GLSL Notes

***This note only covers: key points that I found it's necessary to clarify or remember*

*I'll list some **tutorials** that I found useful **learning shaders** as they are more accurate and in details:*

- *http://cs.uns.edu.ar/cg/clasespdf/GraphicShaders.pdf emm… the textbook talks about theory stuff*
- *https://github.com/mattdesl/lwjgl-basics/wiki great GLSL tutorial*
- *https://thebookofshaders.com/ shaping functions and cool visuals*

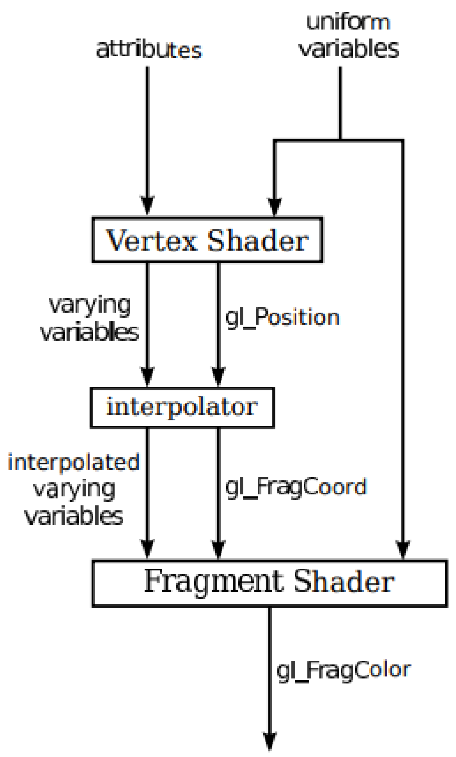
**Before you move on to the world of shaders, you might need to know:*

- *Computer graphics pipeline (especially the role of shaders)*
- *Texture mapping*
- *Basic math in graphics*

Check http://www.opengl-tutorial.org/beginners-tutorials/ or just google anything you don't know :)

Notes start from here

- **What is GLSL?**
  - GLSL stands for OpenGL Shading Language
  - Shaders are like small scripts that let us interact with the GPU more closely
  - Used for a variety of effects and visuals in 2D and 3D
- **Vertex Shaders**
  - Allows you to interact with this vertex information before sending it along the graphics pipeline to be rendered
  - More applicable in 3D programming: you might want to calculate the position of the vertex
- **Fragment Shades**
  - Allows us to modify pixels before they are sent along the graphics pipeline
  - Output is a RGBA color // gl_FragColor
- **Attributes, uniforms and varyings**

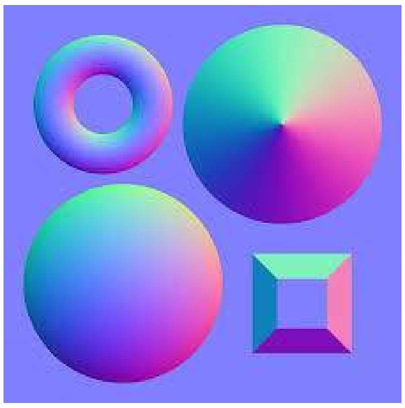


- In order for the fragment shader to use the attributes, we need to “pass them along” by declaring the varyings in the vertex and fragment shader. Our varying names can be anything, as long as they are consistent between fragment.
- Example: we declare fNormal in vertex shader and use fNormal in fragment shader

```
THREE.NormalMappingShader = {
    uniforms: {
            texture1: {type: "t", value: null},
            scale: {type: "f", value: 1.0},
    },
    vertexShader: [
            "varying vec3 fNormal;",
            "void main() {",
            "fNormal = normal;",
            "gl_Position = projectionMatrix * modelViewMatrix * vec4(
position, 1.0 );",
             "}"
    ].join("\n"),

    fragmentShader: [
              // varying variable, we must defind it in both vs and fs
              // its purpose is to carry the value copied from the
attribute
              // "normal" to the fragment shader.
             "varying vec3 fNormal;",
             "void main( void ) {",
             // compose the colour using the normals
             "gl_FragColor = vec4( fNormal, 1. );",
             "}"
    ].join("\n")
};
```

```

```

- Result in normal mapping:



- **The square**

  - Each little square knows its position

```
vec2 position =  gl_FragCoord.xy / resolution.xy;
```

  - [0, 1] x [0, 1] meaning that often times we need to **normalize/remap things**

The functions... and the use of the functions
Most shaping functions are covered by https://thebookofshaders.com/05/

Those are what I found useful:

- Why do we use sin and cos?
  - They return normalized values between -1 and 1 in such a smooth way
- Why do we use fract()?
  - choping the value from 0 to 1
- Why do we use dot product ?
  - It returns a single float value between 0.0 and 1.0 depending on the alignment of two vectors.
- So here we got the random function

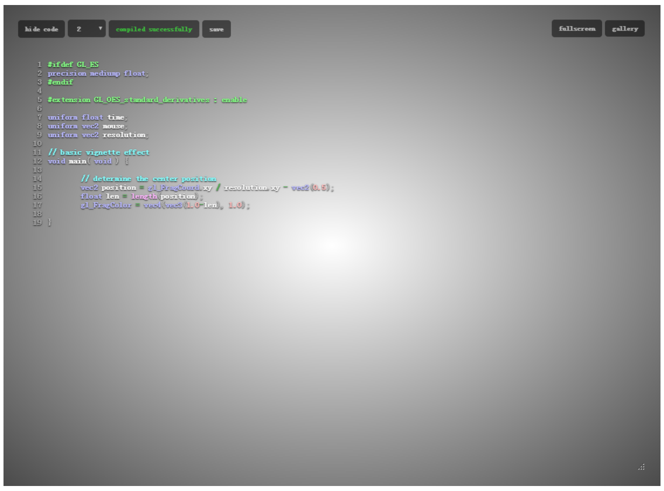
```
 float random (vec2 st) {
    return fract(sin(dot(st.xy,
                         vec2(12.9898,78.233)))*
        43758.5453123);
}

```

- How to find the center position?

```
vec2 position =  gl_FragCoord.xy / resolution.xy - vec(0.5);
```

- caculate the length to the center, then create the basic vignette effect:

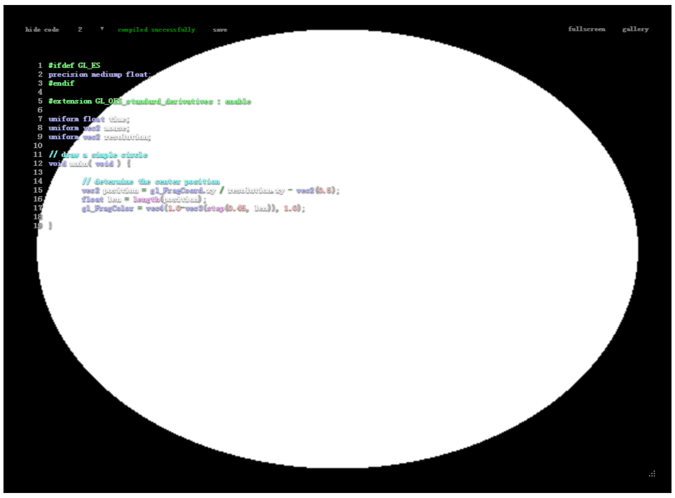
```
float len = length(position);
gl_FragColor = vec4(vec3(1.0 - len), 1.0);
```



```
#ifdef GL_ES
precision mediump float;
#endif

#extension GL_OES_standard_derivatives : enable

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

// basic vignette effect
void main( void ) {

	// determine the center position
	vec2 position = gl_FragCoord.xy / resolution.xy - vec2(0.5);
	float len = length(position);
	gl_FragColor = vec4(vec3(1.0-len), 1.0);

}
```

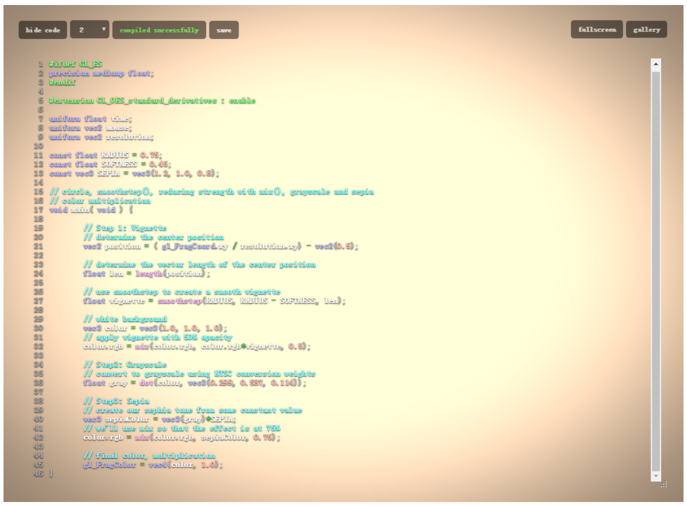
- How to draw a circle using length?
  - Use step() `

```
gl_FragColor = vec4(1.0-vec3(step(0.45, len)), 1.0);
```



```
#ifdef GL_ES
precision mediump float;
#endif

#extension GL_OES_standard_derivatives : enable

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

// draw a simple circle
void main( void ) {

	// determine the center position
	vec2 position = gl_FragCoord.xy / resolution.xy - vec2(0.5);
	float len = length(position);
	gl_FragColor = vec4(1.0-vec3(step(0.45, len)), 1.0);

}
```

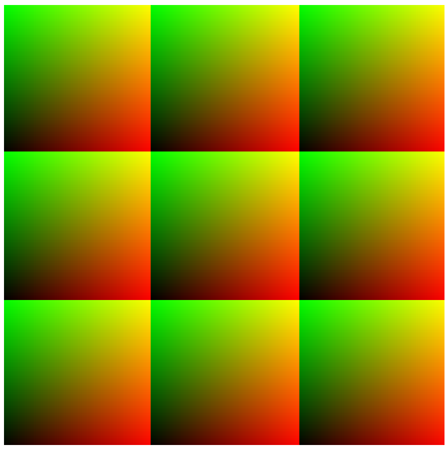
Then we are able to follow this tutorial to create this vignette effect:



```
#ifdef GL_ES
precision mediump float;
#endif

#extension GL_OES_standard_derivatives : enable

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

const float RADIUS = 0.75;
const float SOFTNESS = 0.45;
const vec3 SEPIA = vec3(1.2, 1.0, 0.8);

// circle, smoothstep(), reducing strength with mix(), grayscale and sepia
// color multiplication
void main( void ) {

	// Step 1: Vignette
	// determine the center position
	vec2 position = ( gl_FragCoord.xy / resolution.xy) - vec2(0.5);

	// determine the vector length of the center position
	float len = length(position);

	// use smoothstep to create a smooth vignette
	float vignette = smoothstep(RADIUS, RADIUS - SOFTNESS, len);

	// white background
	vec3 color = vec3(1.0, 1.0, 1.0);
	// apply vignette with 50% opacity
	color.rgb = mix(color.rgb, color.rgb*vignette, 0.5);

	// Step2: Grayscale
	// convert to grayscale using NTSC conversion weights
	float gray = dot(color, vec3(0.299, 0.587, 0.114));

	// Step3: Sepia
	// create our sephia tone from some constant value
	vec3 sepiaColor = vec3(gray)*SEPIA;
	// we'll use mix so that the effect is at 75%
	color.rgb = mix(color.rgb, sepiaColor, 0.75);

	// final color, multiplication
	gl_FragColor = vec4(color, 1.0);
}
```

- How to scale up the space by 3?

```
void main() {
       vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);
    st *= 3.0;      // Scale up the space by 3
    st = fract(st); // Wrap arround 1.0
    // Now we have 3 spaces that goes from 0-1
    color = vec3(circle(st,0.5));
       gl_FragColor = vec4(color,1.0);
}
```



- How to remap the space to –1 to 1?

```
  vec2 st = gl_FragCoord.xy/u_resolution.xy;
  st.x *= u_resolution.x/u_resolution.y;
  // Remap the space to -1. to 1.
  st = st *2. -1.;
```

- Simplex noise?
    - three ways to use noise function”
        - use 2D noise implementation to rotate the sapce (.. where the straight lines are rendered, we can produce the swirly effect that looks like wood.
        - another interesting patterns from noise is to treat it like a field ..
        - use the noise fundtion to modulate a shape / texture coordinate
- Rotate2d(float angle)

```
float rotate2d(float angle) {
  return mat(cos(angle), -sin(angle), sin(angle), cos(angle));
}
```

    - so add noise to the position and rotate??

```
pos = rotate2d(noise(pos))* pos;
```

- use time in the shader?

```
uniforms.time.value = time;
```