(1.0, 2.0, 3.0); float x = **v.x**;	Variable *x* will be 1		vec3 w = **cross**(u, v);	Vector *w* will be the cross product of *u* and *v*
vec3 v = vec3(1.0, 2.0, 3.0); vec2 u = **v.xy**;	Vector *u* will be [1, 2]		Vec3 r = **reflect**(v, n)	Vector *r* will be the reflection vector of *v* off a surface with the normal *n*.

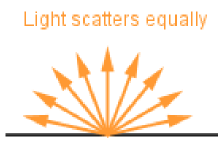
Google: "**webgl api quick reference card**", it is an official reference card, pages 3 and 4 give a great overview of WebGL-s GLSL syntax.



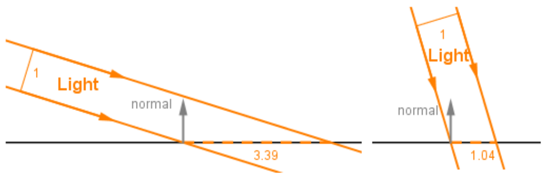
The **uniform**, **attribute** and **varying** variables should be declared before the void main() function in the shaders. Three.js library already sets and sends a lot of **common variables automatically**. For example the **uniform matrices** projectionMatrix, modelViewMatrix. Here are some of them in the **vertex shader**:

Variable	**Meaning**		**Variable**	**Meaning**
uniform mat4 modelMatrix	Transforms coordinates from local space to world space.		uniform vec3 cameraPosition	Camera's coordinates in world space.
uniform mat4 viewMatrix	Transforms coordinates from world space to view space.		attribute vec3 position	Vertex coordinates in local space.
uniform mat4 modelViewMatrix	Transforms coordinates from local space to view space.		attribute vec3 normal	Normal vector of the vertex.
uniform mat4 projectionMatrix	Transforms coordinates from view space to clip space.		attribute vec2 uv	UV-coordinates of the vertex
uniform mat3 normalMatrix	Transforms normal vectors from local space to view space.			

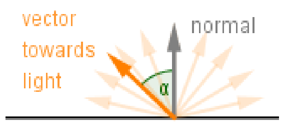
Lambertian reflectance model specifies that light scatters from the surface to all directions equally. The final color we see is independant of the viewer's position. This is a completely **ideal matte surface**.



However, the amount of light reaching the surface depends on the light's direction. Greater the angle between the surface normal and the vector towards the light source, less light will reach one surface unit.



The amount of light reaching one surface unit is proportional to the **cosine** of the angle between the **normal** and **direction towards the light source**.



Cosine of an angle between two vectors can be calculated with a **dot product** if the vectors are **normalized** (have unit length). The cosine could also be negative. In lighting calculations it does not make sense to have a **negative value** there, so usually the **maximum** of 0 and the dot product result is taken.

$$reflectedLight = \max(0, vectorTowardsLight \cdot normal)$$

The reflected light intensity also depends on the intensity of the light source and material's reflective properties. For example, the material only reflects 50% of the incoming light. The light source is dim etc.

$$reflectedLight = lightIntensity \cdot materialReflectivity \cdot \max(0, vectorTowardsLight \cdot normal)$$

We define the color with the **red**, **green** and **blue** intensity components. The Lambertian reflection model is applied to each of those components using the light and material color values.

In real environments light does not come only from a single source, but bounces around everywere. There is almost always some **ambient light** around the environment, illuminating objects indirectly. Calculating that is a great challenge in computer graphics with different solutions balancing **realism** and **computational complexity**. The most simplest way is to **just add a very small value** to the intensity calculations in the model. This way the surface not directly illuminated will not be totally black.