Introduction to Shaders for Visualization

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University

The Basic Computer Graphics Pipeline

Model Transform → View Transform → Per-vertex Lighting → Projection Transform → Homogeneous Division → Viewport Transform → Rasterization → Framebuffer

Rasters Ops → Fragment Processing, Texturing, Per-fragment Lighting
The Basic Computer Graphics Coordinate Systems

Model Transform → View Transform → Per-vertex Lighting → Projection Transform → Homogeneous Division

NDC → Viewport Transform

Viewport Transform → Rasterization

Fragment Processing, Texturing, Per-fragment Lighting → SC

Framebuffer → Rasters Ops

MC = Model Coordinates
WC = World Coordinates
EC = Eye Coordinates
CC = Clip Coordinates
NDC = Normalized Device Coordinates
SC = Screen Coordinates

OpenGL gives you Access to two Transformations

Model Transform → View Transform → Per-vertex Lighting → Projection Transform

These two are lumped together into a single matrix called the **ModelView Matrix**.

In GLSL, this is called **gl_ModelViewMatrix**

This one is called the **Projection Matrix**.

In GLSL, this is called **gl_ProjectionMatrix**

GLSL also provides you with these two already multiplied together.

This is called **gl_ModelViewProjectionMatrix**

MC = Model Coordinates
WC = World Coordinates
EC = Eye Coordinates
CC = Clip Coordinates
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 variable values for later on in the graphics process.

Here's What a Shader Looks Like

```cpp
varying vec4 Color;
varying float X, Y, Z;
varying float LightIntensity;

void main( void )
{
    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 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 first 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

```cpp
varying float LightIntensity;
uniform vec4 Color;
void main()
{
    gl_FragColor = vec4( LightIntensity * Color.rgb, 1.0 );
}
```

```cpp
varying vec3 myColor;
void main(void)
{
    gl_FragColor = vec4( myColor, 1.0 );
}
```
### Fragment Shader: Discarding Fragments

```glsl
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;
    gl_FragColor = vec4(LightIntensity * Color.rgb, 1.);
}
```

### Sample Vertex Shader: Stripes in Model and Eye Coordinates

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

void main(void) {
    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 MCposition = gl_Vertex.xyz;

#ifdef EYE_COORDS
    X = ECposition.x;
    Y = ECposition.y;
    Z = ECposition.z;
#else
    X = MCposition.x;
    Y = MCposition.y;
    Z = MCposition.z;
#endif
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The Fragment shader then sets the color based on the X value.
Sample Fragment Shader: Stripes in Model and Eye Coordinates

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. );
}

Sample Vertex Shader: Stripes in Model and Eye Coordinates

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)
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. );
}
Saturation

Sharpening

Edge Detection
Using Textures as Data

frag file, I

uniform sampler2D       VisibleUnit;
uniform sampler2D       InfraRedUnit;
uniform sampler2D       WaterVaporUnit;
uniform float                  Visible;
uniform float                  InfraRed;
uniform float                  WaterVapor;
uniform float                  VisibleThreshold;
uniform float                  InfraRedThreshold;
uniform float                  WaterVaporThreshold;
uniform float                  Brightness;

void
main()
{
    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;

    if( visibleColor.r - visibleColor.g  > .25   &&   visibleColor.r - visibleColor.b > .25 )
    {
        rgb = vec3( 1., 1., 0. );       // state outlines become yellow
    }
    else
    {
        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. );
}

Using Textures as Data

frag file, II

if( visibleColor.r - visibleColor.g > .25  &&  visibleColor.r - visibleColor.b > .25 )
{
    rgb = vec3( 1., 1., 0. ); // state outlines become yellow
}
else
{
    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. );
Using Textures as Data – Where is it Likely to Snow?

Visible  Infrared  Water vapor

Point Cloud from a 3D Texture Dataset

Low values culled

Full data
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 );
    float scalar = rgba.r;
    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;
}

vec3 HeatedObject( float t )
{
    t = clamp( t, 0., 1. );
    vec3 rgb;
    rgb.r = 3. * ( t - (0./6.) );
    rgb.g = 0.;
    rgb.b = 0.;
    if( t >= (1./3.) )
    {
        rgb.r = 1.;
        rgb.g = 3. * ( t - (1./3.) );
    }
    if( t >= (2./3.) )
    {
        rgb.g = 1.;
        rgb.b = 3. * ( t - (2./3.) );
    }
    return rgb;
}
Visualization -- Don’t Send Colors to the GPU, Send the Raw Data

Use the GPU to turn the data into graphics on-the-fly

3D Probe – Assigning the Transfer Function to Arbitrary Geometry

Visualization by Chris Janik

mjb – February 10, 2010
uniform float        Min;
uniform float        Max;
uniform sampler3D    TexUnit;
varying vec4         ECposition;
const float    SMIN =   0.;
const float    SMAX = 120.;

void
main( void )
{
    vec3 stp = clamp( ( ECposition.xyz + 1. ) / 2., 0., 1. );  // maps [-1.,1.] to [0.,1.]
    vec4 rgba = texture3D( TexUnit, stp );
    float scalar = rgba.r;
    if( scalar < Min )
        discard;
    if( scalar > Max )
        discard;
    float t = ( scalar - SMIN ) / ( SMAX - SMIN );
    vec3 rgb = Rainbow( t );
    gl_FragColor = vec4( rgb, 1. );
}
```glsl
// frag file

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. ) / 2.; // maps [-1.,1.] to [0.,1.]
    if( any(stp) < 0. || any(stp) > 1. )
        discard;

    vec4 rgba = texture3D( TexUnit, stp );
    float scalar = rgba.r;
    if( scalar < Min || scalar > Max )
        discard;

    float t = ( scalar - SMIN ) / ( SMAX - SMIN );
    vec3 rgb = Rainbow( t );
    //vec3 rgb = HeatedObject( t );
    gl_FragColor = vec4( rgb, 1. );
}
```

---

**Volume Rendering – Ray Casting**
Extruding Shapes Along Flow Lines

Extruding a block arrow along a spiral flow line

Adding moving “humps” to create a peristaltic effect

Bump-Mapping for Terrain Visualization

Visualization by Nick Gebbie