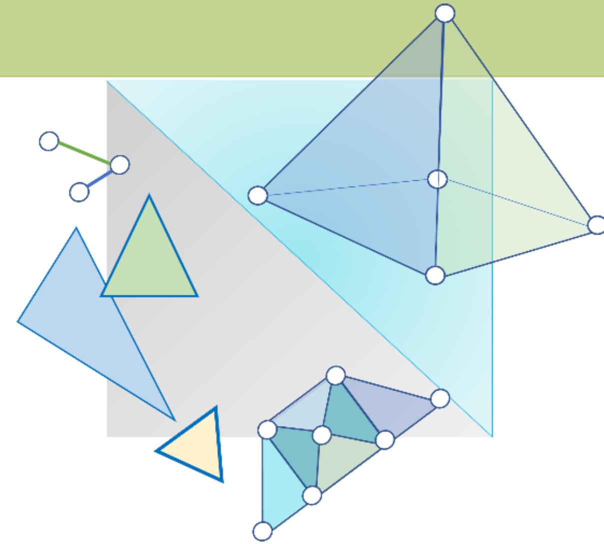
CITS3003 Graphics & Animation

Lecture 17:
Shading in OpenGL



Objectives

- Introduce the OpenGL shading methods
  - per vertex shading
  - per fragment shading
  - where to carry out

- Discuss polygonal shading
  - Flat Shading
  - Smooth Shading
    - Gouraud shading
    - Phong shading

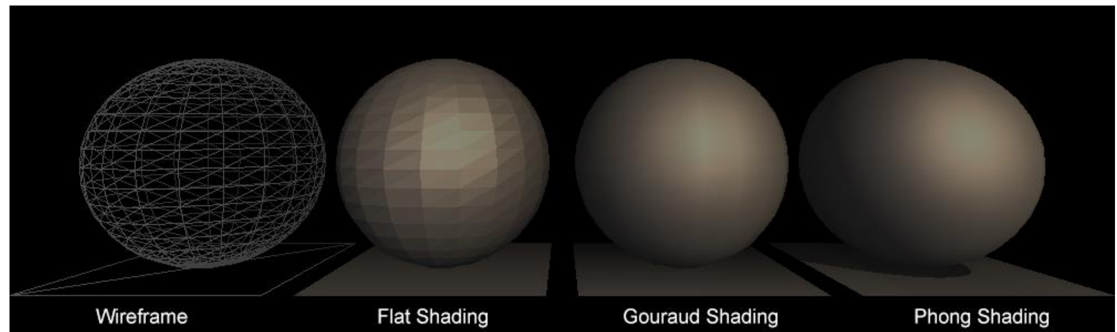


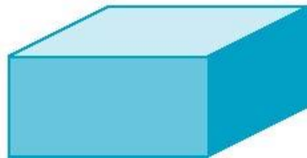
Image source

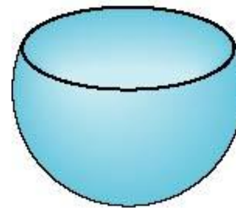
OpenGL Shading

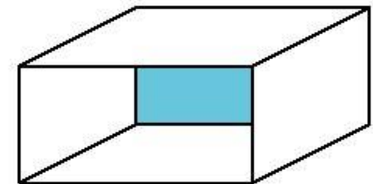
- To compute the shading values, OpenGL needs
  - normal vectors (denoted by **n**)
  - material properties
  - light vectors (denoted by **l**)
- The cosine terms in lighting calculations (see the previous lecture) can be computed using dot product.
- Using unit length vectors simplifies calculation.
- GLSL has a built-in *normalization* function for normalizing vectors to unit length.

Front and Back Faces

- Every surface has a front and back face
- For many objects, we never see the back face, so we don't care how or if it is rendered
- If it matters, we can handle that in the shader





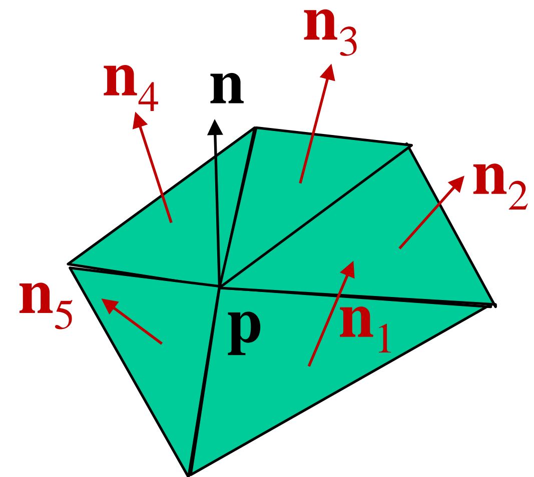


back faces not visible

back faces visible

Vertex Normals

- Complex surfaces are often represented by a triangular mesh.
- Consider 5 adjacent triangles sharing a common vertex **p**.
  - We can compute the normals $\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_5$ of the 5 triangles
  - We can then average these normals to get the normal **n** at **p**:
    $\mathbf{n} = \sum_{i=1}^{5} \mathbf{n}_i$ then $\mathbf{n} \leftarrow \mathbf{n}/\|\mathbf{n}\|$
  - This allows us to have normals defined at vertices
  - We call **n** the **vertex normal** at **p**.



Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix

- Depending on where we place the position (direction) setting function, we can
  - move the light source(s) with the object(s)
  - fix the object(s) and move the light source(s)
  - fix the light source(s) and move the object(s)
  - move the light source(s) and object(s) independently

- In interactive graphics, users should be able to do all of the above.

Specifying a Point Light Source

- For each light source, we can set the RGBA values for the diffuse, specular, and ambient components, and for the light source position:

x y z w

```
vec4 light0_pos = vec4(1.0, 2.0, 3.0, 1.0);
vec4 diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);
```

Red (R) Green (G) Blue (B) Opacity (A or alpha)

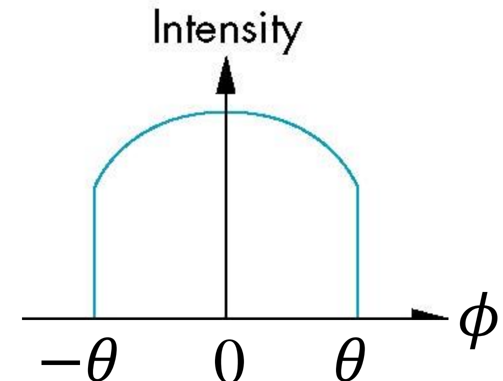
Distance and Direction of a Point Light Source

- The light source color is specified in RGBA
- The light source position is given in homogeneous coordinates:
  - If $w = 1.0$, we are specifying a finite location
  - If $w = 0.0$, we are specifying a parallel source with the given direction vector
- Recall from lecture 15 and 16, we can have a **distance-attenuation coefficient** so objects further from the light source receive less light.
- The distance-attenuation coefficient is usually inversely proportional to the square of the distance d: $1/(a + bd + cd^2)$, where $a, b,$ and c are user-defined constants.

Spotlights

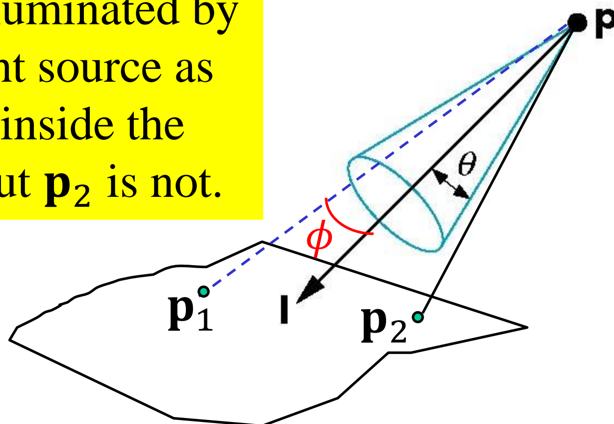
- Spotlights are derived from point sources. Each spotlight thus has a distance and a colour. In addition, it has
  - A direction
  - A cutoff value, θ
  - An attenuation function, usually by $\cos^e\phi$

Large e values ⇒ attenuate faster
Small e values ⇒ attenuate slower



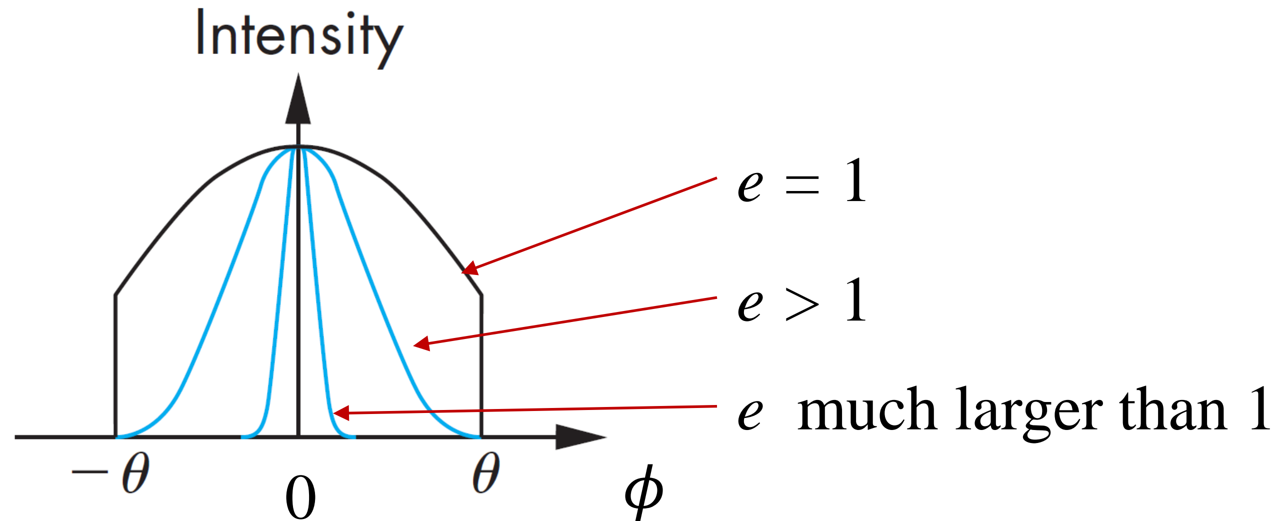
If $\theta = 180^0$ then the spotlight becomes a point source

$\mathbf{p}_1$ is illuminated by the light source as it falls inside the cone but $\mathbf{p}_2$ is not.



Spotlights

- The exponent term e in $\cos^e \phi$ determines how rapidly the light intensity drops off.



Note: the term e here is only a coefficient. It is not equal to exp(1). It is a poor choice of symbol from Angel & Shreiner.

Material Properties

- Material properties should match the terms in the light source model
- Material properties are specified via the ambient, diffuse, and specular reflectivity coefficients (k_a, k_d, and k_s)
- The A value gives opacity, e.g.,

```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(0.8, 0.8, 0.0, 1.0);
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0;
```

R G B A (opacity)

Material Shininess

This is a yellow (not fully saturated though) object. Is it a smooth (shiny) object?

Transparency

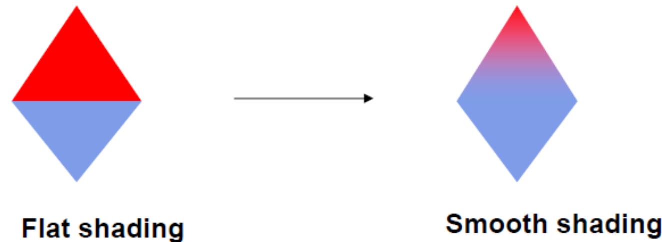
- Material properties are specified as RGBA values.
- The A value can be used to make the surface translucent.
- The default is that all surfaces are opaque.
- This feature can be used in combination with blending

Polygonal Shading Methods

Shading determines color of interior surface pixels of a polygon

There are 2 methods for shading a polygonal mesh:

1. **Flat shading**
2. **Smooth shading**
   - **Gouraud shading**
   - **Phong shading**



Gouraud shading and Phong shading are both smooth shading techniques.

The difference will be clear in the coming slides.

Recall that OpenGL handles only triangles. So the terms "polygon" and "polygonal" from here onward should be interpreted as "triangle" and "triangular".

Image credits :Emmanuel Agu, WPI, CS 543

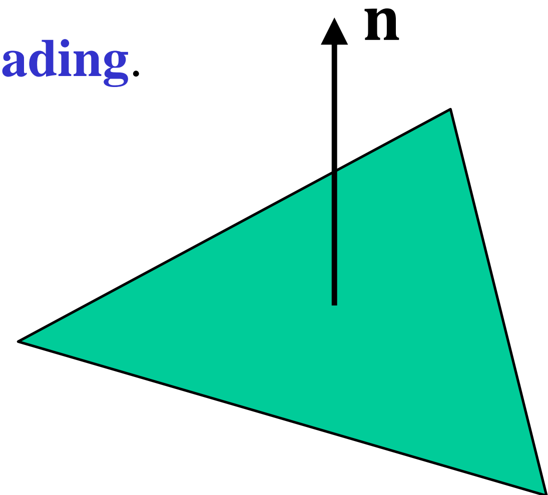
Flat Shading

- Recall that to compute the shading value at a point, we need to know the light vector **l**, the normal vector **n**, and the view vector **v**.

- As we move from one point to the next point inside the polygon, **l** and **v** should vary. In the *flat shading* method, they are assumed to be constant.

⇒ a single shading calculation is performed for each polygon.
⇒ the entire polygon has a single shading value.

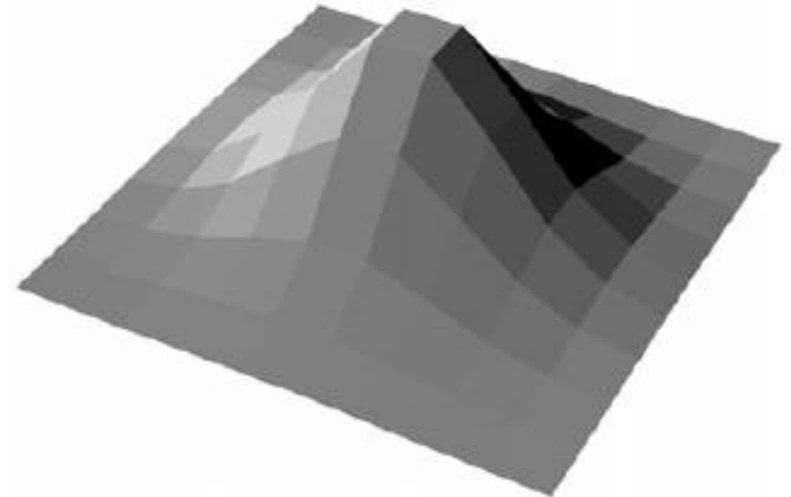
- This technique is thus known as **flat** or **constant shading**.



Flat Shading

- **Advantage**: computationally cheap.
- **Disadvantage**: boundary edges of polygons may show up in the rendered output.
- This shading method is suitable when the viewer and/or light source is far away from the polygon.
- In OpenGL, we specify flat shading as follows:

  glShadeModel(GL_FLAT);



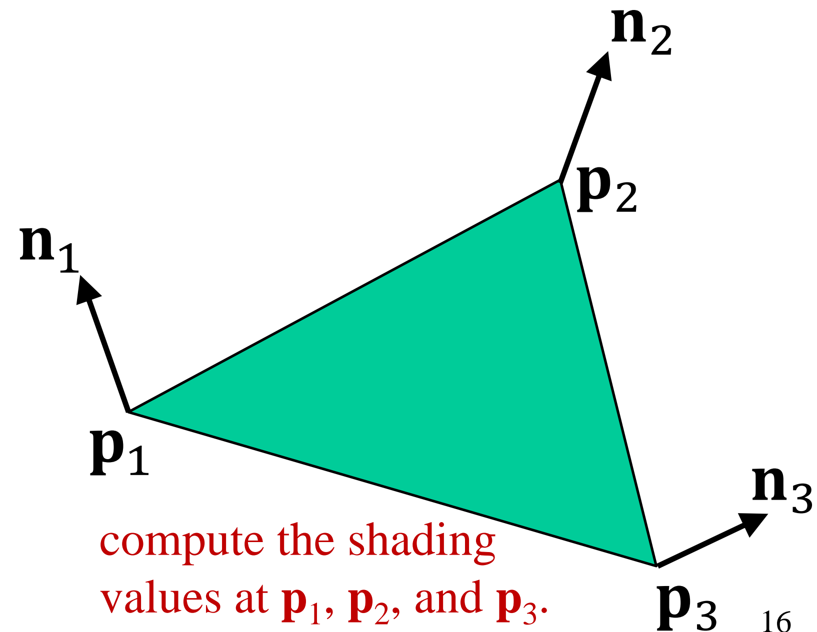
An example of rendering a polygonal mesh using *flat shading*

- Flat shading suffers from "mach band effect"

The Machband describes an effect where the human mind subconsciously increases the contrast between two surfaces with different luminance. The visual system is exaggerating the difference in luminance (contrast) at each edge in order to detect it. source

Smooth Shading or Gouraud Shading

- In the Gouraud shading method,
  - the normal vector at each vertex of the polygon is firstly computed;
  - the shading value at each vertex is then computed using the light vector **l**, normal vector **n**, and view vector **v** at that vertex;
  - shading values of interior points of the polygon are obtained by interpolating the shading values at the vertices.



- As one shading calculation is required for each vertex, it is also known as **per-vertex shading**.

Smooth Shading or Gouraud Shading

- As shading calculation is required for each vertex, it is also known as **per-vertex shading** and sometimes referred to as **per-vertex lighting**.
- **Advantage:** Compared to *flat shading*, the *smooth shading* method gives much better, smoother rendering results
- **Disadvantage:** Compared to *flat shading, smooth shading* is more computationally intensive; however, compared to *Phong shading* (see later), *Gouraud shading* is computationally cheap.
- Smooth shading (or Gouraud shading) is the **default implementation** in OpenGL.
- You can also explicitly set the shading mode as follows:

```
glShadeModel(GL_SMOOTH);
```

Phong Shading

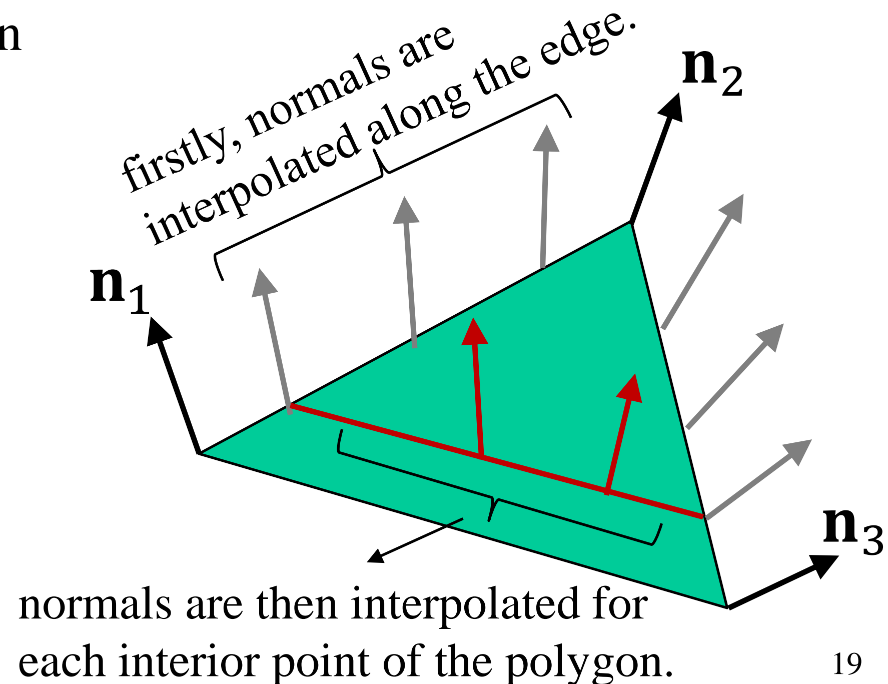
- Although Gouraud shading gives smooth shading, the appearance of Mach bands may still be found, especially at specular highlight regions of the scene.
    - If polygon mesh surfaces have high curvatures, Gouraud shading may show edges

- Phong proposed that
    - Instead of interpolating vertex intensities to get the intensities of the interior of each polygon
    - We interpolate the normals of the vertices of each polygon to get the normal across the interior of each polygon.
    - We then use the interpolated normal at each fragment to compute the intensity of the fragment.

Phong Shading

- Similar to Gouraud shading, the *Phong shading* method firstly computes the normal vector at each vertex of the polygon.

- The differences are then:
  - normals along the edges of the polygon are computed by interpolating the normals at the end-point vertices;
  - normals at interior points of the polygon are obtained by interpolating normals on the edges.



Phong Shading

- The differences are then: …
  - shading value for each point of the polygon is finally computed using the normal, light vector, and view vector at that point.

- As shading calculation is performed for each fragment, Phong shading is also known as **per-fragment shading** or **per-fragment lighting**.

- **Advantage:** Very good rendering results, especially at the highlight regions of highly specular (shiny) surfaces

- **Disadvantage:** Too computationally expensive. Only done in off-line processing. Not suitable for interactive graphics. However, it is reported that the latest graphics cards perform Phong shading in real-time.

Recall that a fragment is a *potential pixel* which carries additional information such as depth value, colour value, opacity, etc.

Comparing Gouraud Shading and Phong Shading

- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges.
- Both need data structures to represent meshes so we can obtain vertex normals.

Gouraud shading or *smooth shading* or *per-vertex shading*

Phong shading or *per-fragment shading*

Note the difference in the highlight region of the teapot. Gouraud shading gives a flat impression in the region.

Example 1: Vertex Shader for Gouraud Shading method

```
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;  //vertex shade
```

- Here, the vertex shade (or vertex colour) is computed in the *vertex shader.*
- The colour is then passed down the pipeline where the *rasterizer* carries out the interpolation of the vertex colour.
- This is **Gouraud shading** or **per-vertex shading**.

(light * reflectivity)

```
// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

These terms (ambient, diffuse and specular) are computed in the application (the .cpp file) and passed to the *vertex shader* as *uniform* variable.

out variables will be interpolated by the rasterizer.

Example 1: Vertex Shader for Gouraud Shading (cont.)

```
void main()
{
   // Transform vertex  position into eye coordinates
   vec3 pos = (ModelView * vPosition).xyz;

   vec3 L = normalize( LightPosition.xyz - pos ); \\ light vector
   vec3 V = normalize( -pos ); \\ view vector
   vec3 H = normalize( L + V );  \\ half-way vector

   // Transform vertex normal into eye coordinates
   vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
```

Built-in function that can be used in the vertex and fragment shaders.

Example 1: Vertex Shader for Gouraud Shading (cont.)

```
// Compute terms in the illumination equation
    vec4 ambient = AmbientProduct;

    float Kd = max( dot(L, N), 0.0 );  \\ (l · n)
    vec4  diffuse = Kd*DiffuseProduct;

    float Ks = pow( max(dot(N, H), 0.0), Shininess );  \\ (n · h )^β
    vec4  specular = Ks * SpecularProduct;
// discard the specular highlight if the light is behind the vertex
    if( dot(L, N) < 0.0 ):  // since we don't use perfect reflector "R" in Blinn-phong model, dot(V,R)  is not used here
            specular = vec4(0.0, 0.0, 0.0, 1.0);

    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```

β (Shininess)

Blinn_Phong Model

Recall that the total shading is (see previous lecture):

$$I = k_d I_d (\mathbf{l} \cdot \mathbf{n}) + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Example 1: Fragment Shader for Gouraud Shading

```
// fragment shader

in vec4 color;

void main()
{
    gl_FragColor = color;
}
```

All the hard work has been done by the application code and the *vertex shader*.

The *fragment shader* simply takes the interpolated colour and assigns it to the fragment.

Example 2: Vertex Shader for the Phong Shading Method

```
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
```

- In these examples, the *vertex shader* passes down fN, fV, and fL variables to the rasterizer to interpolate.
- The fragment colour is computed in the *fragment shader* using the interpolated normal, light, and view vectors.
- This is **Phong shading** or **per-fragment shading.**

```
// output values that will be interpolated per-fragment
out vec3 fN;
out vec3 fV;
out vec3 fL;
```

Declare variables n, v, l as out in vertex shader. These *out* variables will be interpolated by the rasterizer further down the pipeline.

```
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
```

Example 2: Vertex Shader for Phong Shading (cont.)

```
void main()
{
    fN = (ModelView * vNormal).xyz;
    fV = - (ModelView * vPosition).xyz; //notice the negative sign
    fL = LightPosition.xyz – (ModelView * vPosition).xyz;

    gl_Position = Projection * ModelView * vPosition;
}
```

Example 3: Fragment Shader for Phong Shading (cont.)

```
// fragment shader

// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fL;
in vec3 fV;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;
```

Example 3: Fragment Shader for Phong Shading (cont.)

```
void main()
{
   // Normalize the input lighting vectors

   vec3 N = normalize(fN);
    vec3 V = normalize(fV);
    vec3 L = normalize(fL);

    vec3 H = normalize( L + V );
    vec4 ambient = AmbientProduct;
```

Use interpolated variables n, v, l in fragment shader

Example 3: Fragment Shader for Phong Shading (cont.)

```
    float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd*DiffuseProduct;

    float Ks = pow(max(dot(N, H), 0.0), Shininess);
    vec4 specular = Ks*SpecularProduct;

    // discard the specular highlight if the light is behind the vertex
    if( dot(L, N) < 0.0 ):
            specular = vec4(0.0, 0.0, 0.0, 1.0);

    gl_FragColor = ambient + diffuse + specular;
    gl_FragColor.a = 1.0;
}
```

Recall that the total shading is (see previous lecture):

$$I = k_d I_d (\mathbf{l} \cdot \mathbf{n}) + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Exercises

Going back to the OpenGL pipeline architecture and the shaders (vertex and fragment) that we learned from previous lectures. Frome what we learned here, you should be able to answer the following questions:

- If we want to do per-vertex shading
  - Which shader should compute the colour of each pixel?
  - What variables should be declared as *varying* in both shaders?

- If we want to do per-fragment shading
  - Which shader should compute the colour of each pixel?
  - What variables should be declared as *varying* in both shaders?

Hint: the Rasterizer of the pipeline is responsible for interpolating the output variables from the vertex shader. The interpolated values are passed to the fragment shader.

Further Reading

"Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL" by Edward Angel and Dave Shreiner, 6th Ed, 2012

- Sec. 5.5. Polygonal Shading (including all subsections, covering Flat Shading, Smooth or Gouraud Shading, and Phong Shading)
- Appendix A.7 Per-Fragment Lighting of Sphere Model