M30242 – Graphics and Computer Vision

Lecture 09: Shading

Review: Phong Lighting Model

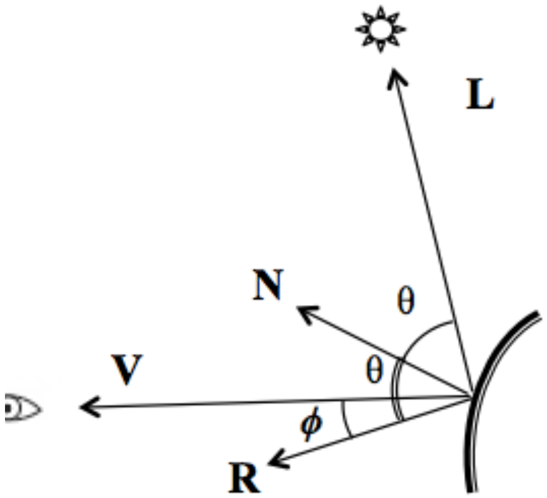
- Phong lighting model states that the light reflected from a point on a surface consist of three components:
  - Ambient reflection
    - Independent of viewing direction.
  - Diffuse reflection
    - Dependent on the incident angle (the angle between surface normal and incident light rays).
    - Viewing angle doesn’t matter
  - Specular reflection
    - Dependent on light source (colour and intensity), incident angle and viewing angle.

Reflection Calculation

- In vector form:

$$I_{total} = c_a + \sum_{i=1}^{all\_lights} \left(c_d \cos\theta + c_s (\cos\phi)^\alpha \right)$$

$$= c_a + \sum_{i=1}^{all\_lights} \left(c_d \mathbf{L_i} \bullet \mathbf{N} + c_s (\mathbf{V} \bullet \mathbf{R})^\alpha \right)$$



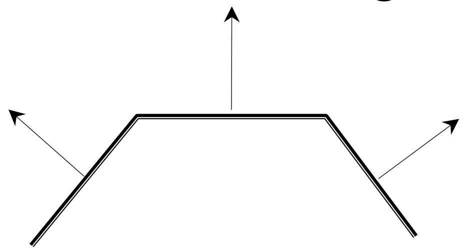
- where unit vectors **N**, **L**$_{\mathbf{i}}$, **V** and **R** represents, respectively
  - the surface normal at a point,
  - the direction to the light sources from the point,
  - the direction to the camera/viewer, and
  - the direction of ideal reflection.

Fragment Shade Calculation

- We have three ways to calculate the shades of the fragments of a polygon:
  - Flat shading: treating the entire triangle as single point and evaluating the lighting model once and assigning the same colour for all the fragment/pixels of the entire triangle.
  - Gouraud shading: applying the lighting model to the vertices of a triangle to produce the vertex shades and determining the colours of other fragemnts from them.
  - Phong shading: applying the lighting model individually to all the fragments of a triangle to get the colour for each of them.
- Both Gouraud & Phong shading models are widely used. Flat shading is efficient but rarely used nowadays.

Flat Shading

- Flat shading calculates a single shade for a polygon.

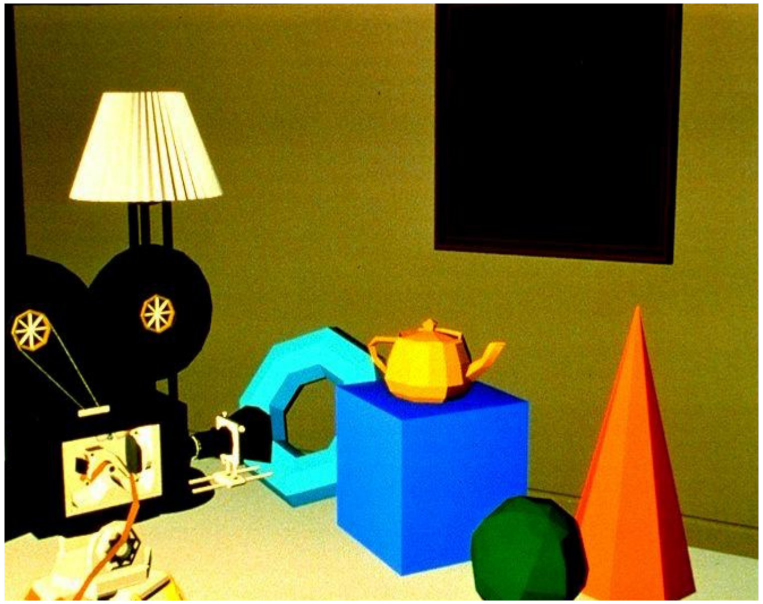


- If a reflection model is chosen, e.g., Phong lighting model, we only need to evaluate the model once and assign the colour to **All** the fragments/pixels of the polygon.

$$I_{total} = k_a I_a + \sum_{i=1}^{all\_lights} \left(I_{d_i} k_d \cos\theta + I_{s_i} k_s (\cos\phi)^{\alpha} \right)$$

Flat Shading

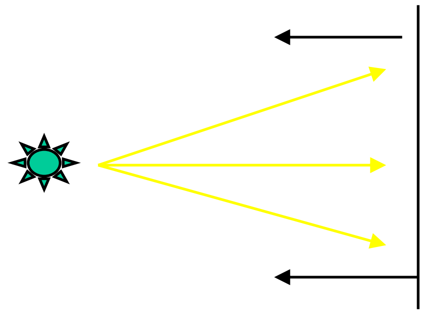
- Flat shading is very efficient and works well for flat surface under certain conditions.

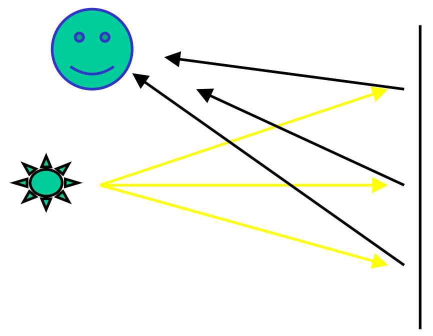


Result of Phong lighting + Flat Shading

Problems with Flat Shading

- In general, flat shading does not produce good visual result even for objects that indeed have flat surfaces
- Reasons: diffuse and specular reflections depend on incident and viewing directions.
- If flat shading is use
  - the incident direction varies across the surface
  - the viewing direction varies across the surface (if the surface is large).
- Result: Patchy surfaces and incorrect highlights





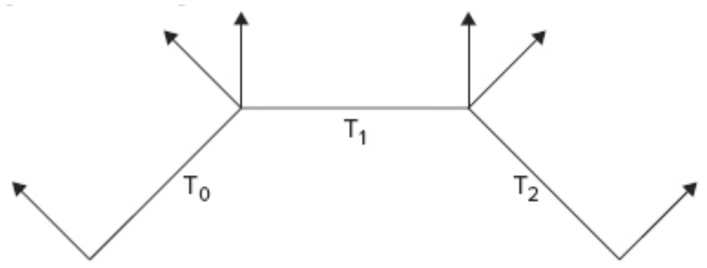
When Use Flat Shading…

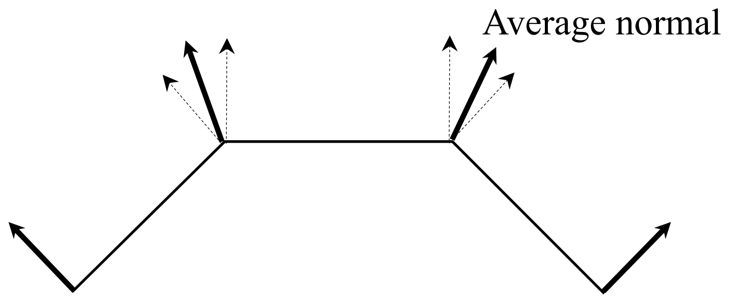
- When using flat shading, we have to notice that it only provide acceptable results under the following circumstances:
  - When the point light source is far enough or a directional light is used, so that the angle of the incident rays is nearly the same across the triangle.
  - When the viewer is far enough so that the view angle is the same for all the fragment/pixels of the triangle.

- It is used in application where rendering efficiency is of primary concern.

Gouraud Shading

- Published by French computer scientist Henri Gouraud in 1971.
- It is also called per-vertex shading, since shading calculation is done for the vertices of a polygon (triangle) and the colours of the vertices are linearly interpolated over the polygon.
- Normally, for better visual realism (smooth transition from one polygon to another), the method uses the average normal of neighbouring polygons.



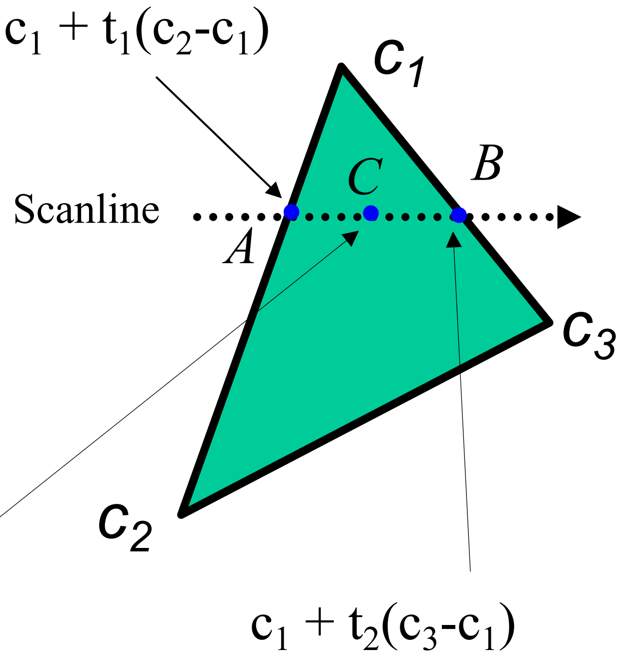


Cont'd

- The reason is that if a polygon is an approximation of a curved surface (e.g., a sphere), the normals that are perpendicular to the polygon at each vertex is incorrect. Instead, we should use a normal that considers the orientation of the underlying curved surface.
- The average normal gives a smoother transition of the colour from one triangle to adjacent triangles, so the seams between triangles are less obvious than they are for flat shading.
- Notice that in the lab session, we do not use average normals for cube.

Fragment Shades

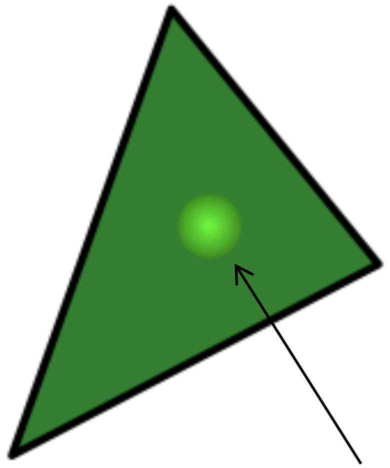
- Fragment shades are calculated using linear interpolation.
- Given the vertex colours, c_1, c_2 and c_3, the interpolation is carried out in three steps:
  - First, two interpolations are done along two edges. This determines two colours for two points on the scanline along the edges, i.e, A and B
  - Then, a 3rd interpolation is done along the scanline to find shade for any fragment C.



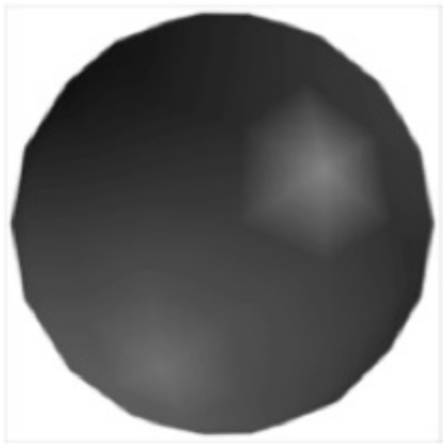
$$c=c_1 + t_1(c_2-c_1) + t_3(c_1 + t_2(c_3-c_1)- c_1 + t_1(c_2-c_1))$$

Cont'd

- In general, Gouraud shading produces a decent result for most matte surfaces.
- But for shiny surfaces, the shading model will produce defects in specular highlights – since shading is only calculated at the vertices, a highlight that falls inside a triangle can be missed (i.e., highlight only happens at a vertex).



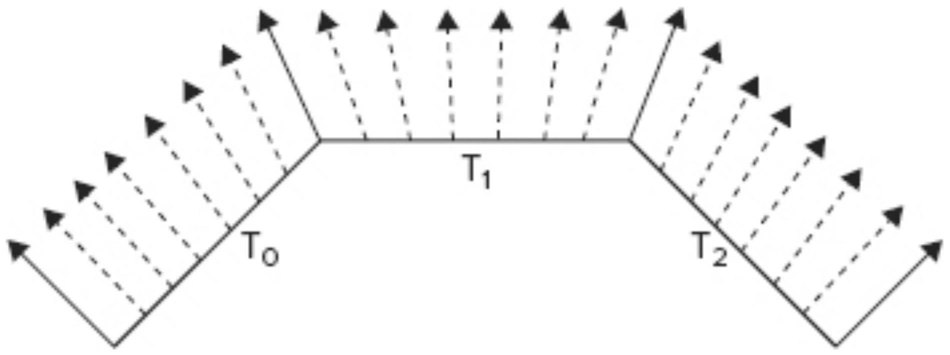
Can't produce such highlights.



Gouraud shading applied to a sphere

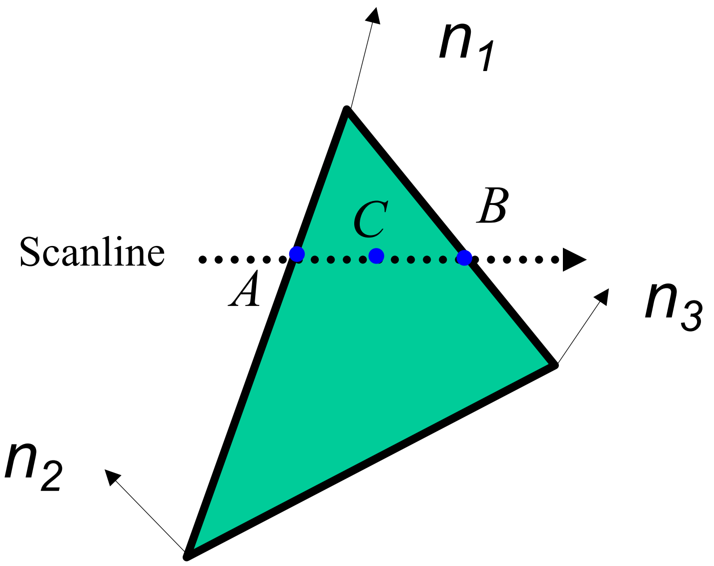
Phong Shading

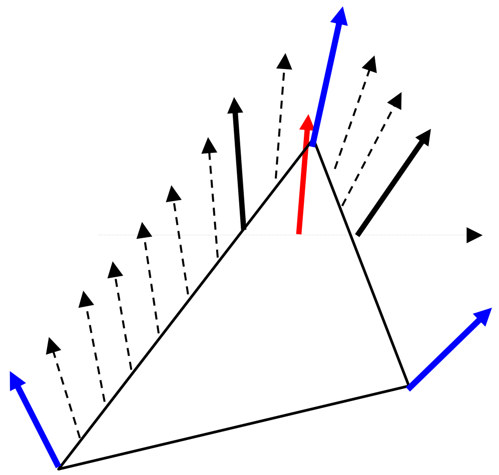
- In Phong shading, vertex normals are interpolated across a polygon/triangle and each fragment has its own normal (more accurate than Gouraud shading).
- Phong shading model evaluates lighting model (equation) for every fragment; therefore, it is called per-fragment shading.



Fragment Normals

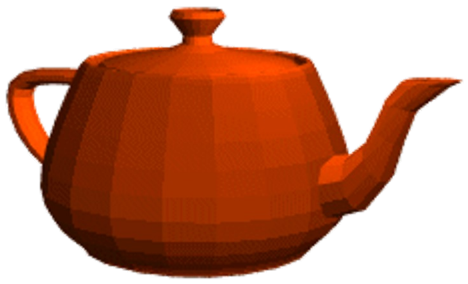
- The normal of a fragment is obtained by linear interpolation of the vertex normals. E.g., the normal for fragment C is obtained by interpolating n_1, n_1, and $n_{1.}$

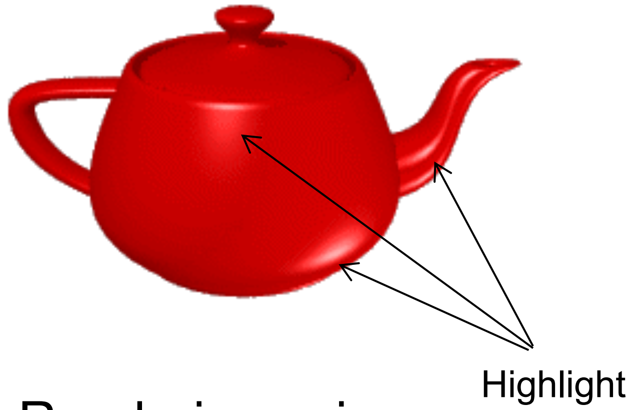




Quality of Renderings

- Better than the result of Gouraud shading.
  - Mach band (prominence of appearance of polygon edges) is less obvious.
  - Correct highlights.
- But, computationally more expensive.

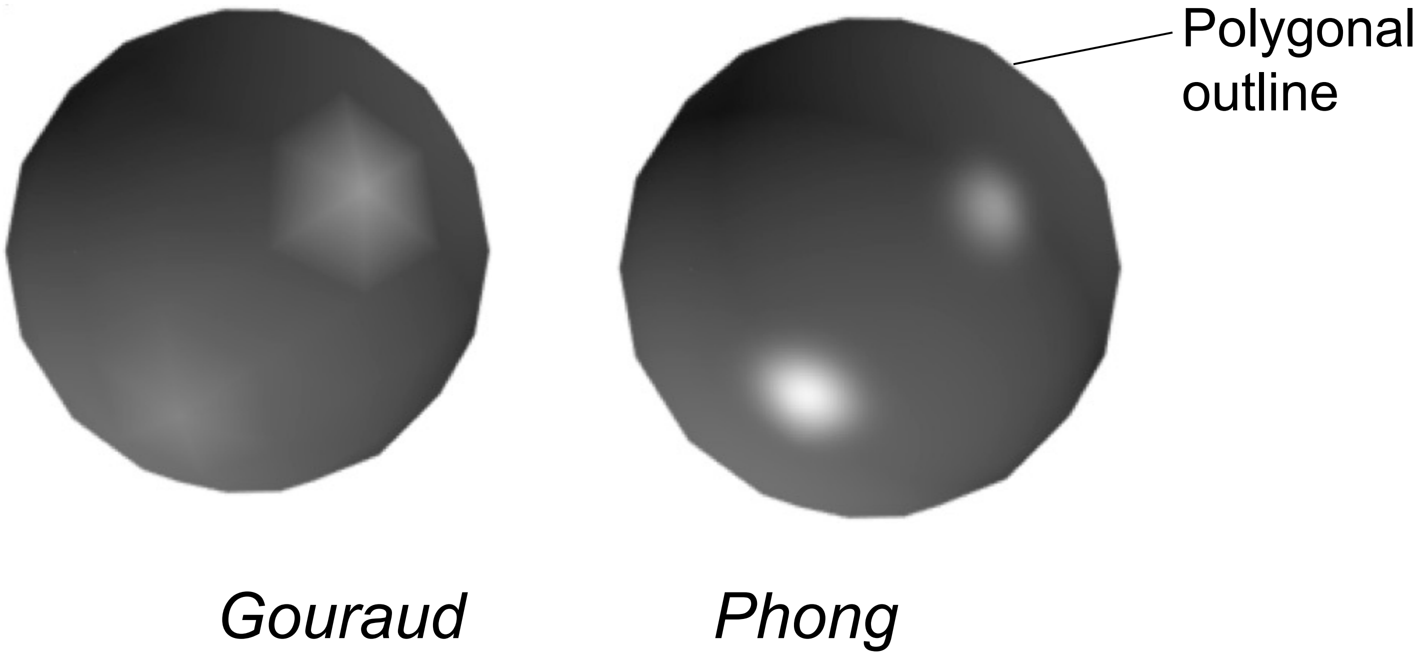

low polygon model of a teapot



Rendering using Phong shading

Limitations

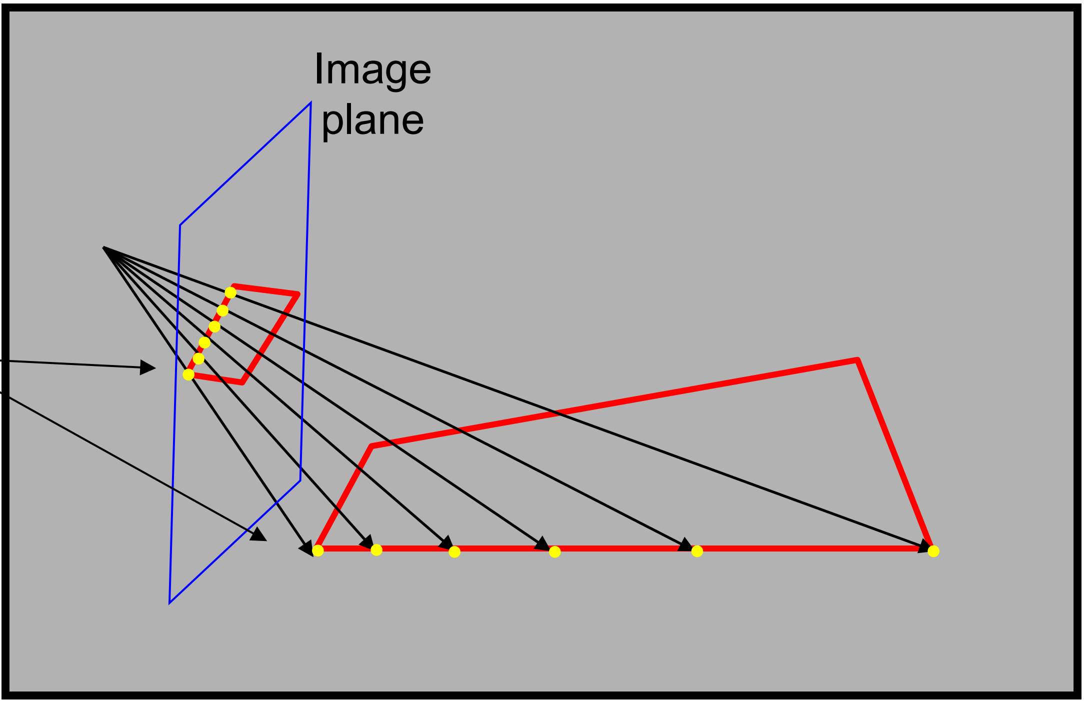
- Both Gouraud & Phong shadings use interpolation at certain stage, therefore they are called interpolated shading.
- These shading models cannot eliminate the polygonal (jagged) boundaries in silhouettes/outlines or shadows.



Perspective Distortion

- Both Gouraud and Phong shadings are subject to perspective distortion – linear interpolation in screen space does not align with linear interpolation in world space, i.e., making interpolation in world space nonlinear.

Evenly spaced points in screen space do not correspond to evenly spaced points in 3D



Solution: making large polygons into small ones to alleviate the foreshortening effect of perspective projection

Implementation in WebGL

- For Gouraud shading, the lighting model evaluation is done at the vertices, so it should be implemented in vertex shader where per-vertex operations are done (see tutorial of Lighting).
- For Phong shading, the vertex normals are interpolated. The lighting model is evaluated for every fragment using the interpolated fragment normals – so lighting calculation has to be done in the fragment shader where per-fragment operations are done.

Vertex Shader

```
<script id="shader-vs" type="x-shader/x-vertex">

    attribute vec3 aVertexPosition;
    attribute vec3 aVertexNormal;
    attribute vec2 aTextureCoordinates;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
    uniform mat3 uNMatrix;

    varying vec2 vTextureCoordinates;
    varying vec3 vNormalEye;
    varying vec3 vPositionEye3;
```

Per-vertex attributes received from data buffers

Per-vertex attributes to be passed to the fragment shader.

Cont’d

```
void main() {
    // Get vertex position in eye coordinates and send to the fragment shader. Notice that
    // vertexPositionEye4 is different from gl_Position.
    vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition,
                                                1.0);
    vPositionEye3 = vertexPositionEye4.xyz /
                                                vertexPositionEye4.w;
    // Transform the vertex normal to eye coordinates and send to fragment shader
    vNormalEye = normalize(uNMatrix * aVertexNormal);

    // Transform the geometry
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition,
                                                1.0);
    vTextureCoordinates = aTextureCoordinates;
}
</script>
```

Fragment Shader

```
<script id="shader-fs" type="x-shader/x-fragment">
    precision mediump float;
```

```
varying vec2 vTextureCoordinates;
varying vec3 vNormalEye;
varying vec3 vPositionEye3;
```

Interpolated version of these variables

```
uniform vec3 uLightPosition;
uniform vec3 uAmbientLightColor;
uniform vec3 uDiffuseLightColor;
uniform vec3 uSpecularLightColor;
uniform sampler2D uSampler;

const float shininess = 32.0;
```

Notice these lighting -related uniform variables are now defined in Fragment shader

Fragment Shader

```
void main() {
    // Calculate the vector (L) to the light source
    vec3 vectorToLightSource = normalize(uLightPosition -
      vPositionEye3);
    // Calculate N dot L for diffuse lighting
    float diffuseLightWeighting = max(dot(vNormalEye,
                                   vectorToLightSource), 0.0);
    // Calculate the reflection vector (R) that is needed for specular light
    vec3 reflectionVector = normalize(reflect(
                         -vectorToLightSource, vNormalEye));
    // Calculate viewVector (v) in eye coordinates as
    // (0.0, 0.0, 0.0) - vPositionEye3
    vec3 viewVectorEye = -normalize(vPositionEye3);
    float rdotv = max(dot(reflectionVector, viewVectorEye),
      0.0);
    float specularLightWeighting = pow(rdotv, shininess);
```

Fragment Shader

```
    // Sum up all three reflection components
    vec3 lightWeighting =
               uAmbientLightColor +
               uDiffuseLightColor * diffuseLightWeighting +
               uSpecularLightColor * specularLightWeighting;

    // Sample the texture
    vec4 texelColor = texture2D(uSampler, vTextureCoordinates);
    // modulate texel color with lightweigthing and write as final color
    gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb,
                      texelColor.a);
}
</script>
```

Further Reading

- Anyuru, A., WebGL Programming – Develop 3D Graphics for the Web
  - Chapter 7