Computer Graphics (CS 543)
Lecture 9 (Part 1): Lighting, Shading and Materials (Part 2)

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Modified Phong Model

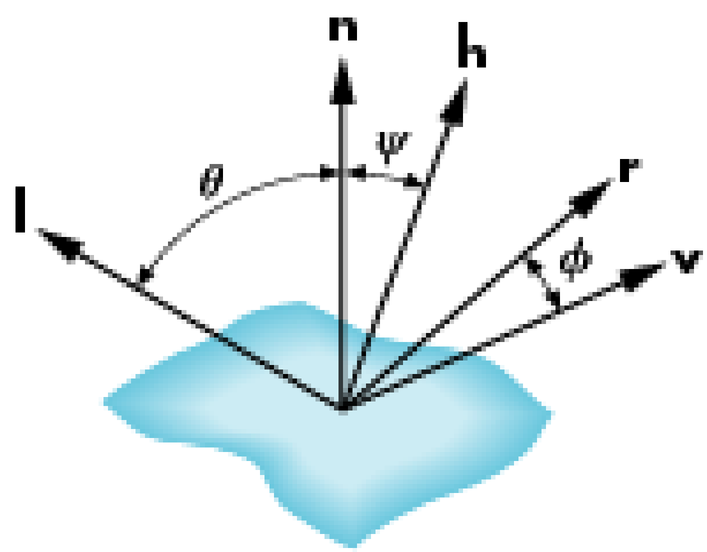
$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$$

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

Used in OpenGL

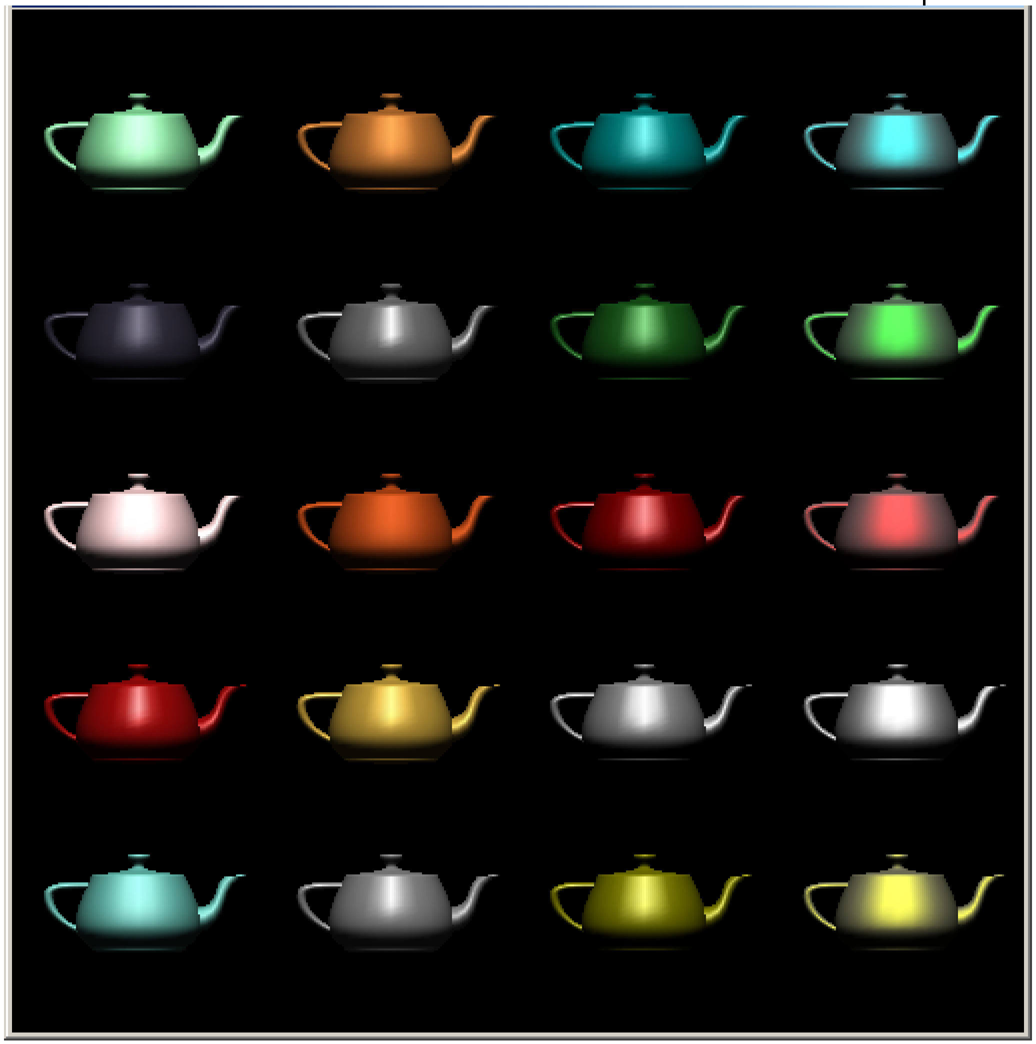
- Blinn proposed using **halfway vector,** more efficient
- **h** is normalized vector halfway between **l** and **v**

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



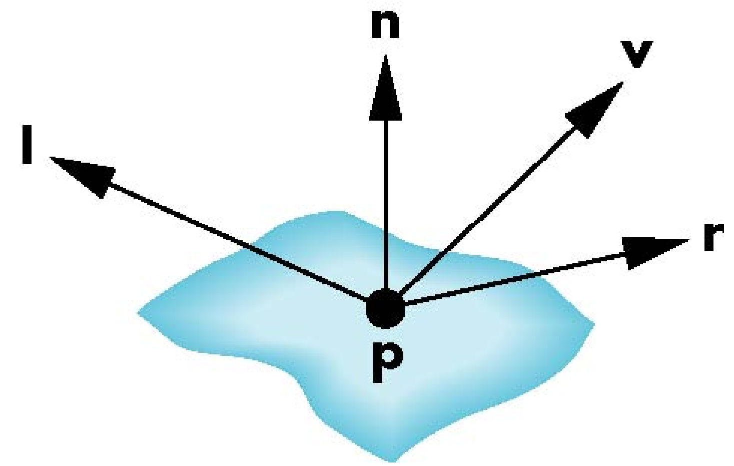
Example

Modified
Phong model gives
Similar results as
original Phong



Computation of Vectors

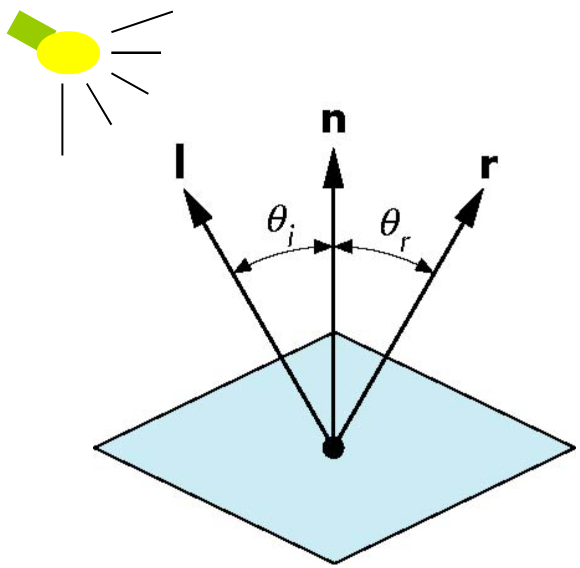
- To calculate lighting at vertex P
  Need **l, n, r** and **v** vectors at vertex P
- User specifies:
  - Light position
  - Viewer (camera) position
  - Vertex (mesh position)
- **l:** Light position – vertex position
- **v:** Viewer position – vertex position
- Normalize all vectors!



Recall: Mirror Direction Vector r

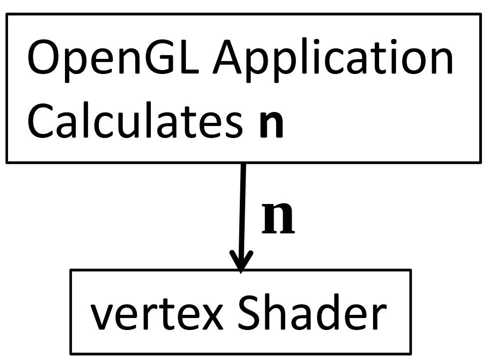
- Can compute **r** from **l** and **n**
- **l**, **n** and **r** are co-planar
- What about determining vertex normal **n**?

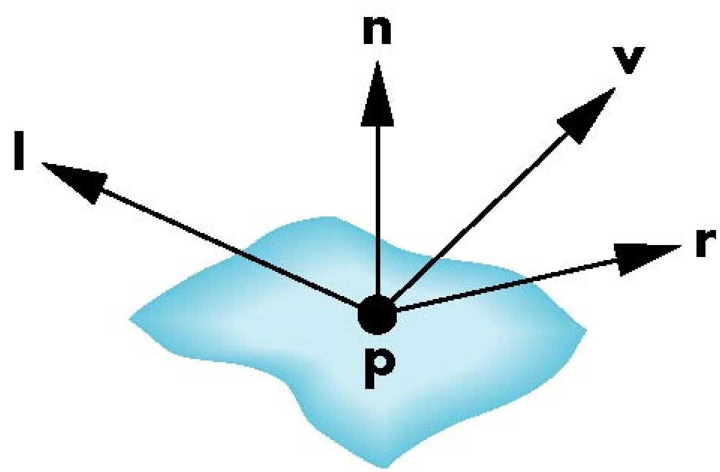
$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n})\,\mathbf{n} - \mathbf{l}$$



Finding Normal, n

- Normal calculation should be done in application
  - OpenGL previously calculated normal for GLU quadrics and Bezier surfaces. Now deprecated
- Passed to vertex shader
- **n** calculation differs depending on surface representation





Recall: Newell Method for Normal Vectors

- Formulae: Normal N = (mx, my, mz)

$$m_x = \sum_{i=0}^{N-1} \left(y_i - y_{next(i)}\right)\left(z_i + z_{next(i)}\right)$$

$$m_y = \sum_{i=0}^{N-1} \left(z_i - z_{next(i)}\right)\left(x_i + x_{next(i)}\right)$$

$$m_z = \sum_{i=0}^{N-1} \left(x_i - x_{next(i)}\right)\left(y_i + y_{next(i)}\right)$$

OpenGL shading

- Need
  - Normals
  - material properties
  - Lights
- State-based shading functions (glNormal, glMaterial, glLight) have been deprecated
- 2 options:
  - Compute lighting in application
  - or send attributes to shaders

Specifying a Point Light Source

- For each light source component, set RGBA and position
- alpha = transparency

Red Green Blue Alpha

```
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);
```

x y z w

Distance and Direction

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

x y z w

- Position is in homogeneous coordinates
  - If w =1.0, we are specifying a finite (x,y,z) location
  - If w =0.0, light at infinity

    (x/w = infinity if w = 0)

Material Properties

- Material properties also has ambient, diffuse, specular
- Material properties specified as RGBA
- Reflectivities
- w component gives opacity
- **Default?** all surfaces are opaque

Red Green Blue Opacity

```
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
```

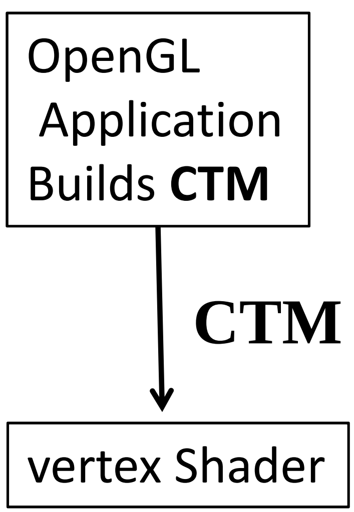
Material
Shininess

Recall: CTM Matrix passed into Shader

- **Recall: CTM** matrix concatenated in application

    mat4 ctm = RotateX(30)*Translate(4,6,8);

- Connected to matrix **ModelView** in shader
- Recall: CTM matrix contains object transform + Camera



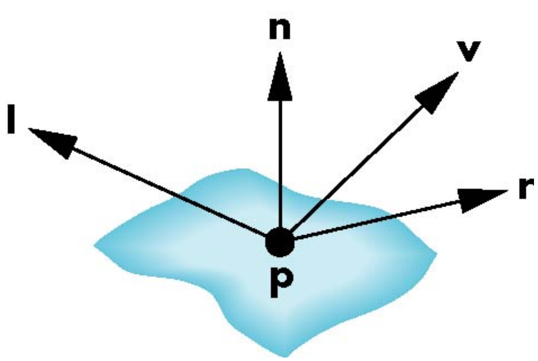
CTM passed in

```
in vec4 vPosition;
Uniform mat4 ModelView ;

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

Computation of Vectors

- CTM transforms vertex position into eye coordinates
  - Eye coordinates? Object, light distances measured from eye
- Normalize all vectors! (magnitude = 1)
- GLSL has a **normalize** function
- **Note:** vector lengths affected by scaling

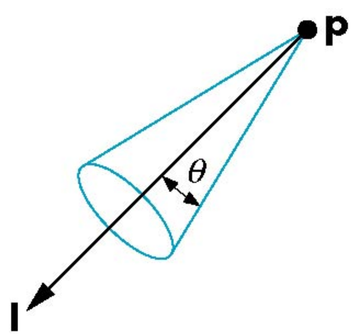


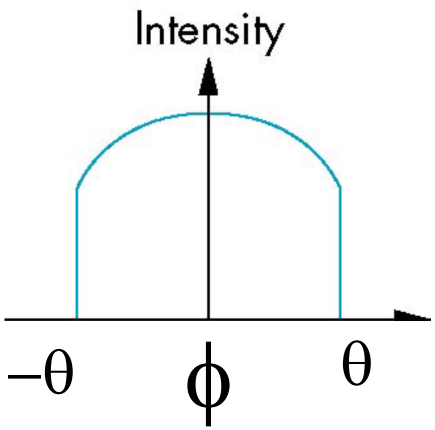

```
// Transform vertex  position into eye coordinates
  vec3 pos = (ModelView * vPosition).xyz;

  vec3 L = normalize( LightPosition.xyz - pos );     // light vector
  vec3 E = normalize( -pos );                        // view vector
  vec3 H = normalize( L + E );                       // Halfway vector
```

Spotlights

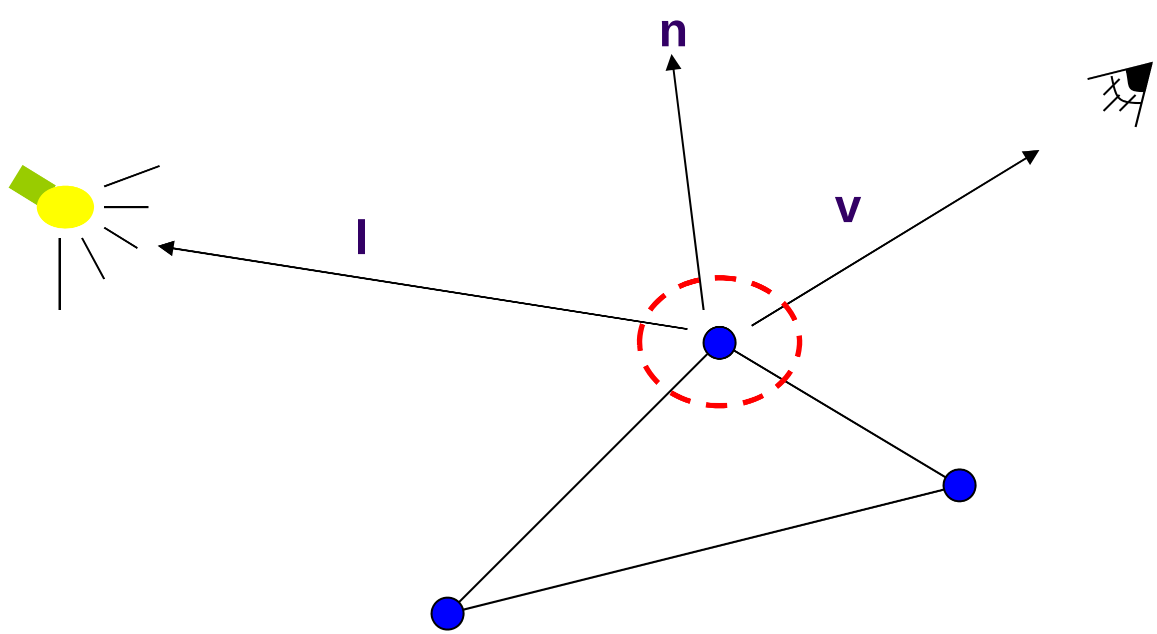
- Derive from point source
  - **Direction I** (of lobe center)
  - **Cutoff:** No light outside θ
  - **Attenuation:** Proportional to $\cos^{\alpha}\phi$





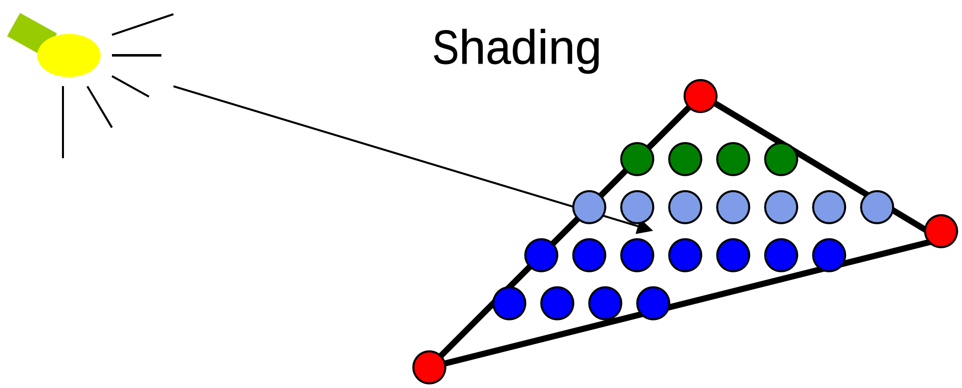
Recall: Lighting Calculated Per Vertex

- Phong model (ambient+diffuse+specular) calculated at each vertex to determine vertex color
- Per vertex calculation? Usually done in vertex shader



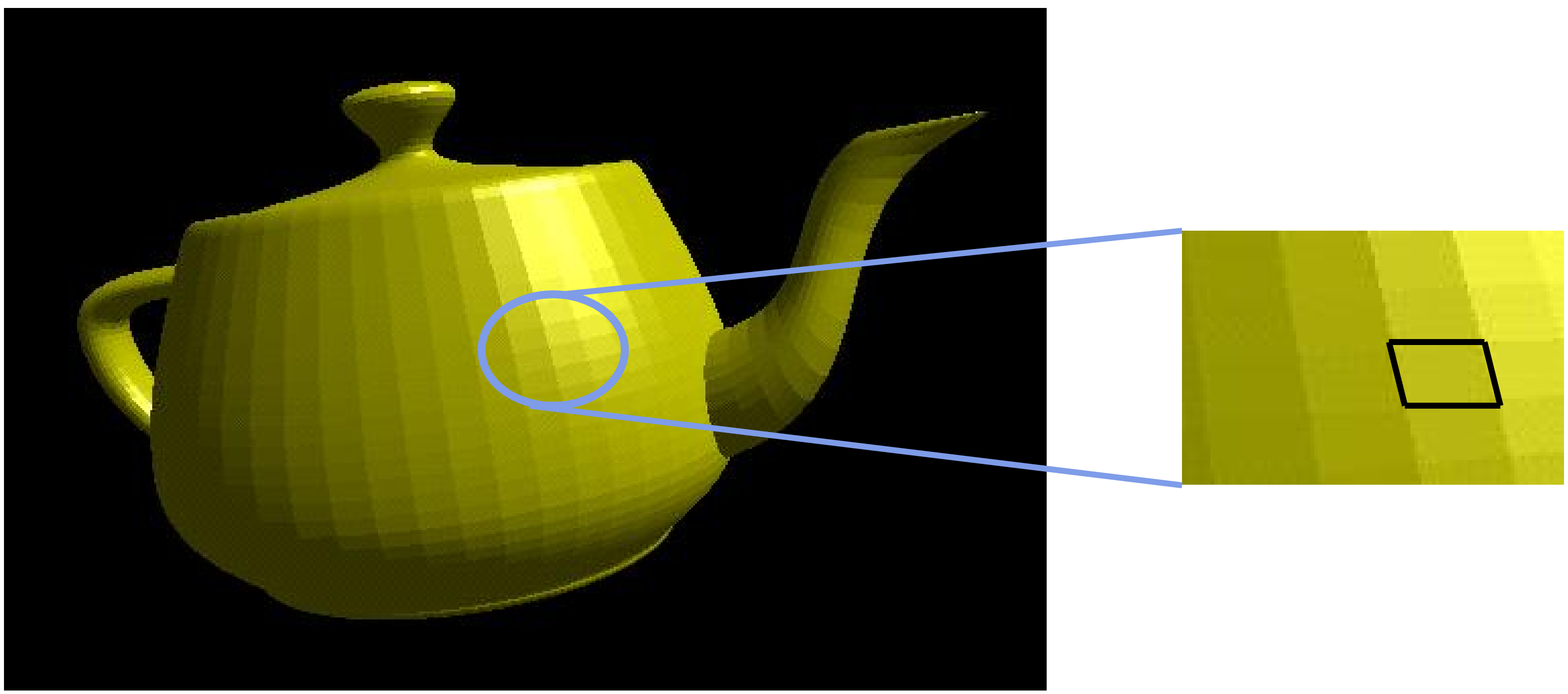
Shading?

- After triangle is rasterized/drawn
  - Per-vertex lighting calculation means we know color of pixels coinciding with vertices (red dots)
- Shading determines color of interior surface pixels
- Two shading
  - Assume linear change => interpolate (Smooth shading)
  - No interpolation (Flat shading)



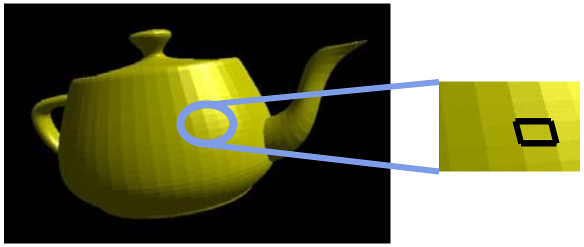
Flat Shading

- compute lighting once for each face, assign color to whole face



Flat shading

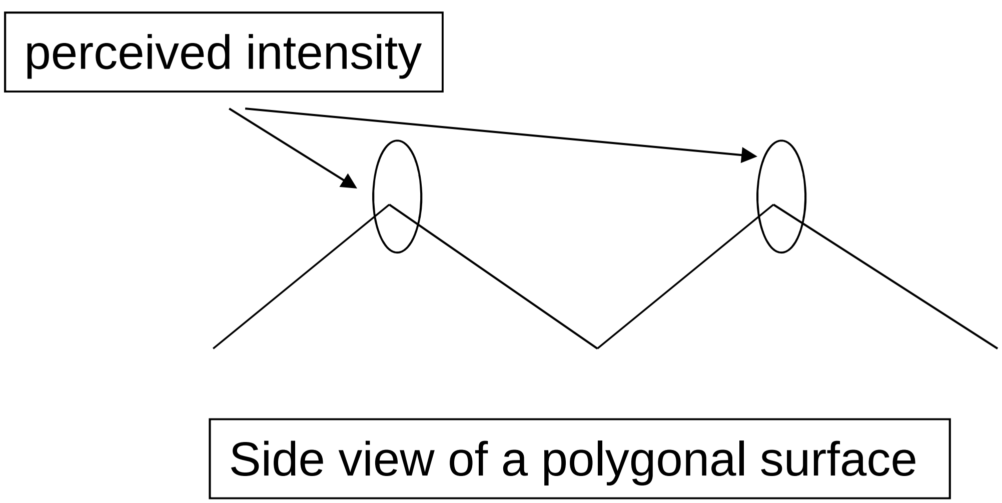
- Only use face normal for all vertices in face and material property to compute color for face
- Benefit: Fast!
- Used when:
  - Polygon is small enough
  - Light source is far away (why?)
  - Eye is very far away (why?)

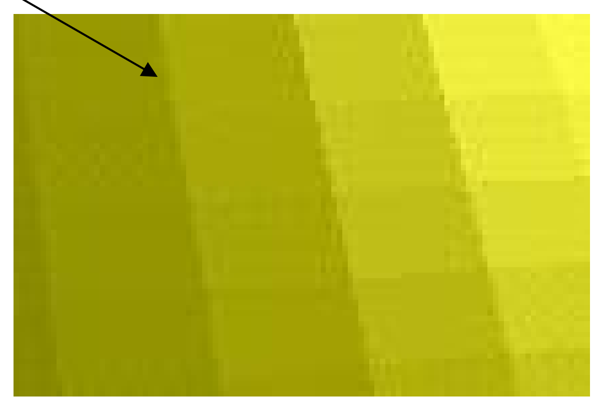


- Previous OpenGL command: glShadeModel(GL_FLAT) **deprecated!**

Mach Band Effect

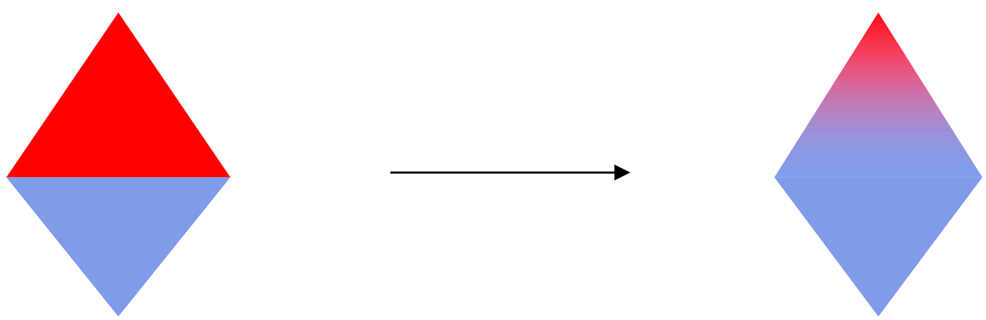
- Flat shading suffers from "mach band effect"
- Mach band effect – human eyes accentuate the discontinuity at the boundary





Smooth shading

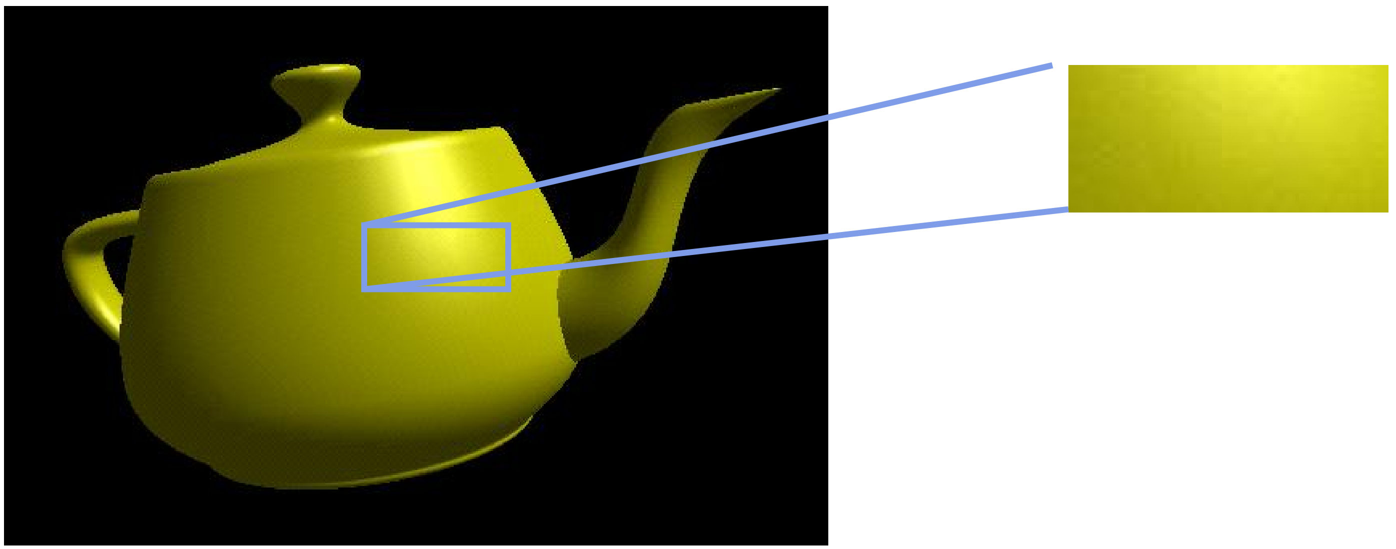
- Fix mach band effect – remove edge discontinuity
- Compute lighting for more points on each face
- 2 popular methods:
  - Gouraud shading
  - Phong shading



Flat shading **Smooth shading**

Gouraud Shading

- Lighting calculated for each polygon vertex
- Colors are interpolated for interior pixels
- Interpolation? Assume linear change from one vertex color to another
- Gouraud shading (interpolation) is OpenGL default



Per-Vertex Lighting Shaders I

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

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

Ambient, diffuse, specular (light * reflectivity) specified by user

Per-Vertex Lighting Shaders II

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

    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;
```

Per-Vertex Lighting Shaders III

```
// Compute terms in the illumination equation
    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;
    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;
    color.a = 1.0;
}
```

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

Per-Vertex Lighting Shaders IV

```
// fragment shader

in vec4 color;

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

Flat Shading Implementation

- Default is **smooth shading**
- Colors set in vertex shader interpolated
- **Flat shading?** Stop color interpolation
- In vertex shader, add keyword **flat** to output **color**

```
flat out vec4 color;  //vertex shade
 ……
   color = ambient + diffuse + specular;
   color.a = 1.0;
```

Flat Shading Implementation

- Also, in fragment shader, add keyword **flat** to color received from vertex shader

```
flat in vec4 color;

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

References

- Interactive Computer Graphics (6th edition), Angel and Shreiner
- Computer Graphics using OpenGL (3rd edition), Hill and Kelley