What does a Vertex Shader Do?

The basic function of a vertex shader is to take the vertex coordinates as supplied by the application, and perform whatever transformation of them is required. At the same time, the vertex shader can perform various analyses based on those vertex coordinates and prepare various values for later use in the graphics process.

```glsl
varying vec4 Color;
varying float X, Y, Z;

void main() {
    vec3 TransNorm = normalize( gl_NormalMatrix * gl_Normal );
    vec3 LightPos = vec3( 0., 0., 10. );
    vec3 ECposition = ( gl_ModelViewMatrix * gl_Vertex ).xyz;
    float LightIntensity = dot( normalize(LightPos - ECposition), TransNorm );
    LightIntensity = abs( LightIntensity );
    Color = gl_Color;
    vec3 MCposition = gl_Vertex.xyz;
    X = MCposition.x;
    Y = MCposition.y;
    Z = MCposition.z;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Don't worry about the details right now, just take comfort in the fact that it is C-like and that there appears to be a lot of support routines for you to use.
A Vertex Shader Replaces These Operations:

- Vertex transformations
- Normal transformations
- Normal normalization
- Handling of per-vertex lighting
- Handling of texture coordinates

What does a Fragment Shader Do?

The basic function of a fragment shader is to take uniform variables, the output from the rasterizer, and texture information and then compute the color of the pixel for each fragment. This figure illustrates this process, showing how the distinct vertices of a primitive are processed by the rasterizer to form the set of fragments that make up the primitive.

A Fragment Shader Replaces These Operations:

- Color computation
- Texturing
- Color arithmetic
- Handling of per-pixel lighting
- Fog
- Blending
- Discarding fragments

The Fragment shader then sets the color based on the X value.

Sample Vertex Shader: Stripes in Model and Eye Coordinates

varying vec4 Color;

varying float LightIntensity;

uniform float Density;

uniform float Frequency;

void main()
{
    vec2 st = gl_TexCoord[0].st;
    vec2 stf = st * Frequency;
    if (all(fract(stf) >= Density))
        discard;
    vec3 TransNorm = normalize(gl_NormalMatrix * gl_Normal);
    vec3 LightPos = vec3(0., 0., 10.);
    vec3 ECposition = (gl_ModelViewMatrix * gl_Vertex).xyz;
    LightIntensity = dot(normalize(LightPos - ECposition), TransNorm);
    LightIntensity = abs(LightIntensity);
    Color = gl_Color;
    vec3 myColor = gl_TexCoord[0].st;
    #ifdef EYE_COORDS
        X = ECposition.x;
        Y = ECposition.y;
        Z = ECposition.z;
    #endif
    #ifdef MODEL_COORDS
        X = MCposition.x;
        Y = MCposition.y;
        Z = MCposition.z;
    #endif
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
varying vec4 Color;
varying float X, Y, Z;
varying float LightIntensity;

uniform float A;
uniform float P;
uniform float Tol;

void main( void )
{
    const vec3 WHITE = vec4( 1., 1., 1. ) ;
    float f = fract( A*X );
    float t = smoothstep( 0.5-P-Tol, 0.5-P+Tol, f )  - smoothstep( 0.5+P-Tol, 0.5+P+Tol, f );
    vec3 color = mix( WHITE, Color.rgb, t );
    gl_FragColor= vec4( LightIntensity*color, 1. );
}

The 2 shaders might (momentarily) look the same, but they don't act the same!

Per-vertex vs. Per-fragment Lighting
In per-vertex lighting, the normal at each vertex is turned into a light intensity. That intensity is then interpolated throughout the polygon. This gives splotchy polygon artifacts, like this.

In per-fragment lighting, the normal is interpolated throughout the polygon and turned into a lighted intensity at each fragment. This gives smoother results, like this.

Think carefully about what you want as a varying variable – it can make a difference!

Image Basics
Treat the image as a texture and read it into the fragment shader

To get from the current texel to a neighboring texel, add ±(1./ResS , 1./ResT) to the current (S,T)

Image Negative
uniform sampler2D ImageUnit;
void main()
{
    vec2 st = gl_TexCoord[0].st;
    vec3 rgb = texture2D(ImageUnit, st ).rgb;
    vec3 neg = vec3(1.,1.,1.) - rgb;
    gl_FragColor = vec4( neg, 1. );
}

Brightness
Contrast
Using Textures as Data

```
fragment I
{
    vec3 visibleColor = texture2D(VisibleUnit, gl_TexCoord[0].st).rgb;
    vec3 infraredColor = texture2D(InfraRedUnit, gl_TexCoord[0].st).rgb;
    infraredColor = vec3(1.,1.,1.) - infraredColor;
    vec3 waterVaporColor = texture2D(WaterVaporUnit, gl_TexCoord[0].st).rgb;
    vec3 rgb = vec3(0.,0.,0.);
    if (visibleColor.r - visibleColor.g > 0.25 && visibleColor.r - visibleColor.b > 0.25)
    {
        rgb = vec3(1.,1.,0.); // state outlines become yellow
    }
    else
    {
        rgb = Visible*visibleColor + InfraRed*infraredColor + WaterVapor*waterVaporColor;
        rgb /= 3.;
        vec3 coefs = vec3(0.296, 0.240, 0.464);
        float visibleInten = dot(coefs,visibleColor);
        float infraredInten = dot(coefs,infraredColor);
        float waterVaporInten = dot(coefs,waterVaporColor);
        if (visibleInten > VisibleThreshold && infraredInten < InfraRedThreshold && waterVaporInten > WaterVaporThreshold)
        {
            rgb = vec3(0.,1.,0.);
        }
        else
        {
            rgb *= Brightness;
            rgb = clamp(rgb, 0., 1.);
        }
    }
    gl_FragColor = vec4(rgb, 1.);
}
```
uniform float                  Min;
uniform float                  Max;
uniform sampler3D       TexUnit;
const float SMIN =   0.;
const float SMAX = 100.;
void
main( void )
{
vec4 rgba = texture3D( TexUnit, gl_TexCoord[0].stp );
if( scalar < Min )
discard;
if( scalar > Max )
discard;
float t = ( scalar - SMIN ) / ( SMAX - SMIN );
vec3 rgb = Rainbow( t );
gl_FragColor = vec4( rgb, 1. );
}

vec3 Rainbow( float t )
{
t = clamp( t, 0., 1. );
vec3 rgb;
// b -> c
rgb.r = 0.;
rgb.g = 4. * ( t - (0./4.) );
rgb.b = 1.;
// c -> g
if( t >= (1./4.) )
{rgb.r = 0.;
rgb.g = 1.;
rgb.b = 1. - 4. * ( t - (1./4.) );
}
// g -> y
if( t >= (2./4.) )
{rgb.r = 4. * ( t - (2./4.) );
rgb.g = 1.;
rgb.b = 0.;
}
// y -> r
if( t >= (3./4.) )
{rgb.r = 1.;
rgb.g = 1. - 4. * ( t - (3./4.) );
rgb.b = 0.;
}
return rgb;
}

3D Probe – Assigning the Transfer Function to Arbitrary Geometry

Visualization – Don’t Send Colors to the GPU, Send the Raw Data

Visualization Transfer Functions

Cutting Planes
uniform float Min;
uniform float Max;
uniform sampler3D TexUnit;

varying vec3 MCposition;

const float SMIN = 0.;
const float SMAX = 120.;

void main( void )
{
    vec3 stp = ( MCposition + 1.0 ) / 2.;
    if( any(stp) < 0.0 || any(stp) > 1.0 )
        discard;
    vec3 rgb = Rainbow( t );
    //vec3 rgb = HeatedObject( t );
    gl_FragColor = vec4( rgb, 1.0, );
}