Midterm Statistics

Highest: 106

Median: 93

Mean: 92.52

Standard Deviation: 8.61

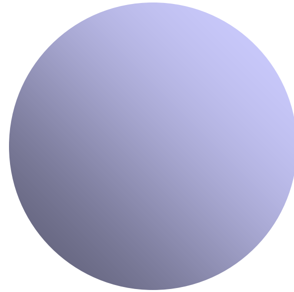
Topics

Lighting and shading

Shading in OpenGL

Shading

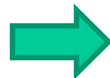
Why does the image of a real sphere look like



Light-material interactions cause each point to have a different color or shade

Need to consider

- Light sources
- Material properties
- Location of viewer
- Surface orientation

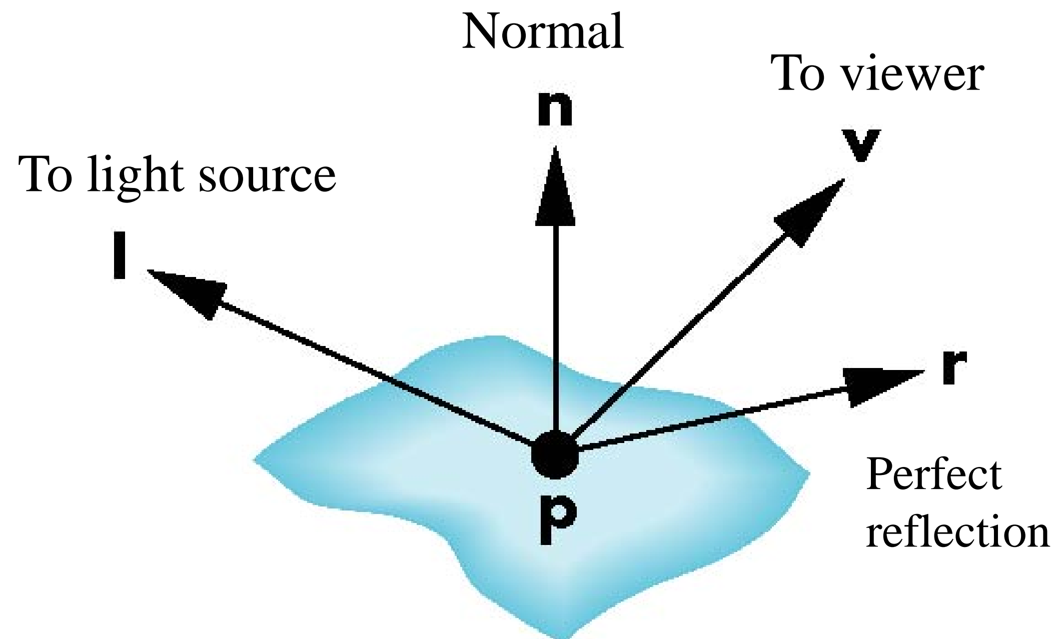


4 Key elements of image formation

Phong Model

Uses four unit vectors to calculate a color on a surface

- Surface normal **n**
- To viewer **v**
- To light source **l**
- Perfect reflector **r**



Phong Model

A simple model that can be computed rapidly

Each light source has three components

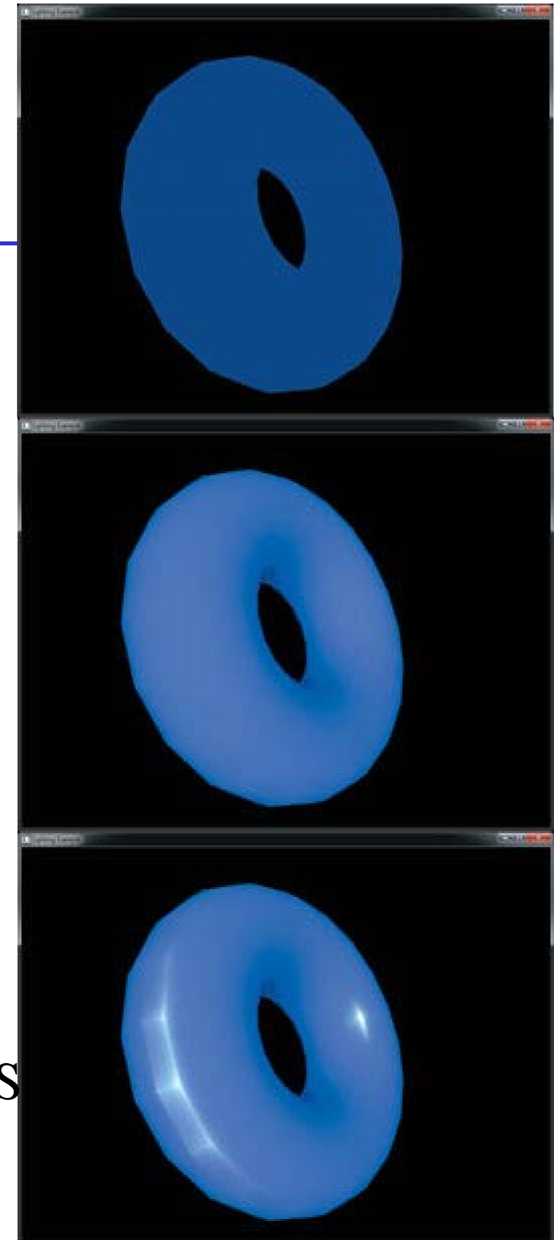
- Ambient
- Diffuse
- Specular

Point light source (Diffuse, Specular)

A

A+D

A+D+S



Shreiner et al

Computation of Vectors

Need to compute the four vectors

- Surface normal **n**
- To viewer **v**
- To light source **l**
- Perfect reflector **r**

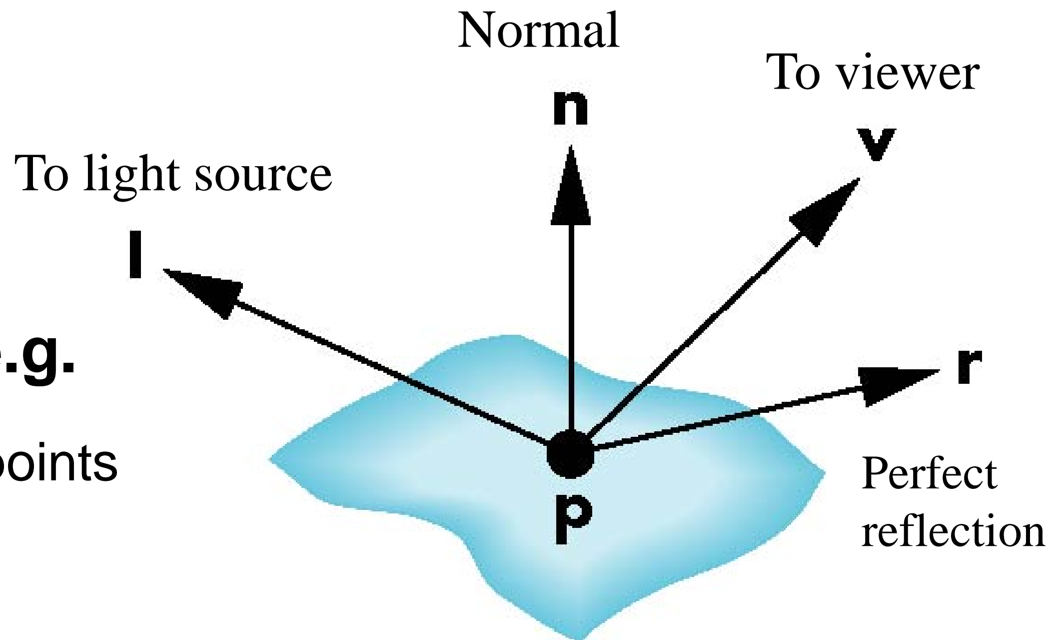
Simplifications can apply, e.g.

- Normal can be the same for all points

a flat polygon

- Light direction is the same for all points

if the light is far away from the surface



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Computation of Vectors

l and **v** are specified by the application

h can be computed from **l** and **v**

How to calculate **n**?

Depending on surface

OpenGL leaves determination of normal to application, e.g., the obj file contains the normals

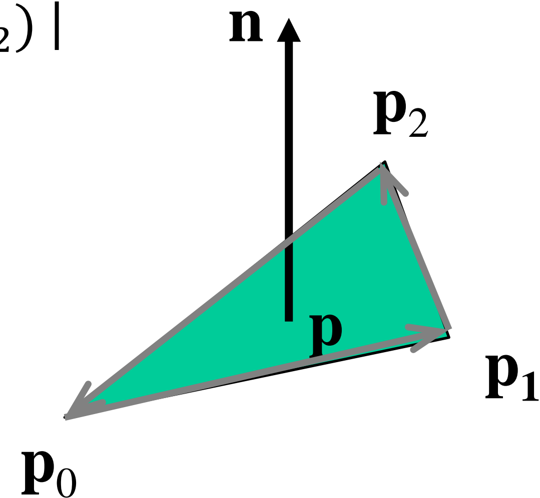
Plane Normals

Plane can be determined by three points P_1, P_2, P_3 or normal $\mathbf{n}$ and P_0

Given three noncolinear points, e.g., the three vertices of a triangle P_1, P_2, and P_3, the outfacing normal can be obtained by

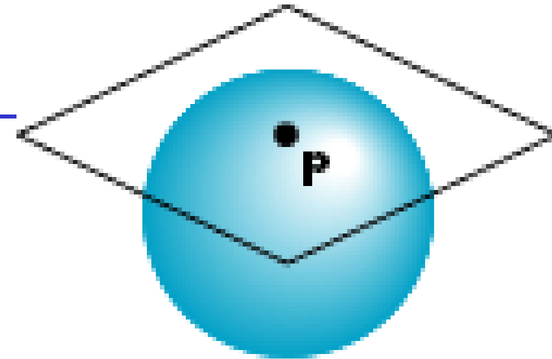
$$\mathbf{n} = \frac{(P_2 - P_1) \times (P_0 - P_2)}{|(P_2 - P_1) \times (P_0 - P_2)|}$$

Order of vectors is important!



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Normal to Sphere



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

How we compute normals for curved surfaces?

Depend on how we model the surface.

Implicit function *of a unit sphere centered at the origin*

$$f(x, y.z) = x^2 + y^2 + z^2 - 1 = 0$$

Or in vector form

$$f(\mathbf{p}) = \mathbf{p} \cdot \mathbf{p} - 1 = 0$$

Normal is given by gradient

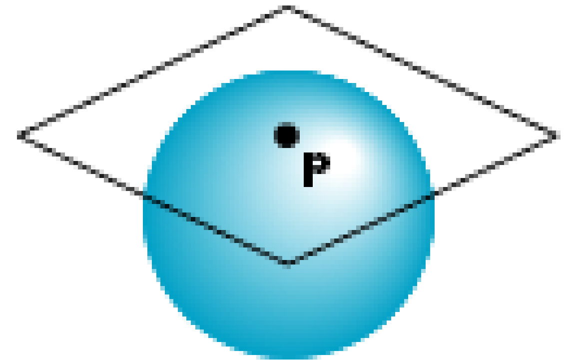
$$\mathbf{n}' = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \\ 2z \end{bmatrix} = 2\mathbf{p} \quad \Rightarrow \quad \mathbf{n} = \frac{\mathbf{n}'}{|\mathbf{n}'|} = \mathbf{p}$$

Parametric Form

Parametric form of a sphere, $-\frac{\pi}{2} < u, v < \frac{\pi}{2}$

independent

$$\begin{cases} x = x(u,v) = \cos u \sin v \\ y = y(u,v) = \cos u \cos v \\ z = z(u,v) = \sin u \end{cases}$$



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Normal calculated from the ***tangent plane***

Tangent plane determined by vectors

$$\frac{\partial \mathbf{p}}{\partial u} = \begin{bmatrix} \frac{\partial x}{\partial u} \\ \frac{\partial y}{\partial u} \\ \frac{\partial z}{\partial u} \end{bmatrix} \quad \frac{\partial \mathbf{p}}{\partial v} = \begin{bmatrix} \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial v} \\ \frac{\partial z}{\partial v} \end{bmatrix}$$

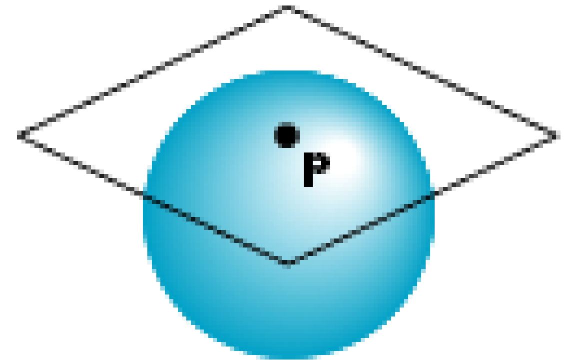
Normal given by cross product

$$\mathbf{n} = \frac{\frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v}}{\left| \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v} \right|}$$

Parametric Form

Parametric form of a sphere, $-\frac{\pi}{2} < u, v < \frac{\pi}{2}$

$$x = x(u,v) = \cos u \sin v$$
$$y = y(u,v) = \cos u \cos v$$
$$z = z(u,v) = \sin u$$



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Tangent plane determined by vectors

$$\frac{\partial \mathbf{p}}{\partial u} = \begin{bmatrix} \frac{\partial x}{\partial u} \\ \frac{\partial y}{\partial u} \\ \frac{\partial z}{\partial u} \end{bmatrix} = \begin{bmatrix} -\sin u \, sin\, v \\ -sin\, u \, cos\, v \\ \cos u \end{bmatrix} \quad \frac{\partial \mathbf{p}}{\partial v} = \begin{bmatrix} \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial v} \\ \frac{\partial z}{\partial v} \end{bmatrix} = \begin{bmatrix} \cos u \cos v \\ -\cos u \sin v \\ 0 \end{bmatrix}$$

Normal given by cross product

$$\mathbf{n}' = \cos u \begin{bmatrix} \cos u \sin v \\ \cos u \cos v \\ \sin u \end{bmatrix} = (\cos u)\,\mathbf{p}$$

$$\mathbf{n} = \mathbf{p}$$

General Case

We can compute parametric normals for other simple cases

- Quadrics
- Parameteric polynomial surfaces
  - Bezier surface patches (Chapter 10)

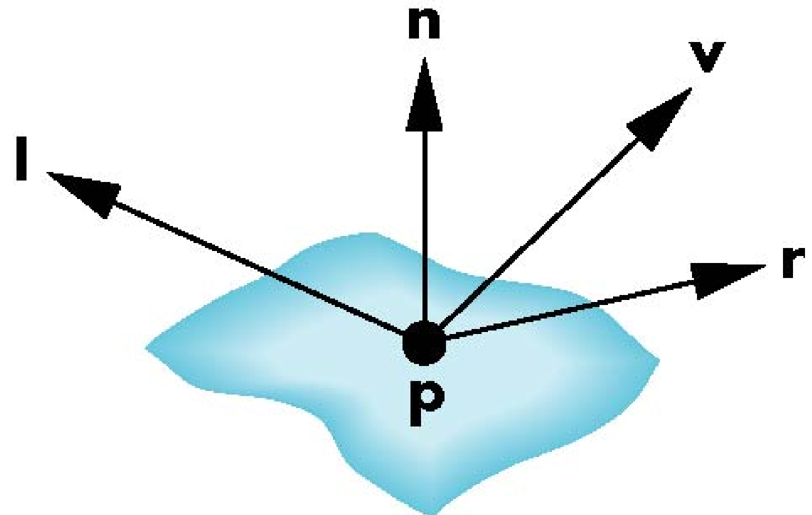
Recall Blinn-Phong Model

For each light source and each color component, the Blinn-Phong model can be written as

$$I = \frac{k_d\, L_d max(\mathbf{l} \cdot \mathbf{n}, 0) + k_s L_s \max\left((\mathbf{n} \cdot \mathbf{h})^{\beta}, 0\right)}{a + bd + cd^2} + k_a I_a$$

For each color component we add

contributions from all sources



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

OpenGL shading

Need

- Lights
- Material properties – reflection coefficients
- Vectors: $\mathbf{l}$, $\mathbf{n}$, $\mathbf{v}$, $\mathbf{h}$ – should be normalized to unit vectors

- **State-based shading functions have been deprecated (glNormal, glMaterial, glLight)**

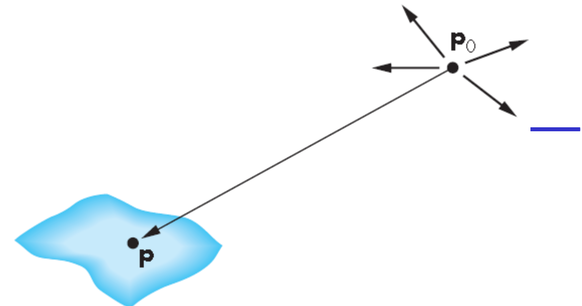
- **Calculation can be done in**
  - Application
  - Vertex shader
  - Fragment shader

Specifying a Point Light Source

For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and specify the position

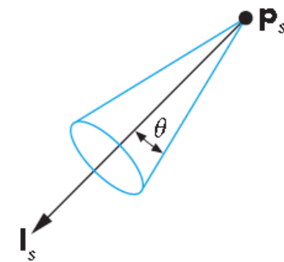
```
vec4 diffuse0 =vec4(1.0, 0.0, 0.0, 1.0);
vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);

vec4 light0_pos =vec4(1.0, 2.0, 3,0, 1.0);
```

Recall Simple Light Sources



Point source

- Emits light equally in all direction
- Model with position and color – proportional to the inverse square of the distance
- Distant source = infinite distance away (parallel)



Spotlight

- Restrict light from ideal point source

Ambient light

- Uniform illumination everywhere in scene -- An intensity identical at every point

Point Source

The position is given in homogeneous coordinates

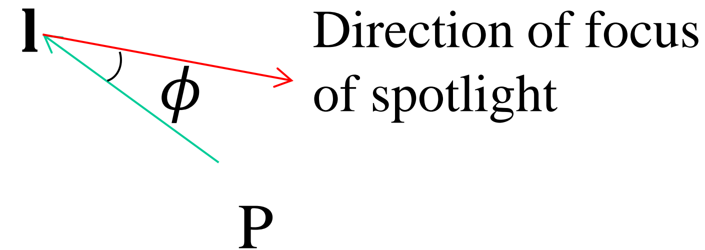
- If w =1.0 -- a regular point light source at a finite location
- If w =0.0 – a distant light source at infinity = a parallel source with the given direction vector

The coefficients in distance terms are usually quadratic ($\frac{1}{a+bd+cd^2}$) where *d* is the distance from the point being rendered to the light source

Spotlights

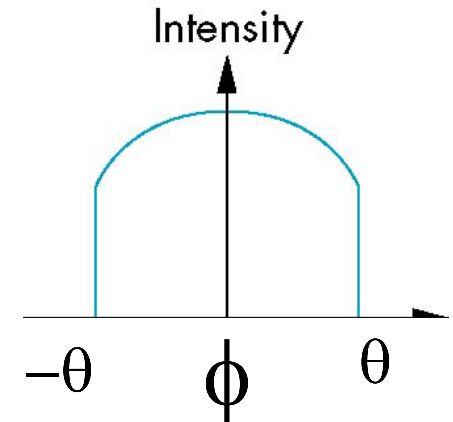
Derive from point source

- Angle between **l** (direction to the light) and the focus of spotlight
- Cutoff angle θ
- Attenuation proportional to $cos^{\alpha}\phi$,




```
// how close are we to being in the spot?
float spotCos = dot(lightDirection, -
ConeDirection);
// attenuate more, based on spot-relative
position
if (spotCos < SpotCosCutoff)
attenuation = 0.0;
else
attenuation *= pow(spotCos,SpotExponent);
```

Shreiner et al., OpenGL Programming Guide



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Global Ambient Light

Ambient light depends on

- color of light sources
- Reflective properties of surfaces
- E.g., a red light in a white room will cause a red ambient term that disappears when the light is turned off

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

Material Properties

Material properties should match the terms in the light model and are specified as RGBA values

The A value can be used to make the surface translucent

The default is that all surfaces are opaque regardless of A

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

Emissive Term

A *light source* is allowed in the scene, e.g., the moon, in OpenGL

-- specify an emissive component to a material

```
color4 emission = color4(0.0, 0.3, 0.3, 1.0);
```

This self-emission component

- unaffected by any other sources
- Not affect other surfaces

Structure to Hold Material Properties

```
struct MaterialProperties {
vec3 emission; // light produced by the
material
vec3 ambient; // what part of ambient light
is reflected
vec3 diffuse; // what part of diffuse light
is scattered
vec3 specular; // what part of specular
light is scattered
float shininess; // exponent for sharpening
specular reflection
// other properties you may desire
};
```

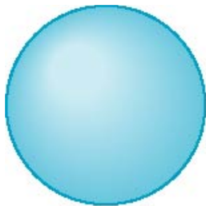
Shreiner et al., OpenGL Programming Guide

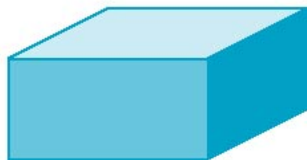
Front and Back Faces

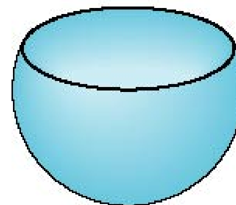
Every face has a front and back

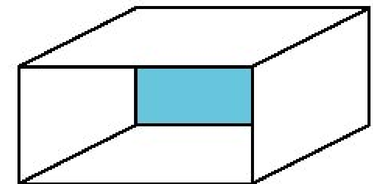
For many objects, we never see the back face so we don't care how or if it's rendered

If it matters, we can handle in shader









back faces not visible

back faces visible

Polygonal Shading

In per vertex shading, shading calculations are done for each vertex

- Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
- Alternately, we can send the parameters to the vertex shader and have it compute the shade

By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)

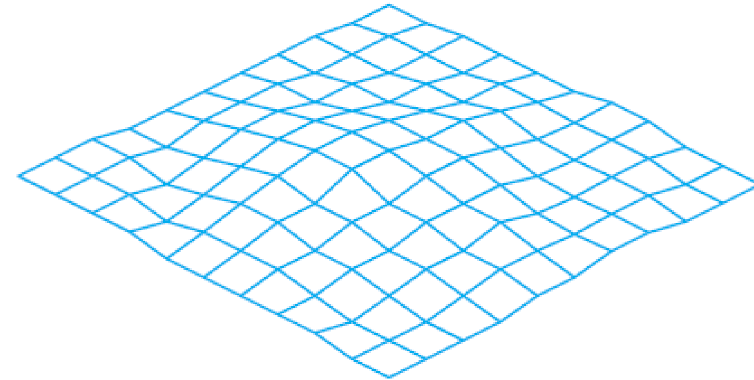
We can also use uniform variables to shade with a single shade (flat shading)

Polygonal Shading – Flat Shading

Need to calculate $\mathbf{n}, \mathbf{v}, \mathbf{l}$ for every point on a surface

Simplifications:

- $\mathbf{n}$ is a constant for a flat polygon and can be precomputed in an obj file
- $\mathbf{v}$ is a constant for a distant viewer
- $\mathbf{l}$ is a constant for a distant light



E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

If all the three vectors are constants,

the shading calculation can be done once for each polygon

⇒ Every point has the same color/shade on the polygon

Polygonal Shading

Triangles have a single normal

- Flat shading
- Shades at the vertices as computed by the Phong model can be almost same
- Identical for a distant viewer (default) or if there is no specular component

Consider model of sphere

- Normals of different triangles change significantly
- Shades of sphere change discontinuously

Smooth Shading

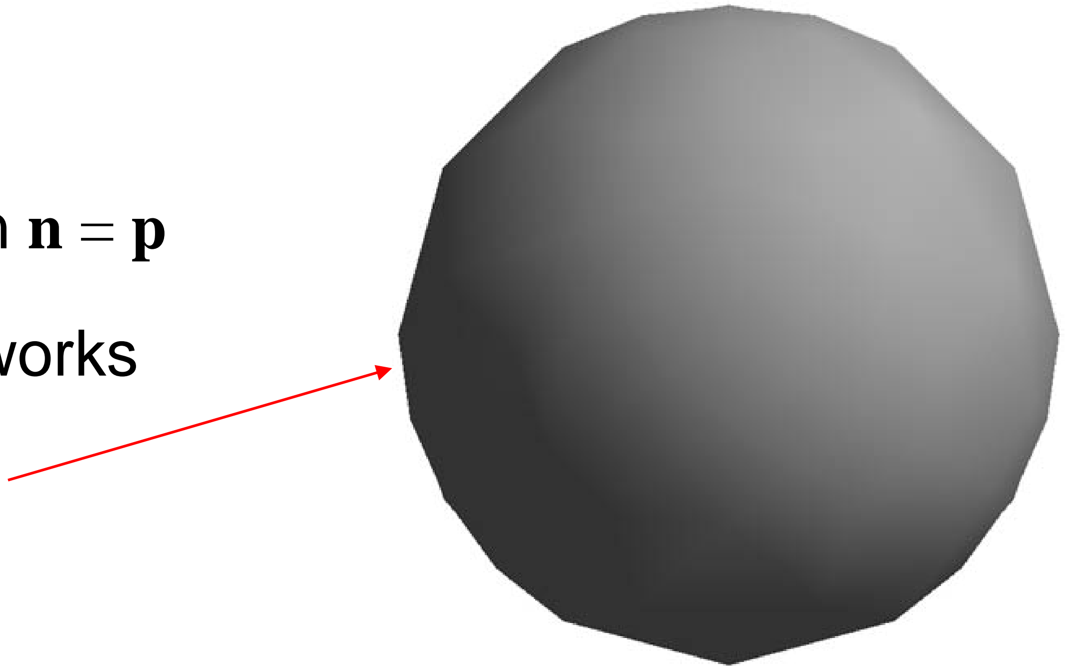
We can set a new normal at each vertex

Easy for sphere model

- If centered at origin $\mathbf{n} = \mathbf{p}$

Now smooth shading works

Note ***silhouette edge***



Gouraud Shading

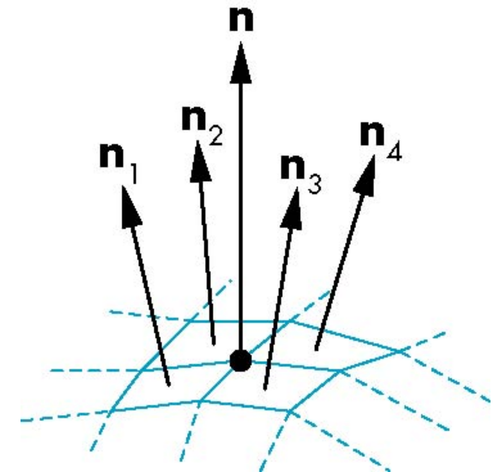
The previous example is not general because we knew the normal at each vertex analytically

For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex

$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4) / |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$

Gouraud Shading

- Find average normal at each vertex
- Apply modified Phong model at each vertex
- Interpolate vertex shades across each polygon



Phong Shading

- Find average vertex normals $\mathbf{n}_A$ and $\mathbf{n}_B$
- Interpolate vertex normals $\mathbf{n}_C$ and $\mathbf{n}_D$

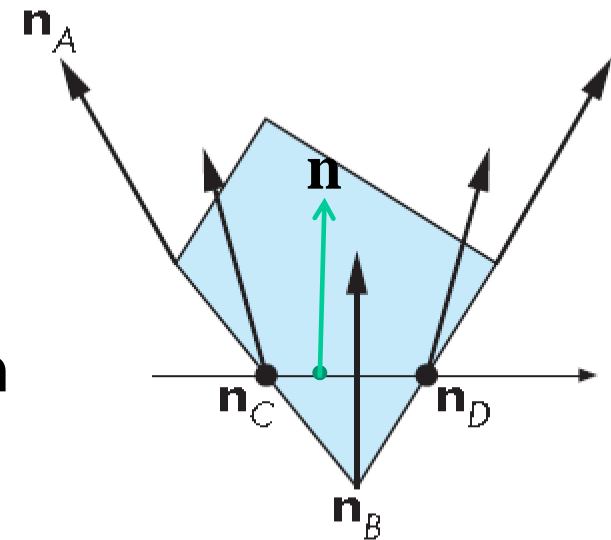
across edges

$$\mathbf{n}_C(\alpha) = (1-\alpha)\mathbf{n}_A + \alpha\mathbf{n}_B$$

- Interpolate edge normals across polygon

$$\mathbf{n}\,(\alpha,\beta) = (1-\beta)\mathbf{n}_C + \beta\mathbf{n}_D$$

- Apply modified Phong model at each fragment



Comparison

If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth 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 we can obtain vertex normals

Examples

Example 1: vertex lighting

- Lighting is handled in vertex shader
- Color is computed for each vertex, then interpolated for each pixel

Example 2: fragment lighting

- Lighting is handled in fragment shader
- Color is computed for each fragment (potential pixel)

Example 1: Vertex Lighting Shaders (Vertex Shader)

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

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct,
SpecularProduct;
uniform vec4 LightPosition;
uniform float Shininess;

uniform mat4 ModelView;
uniform mat4 Projection;
```

Example 1: Vertex Lighting Shaders (Vertex Shader)

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

    // Compute the four vectors
    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

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

Example 1: Vertex Lighting Shaders (Vertex Shader)

```
// Compute terms in the illumination equation
    // Ambient term
    vec4 ambient = AmbientProduct;
    // Diffuse term
    float Kd = max( dot(L, N), 0.0 );
    vec4  diffuse = Kd*DiffuseProduct;
    // Specular term
    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);
    color = ambient + diffuse + specular;
    color.a = 1.0;
    gl_Position = Projection * ModelView * vPosition;
}
```

Example 1: Vertex Lighting Shaders (Fragment Shader)

```
// fragment shader

in vec4 color;

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

Example 2: Fragment Lighting Shaders (Vertex Shader)

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

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

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

Example 2: Fragment Lighting Shaders (Vertex Shader)

```
void main()
{
    vec3 pos = (ModelView*vPosition).xyz;

    fN = (ModelView*vec4(vNormal, 0.0)).xyz ;
    fE = -pos.xyz;
    fL = (ModelView*LightPosition).xyz - pos;


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

Example 2: Fragment Lighting Shaders (Fragment Shader)

```
// fragment shader

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

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

Example 2: Fragment Lighting Shaders (Fragment Shader)

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

   vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);

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

Example 2: Fragment Lighting Shaders (Fragment Shader)

```
    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);

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

Per-vertex Lighting vs Per-fragment Lighting

Per-vertex lighting

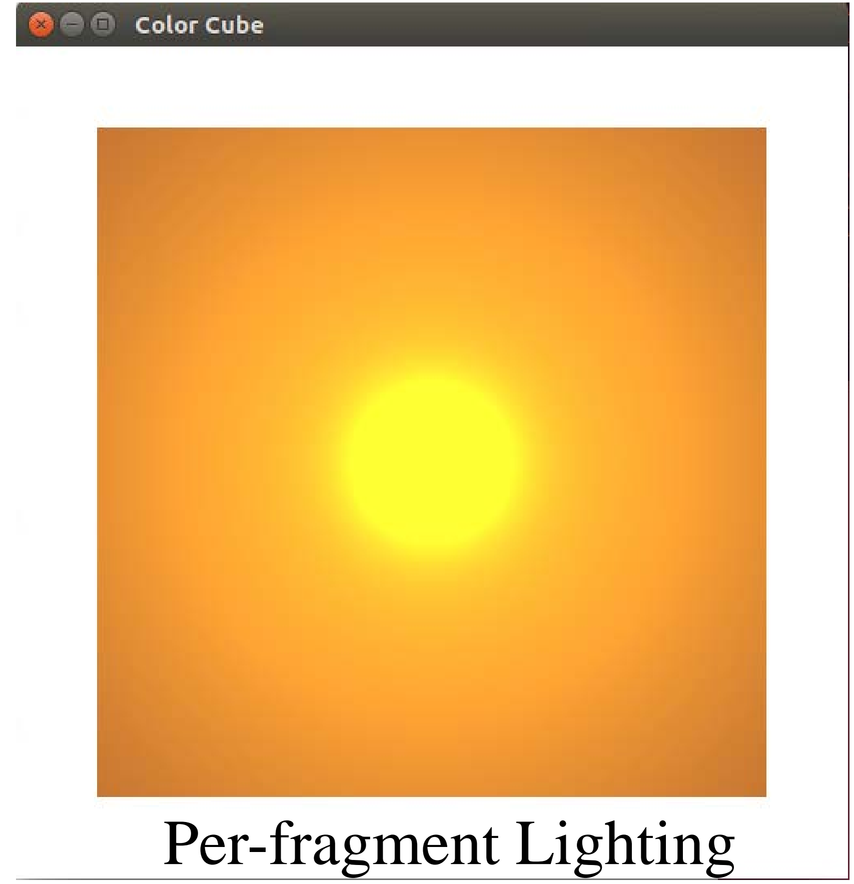
- Lighting is handled in vertex shader
- Color is computed for each vertex, then interpolated for each pixel
- Efficient, but rough

Per-fragment lighting

- Lighting is handled in fragment shader
- Color is computed for each fragment (potential pixel)
- Sophisticated, but slow

Per-vertex Lighting vs Per-fragment Lighting

Per-vertex Lighting



Per-fragment Lighting