Comp 410/510

Computer Graphics

Spring 2023

OpenGL Shading

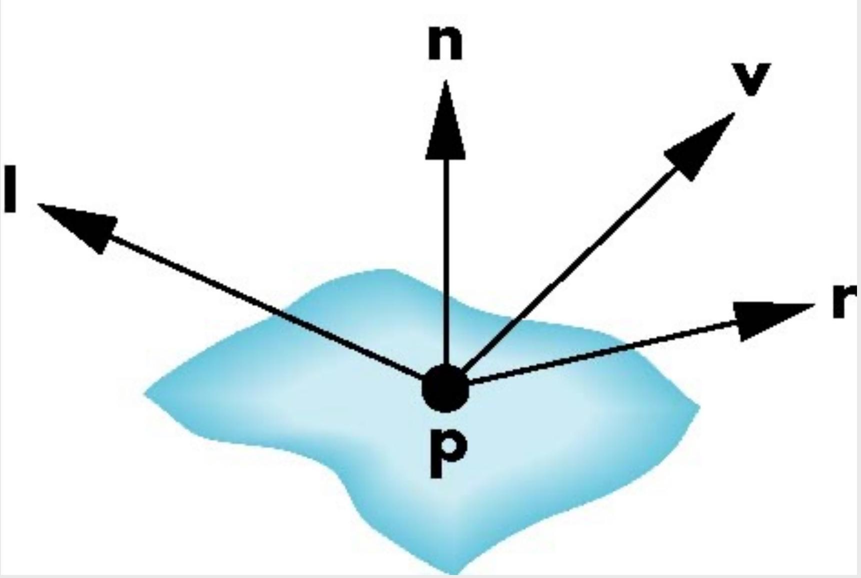
Objectives

- Introduce the OpenGL shading methods
  - per vertex vs per fragment shading

- Discuss polygonal shading
  - Flat shading
  - Phong shading
  - Gouraud shading

Recap: Phong Illumination Model

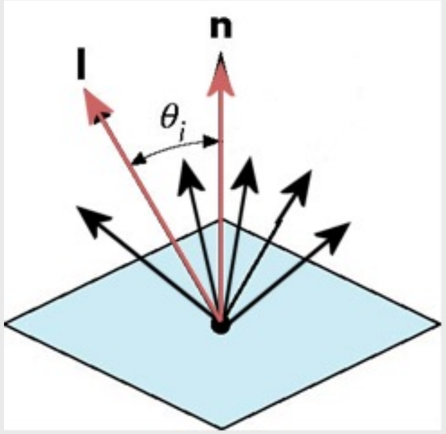
- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source
  - To viewer
  - Normal
  - Perfect reflector



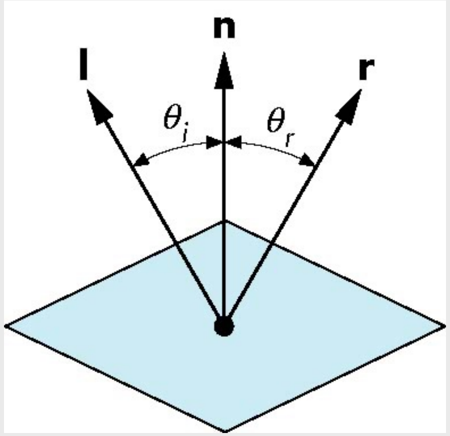
$$I = \frac{1}{a + bd + cd^2} \left(k_d L_d \, \boldsymbol{l} \cdot \boldsymbol{n} + k_s L_s (\boldsymbol{v} \cdot \boldsymbol{r})^{\alpha} \right) + k_a L_a$$

Recap: Diffuse vs Specular

- Perfectly diffuse (Lambertian) surfaces
  - Light scattered equally in all directions
  - Amount of light reflected is proportional to the vertical component of incoming light
- Perfectly specular surfaces
  - Ideal reflectors (mirror-like)
  - Angle of incidence = Angle of reflection



perfectly diffuse



perfectly specular

OpenGL shading

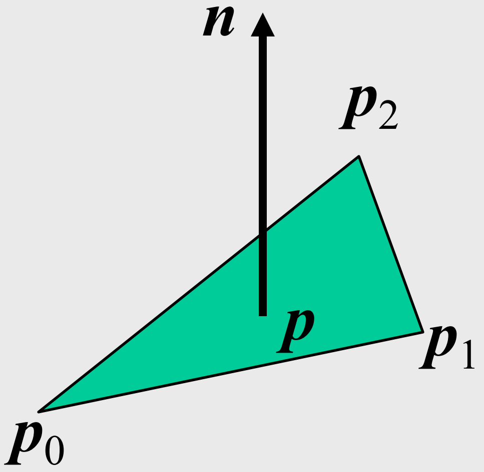
- We need
  - Normals
  - Material properties
  - Lights
- State-based shading functions have been deprecated (such as glNormal, glMaterial, glLight)
- Now use shaders

Computation of Normal for Triangles

Plane equation: $\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{p}_0) = 0$

Normal: $\boldsymbol{n} = (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)$

Normalize by $\boldsymbol{n} \leftarrow \boldsymbol{n} / |\boldsymbol{n}|$



Note that right-hand rule determines outward face

Normalization

- Several dot products involved in lighting calculation
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length
- But need to be careful since
  - affine transformations do not generally preserve length
  - e.g., scaling transformation
- GLSL has a normalization function

$$I = \frac{1}{a + bd + cd^2} \left(k_d L_d \, \mathbf{l} \cdot \mathbf{n} + k_s L_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} \right) + k_a L_a$$

Defining a Light Source

- For each light source, we can set an RGBA for the diffuse, specular, and ambient parts:

```
vec4 diffuse0[]={1.0, 0.0, 0.0, 1.0};
vec4 ambient0[]={1.0, 0.0, 0.0, 1.0};
vec4 specular0[]={1.0, 0.0, 0.0, 1.0};
```

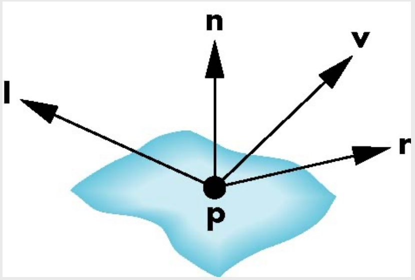
$$I = \frac{1}{a + bd + cd^2} \left(k_d L_d \, \boldsymbol{l} \cdot \boldsymbol{n} + k_s L_s (\boldsymbol{v} \cdot \boldsymbol{r})^{\alpha} \right) + k_a L_a$$

Point vs Directional Light Source

- We need light position to compute $\boldsymbol{l}$ and $\boldsymbol{r}$ vectors
- The light position is given in homogeneous coordinates
  - If $w = 1.0$, we are specifying a point source
  - If $w = 0.0$, we are specifying a parallel source with the given direction vector

```
vec4 light0_pos[]={1.0, 2.0, 3,0, 1.0};
```

$$I = \frac{1}{a + bd + cd^2} \left(k_d L_d \, \boldsymbol{l} \cdot \boldsymbol{n} + k_s L_s (\boldsymbol{v} \cdot \boldsymbol{r})^{\alpha} \right) + k_a L_a$$



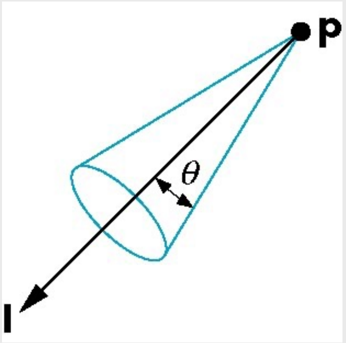
Distance term

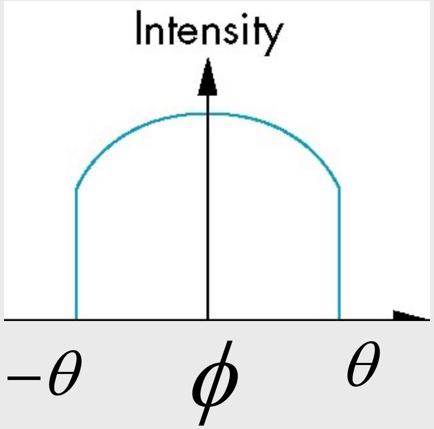
- In the distance term, d is the distance from the point being rendered to the light source
- a, b and c are usually uniform variables to set

$$I = \frac{1}{a + bd + cd^2}\left(k_d L_d \, \boldsymbol{l} \cdot \boldsymbol{n} + k_s L_s (\boldsymbol{v} \cdot \boldsymbol{r})^{\alpha} \right) + k_a L_a$$

Spotlights

- Can be implemented as a point light source with
  - Direction: $\boldsymbol{l}$
  - Cutoff: θ
  - Attenuation proportional to $\cos^{\alpha}\phi$





Moving Light Sources

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

- Depending on how we set the light position (or direction) within the code, 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

Global Ambient Light

- Ambient light depends on color and amount of light sources
  - A red light in a white room will cause a red ambient term that disappears when the light is turned off

- An additional **global** ambient term is often used and helpful, such as

```
vec4 global_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
```

Since these numbers yield a small amount of white ambient light, even if you don't add a specific light source to your scene, you can still see the objects.

Material Properties

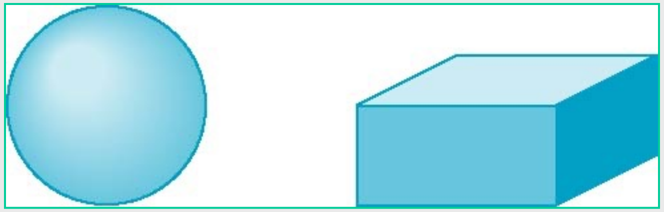
- Material properties that match the terms in Phong illumination model:

```
vec4 ambient[] = {0.2, 0.2, 0.2, 1.0};
vec4 diffuse[] = {1.0, 0.8, 0.0, 1.0};
vec4 specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat shine = 100.0
```

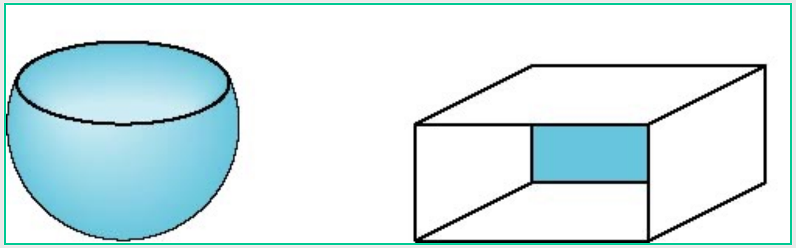
$$I = \frac{1}{a + bd + cd^2} \left(k_d L_d \, \boldsymbol{l} \cdot \boldsymbol{n} + k_s L_s (\boldsymbol{v} \cdot \boldsymbol{r})^{\alpha} \right) + k_a L_a$$

How to treat front and back faces?

- Every triangle has a front and a back face (specified by the order of vertices)
- For many objects, we never see the back face, so we don't care how or whether it's rendered (normally both faces are rendered in the same way)
- If it matters, we can handle it in shader
  - Compute two different shades for each vertex in vertex shader, one for front face and the other for back
  - Then use boolean `gl_FrontFacing` to determine which one to use in fragment shader



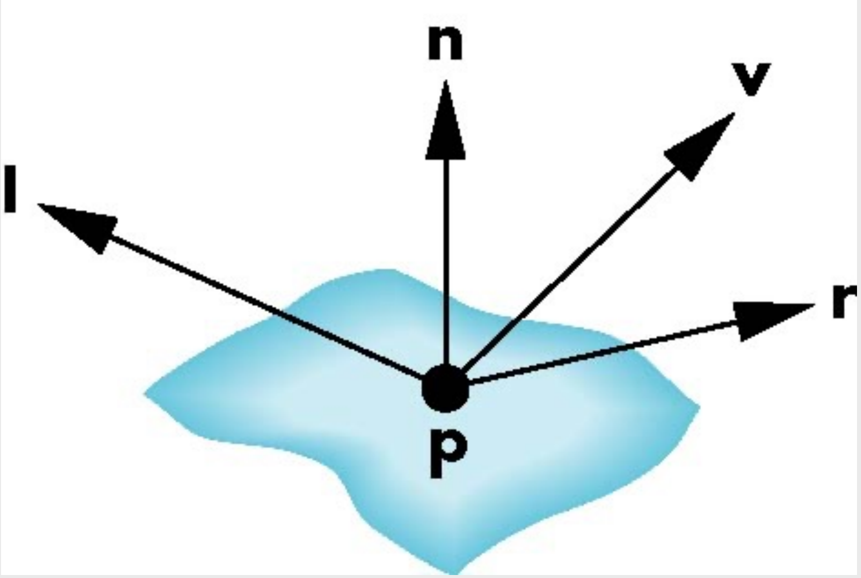
back faces not visible



back faces visible

Recap: Modified Phong Illumination Model

- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source
  - To viewer
  - Normal
  - Perfect reflector



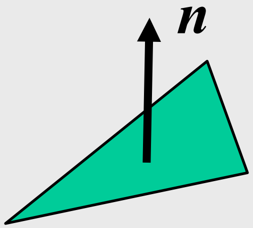
$$I = \frac{1}{a + bd + cd^2} \left(k_d L_d \, \boldsymbol{l} \cdot \boldsymbol{n} + k_s L_s (\boldsymbol{n} \cdot \boldsymbol{h})^{\beta} \right) + k_a L_a$$

$$\boldsymbol{h} = (\boldsymbol{l} + \boldsymbol{v}) \,/\, |\, \boldsymbol{l} + \boldsymbol{v} \,|$$

Polygonal Shading

- In per vertex shading, shading calculations are done for each vertex
  - Vertex shades become vertex colors, and can be sent to the vertex shader as a vertex attribute
  - Alternately, we can send the parameters to vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as an `in` variable, that is smooth shading

Polygon Normals



- Polygons normally have one single normal
  - Shades at the vertices as computed by the Phong illumination model appear almost the same
  - Identical for a distant viewer and distant light source

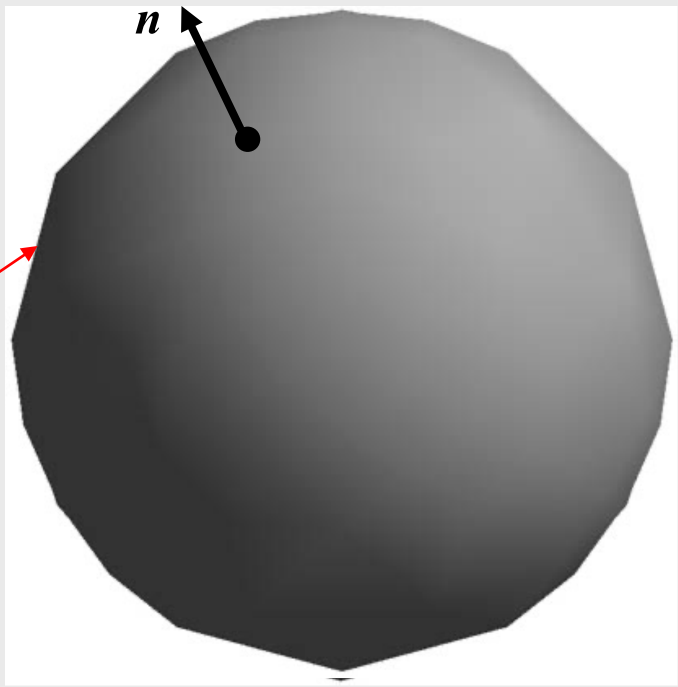
$$I = \frac{1}{a+bd+cd^2}\left(k_d L_d\, \boldsymbol{l}\cdot\boldsymbol{n} + k_s L_s (\boldsymbol{v}\cdot\boldsymbol{r})^{\alpha}\right) + k_a L_a$$

- But consider the model of a sphere:

- Not smooth

Smooth Shading

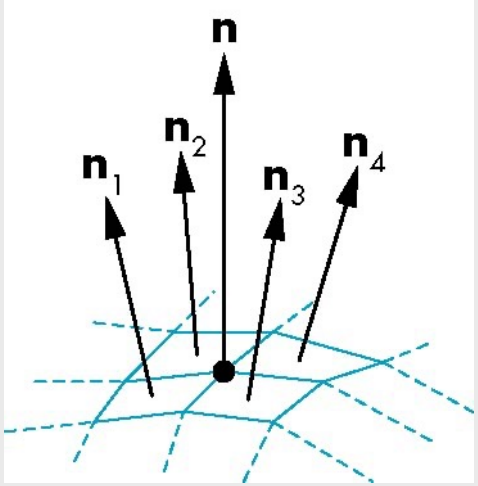
- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin
  - $\boldsymbol{n} = \boldsymbol{p}$
- Now smooth shading works
- But note *silhouette edges*



Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically
- For polygonal models, Gouraud proposed to use the average of normals around a mesh vertex

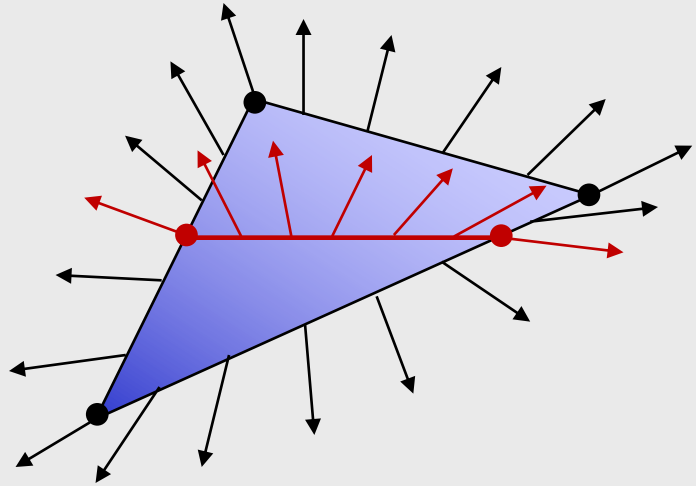
$$\boldsymbol{n} = \frac{\boldsymbol{n}_1 + \boldsymbol{n}_2 + \boldsymbol{n}_3 + \boldsymbol{n}_4}{|\,\boldsymbol{n}_1 + \boldsymbol{n}_2 + \boldsymbol{n}_3 + \boldsymbol{n}_4\,|}$$



Gouraud vs Phong Shading

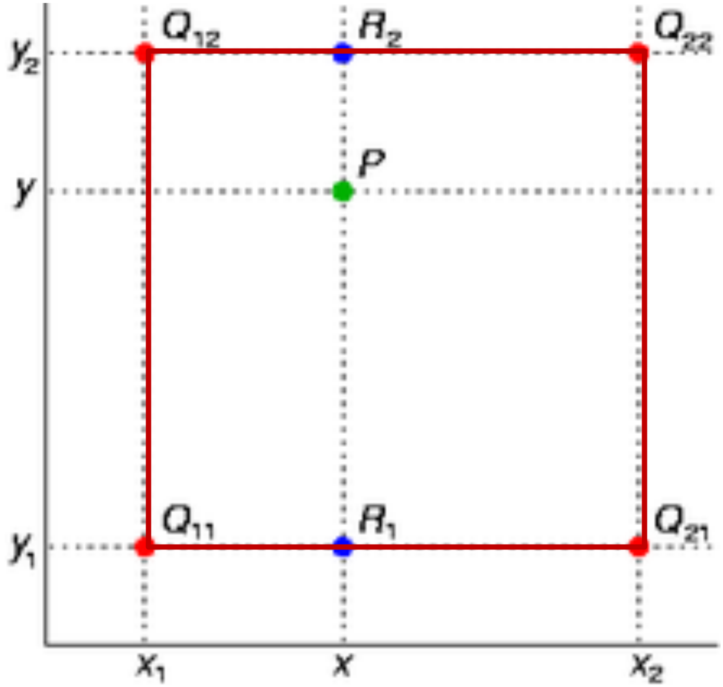
- **Gouraud (per vertex) shading**
  - Find average normal vector at each vertex
  - Apply Phong (or modified Phong) model at each vertex
  - Interpolate vertex shades across each polygon

- **Phong (per fragment) shading** (not to confuse with Phong illumination model)
  - Find average normal vector at each vertex
  - Interpolate vertex normals across polygons
  - Apply Phong (or modified Phong) model at each fragment to find shades



vertex normal
interpolation

Bilinear Interpolation (across a polygon)



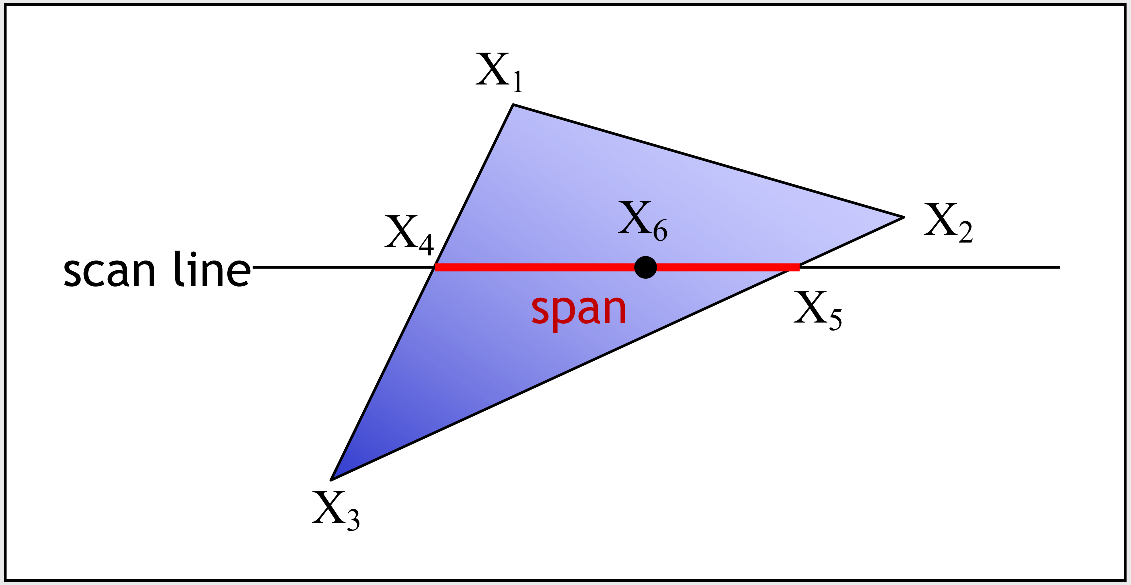
$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2).$$

Filling with Interpolation

- $X_1 X_2 X_3$ can be color attribute, shade, normal, texture coordinate, etc.
- X_4 is determined by interpolating between X_1 and X_3
- X_5 is determined by interpolating between X_2 and X_3
- Then interpolate between X_4 and X_5 along the span



$$X_4 = \alpha_1 X_1 + (1 - \alpha_1) X_3$$

$$X_5 = \alpha_2 X_2 + (1 - \alpha_2) X_3$$

$$X_6 = \alpha_3 X_4 + (1 - \alpha_3) X_5$$

α's are weights based on the distances of the underlying point to the vertices (see the previous slide).

Interpolation is carried out in screen coordinates during rasterization

Comparison (Gouraud vs Phong)

- If the polygon mesh approximates a surface with high curvatures, *Phong* shading may look smoother while *Gouraud* shading may show edges
- *Phong* shading requires much more work than *Gouraud* shading
  - Until recently not available in real time systems
  - Now can be done using fragment shaders
- Both need data structures to represent meshes so that we can obtain vertex normals from incident triangles in the neighborhood

Gouraud shading example

vertex shader

$$I = (\, k_d L_d\, \boldsymbol{l} \cdot \boldsymbol{n} \;+ k_s L_s\, (\boldsymbol{n} \cdot \boldsymbol{h}\,)^{\beta}\,) + k_a L_a$$

```
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct; // k_a * L_a, k_d * L_d, k_s * L_s
uniform mat4 ModelView, Projection;
uniform vec4 LightPosition;
uniform float Shininess;
void main(){
    // Transform vertex position into camera coord.
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize( LightPosition.xyz - pos ); //  assume a point light source and
                                                     compute the direction l
    vec3 V = normalize( -pos ); // viewer direction v (camera is at origin)
    vec3 H = normalize( L + V ); // halfway vector h
.
    // Transform vertex normal n into camera coord.
    vec3 N = normalize(ModelView * vec4(vNormal, 0.0)).xyz;

    // Compute terms in the illumination equation
    vec4 ambient = AmbientProduct; // k_a * L_a

    float Kd = max( dot(L, N), 0.0 ); //set diffuse to 0 if light is behind the surface point
    vec4 diffuse = Kd*DiffuseProduct; // k_d * L_d

    float Ks = pow( max(dot(N, H), 0.0), Shininess ); // ignore specular component if negative
    vec4 specular = Ks * SpecularProduct; // k_s * L_s

    //ignore specular component also if light is behind the surface point
    if( dot(L, N) < 0.0 ) specular = vec4(0.0, 0.0, 0.0, 1.0);

    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
}
```

Gouraud shading example

fragment shader

```
in  vec4 color;
out vec4 fcolor

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

Phong shading example

vertex shader

$$I = (\, k_d L_d\, \boldsymbol{l} \cdot \boldsymbol{n} \; + k_s L_s\, (\boldsymbol{n} \cdot \boldsymbol{h}\,)^{\beta}\,) + k_a L_a$$

```
in   vec4 vPosition;
in   vec3 vNormal;
// output values that will be interpolated per-fragment
out  vec3 fN;
out  vec3 fV;
out  vec3 fL;

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

void main()
{
    // Transform vertex position into camera coord.
    vec3 pos = (ModelView * vPosition).xyz;

    fN = (ModelView*vec4(vNormal, 0.0)).xyz; // normal direction in camera coordinates
    fV = -pos; // viewer direction in camera coordinates

    fL = LightPosition.xyz; // light direction if directional light source
    if( LightPosition.w != 0.0 ) fL = LightPosition.xyz - pos;  // if point light source

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

Phong shading example

fragment shader

$$I = (\, k_d L_d\, \boldsymbol{l} \cdot \boldsymbol{n} \; + k_s L_s\, (\boldsymbol{n} \cdot \boldsymbol{h}\,)^{\beta}\,) + k_a L_a$$

```
// per-fragment interpolated values from the vertex shader
in  vec3 fN, fL, fV;
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform float Shininess;
out fcolor;

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;

        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's behind the vertex
        if( dot(L, N) < 0.0 ) specular = vec4(0.0, 0.0, 0.0, 1.0);

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