

CS 4731/543: Computer Graphics
Lecture 5 (Part III): Shading

Emmanuel Agu

Recall: Setting Light Property



- Define colors and position a light

```
GLfloat light_ambient[] = {0.0, 0.0, 0.0, 1.0};  
GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat light_position[] = {0.0, 0.0, 1.0, 1.0};
```

← colors

← Position

```
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);  
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);  
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);  
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

What if I set
Position to
(0,0,1,0)?

Recall: Setting Material Example



- Define ambient/diffuse/specular reflection and shininess

```
GLfloat mat_amb_diff[] = {1.0, 0.5, 0.8, 1.0};  
GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};  
GLfloat shininess[] = {5.0};
```

← refl. coeff. (range: dull 0 – very shiny 128)

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,  
             mat_amb_diff);
```

```
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

```
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);
```

Recall: Calculating Color at Vertices

- Illumination from a light:

$$\begin{aligned} \mathbf{Illum} &= \mathbf{ambient} + \mathbf{diffuse} + \mathbf{specular} \\ &= \mathbf{K_a} \times \mathbf{I} + \mathbf{K_d} \times \mathbf{I} \times (\cos \theta) + \mathbf{K_s} \times \mathbf{I} \times \cos^n(\phi) \end{aligned}$$

- If there are N lights

$$\mathbf{Total\ illumination\ for\ a\ point\ P} = \mathbf{S}(\mathbf{Illum})$$

- Sometimes light or surfaces are colored
- Treat R,G and B components separately
- i.e. can specify different RGB values for either light or material

To:

$$\mathbf{Illum_r} = \mathbf{K_{ar}} \times \mathbf{I_r} + \mathbf{K_{dr}} \times \mathbf{I_r} \times (\cos \theta) + \mathbf{K_{sr}} \times \mathbf{I_r} \times \cos^n(\phi)$$

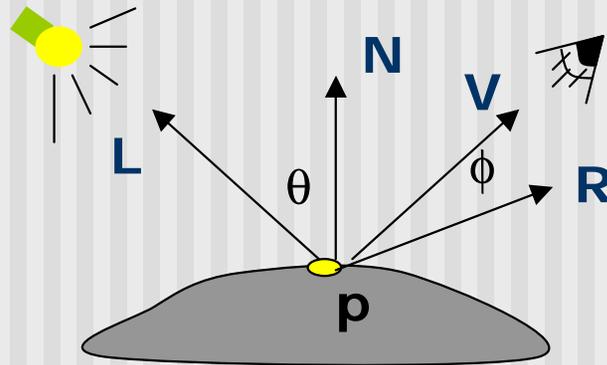
$$\mathbf{Illum_g} = \mathbf{K_{ag}} \times \mathbf{I_g} + \mathbf{K_{dg}} \times \mathbf{I_g} \times (\cos \theta) + \mathbf{K_{sg}} \times \mathbf{I_g} \times \cos^n(\phi)$$

$$\mathbf{Illum_b} = \mathbf{K_{ab}} \times \mathbf{I_b} + \mathbf{K_{db}} \times \mathbf{I_b} \times (\cos \theta) + \mathbf{K_{sb}} \times \mathbf{I_b} \times \cos^n(\phi)$$

Recall: Calculating Color at Vertices

$$\begin{aligned} \mathbf{Illum} &= \text{ambient} + \text{diffuse} + \text{specular} \\ &= K_a \times \mathbf{I} + K_d \times \mathbf{I} \times (\cos \theta) + K_s \times \mathbf{I} \times \cos(\phi) \end{aligned}$$

- $(\cos \theta)$ and $\cos(\phi)$ are calculated as dot products of Light vector \mathbf{L} , Normal \mathbf{N} , and Mirror direction vector \mathbf{R}



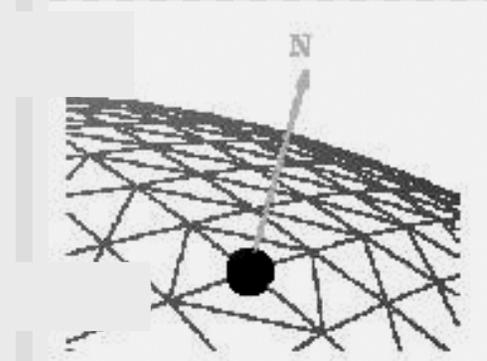
- To give
$$\mathbf{Illum} = K_a \times \mathbf{I} + K_d \times \mathbf{I} \times (\mathbf{N} \cdot \mathbf{L}) + K_s \times \mathbf{I} \times (\mathbf{R} \cdot \mathbf{V})^n$$

Surface Normals



- Correct normals are essential for correct lighting
- Associate a normal to each vertex

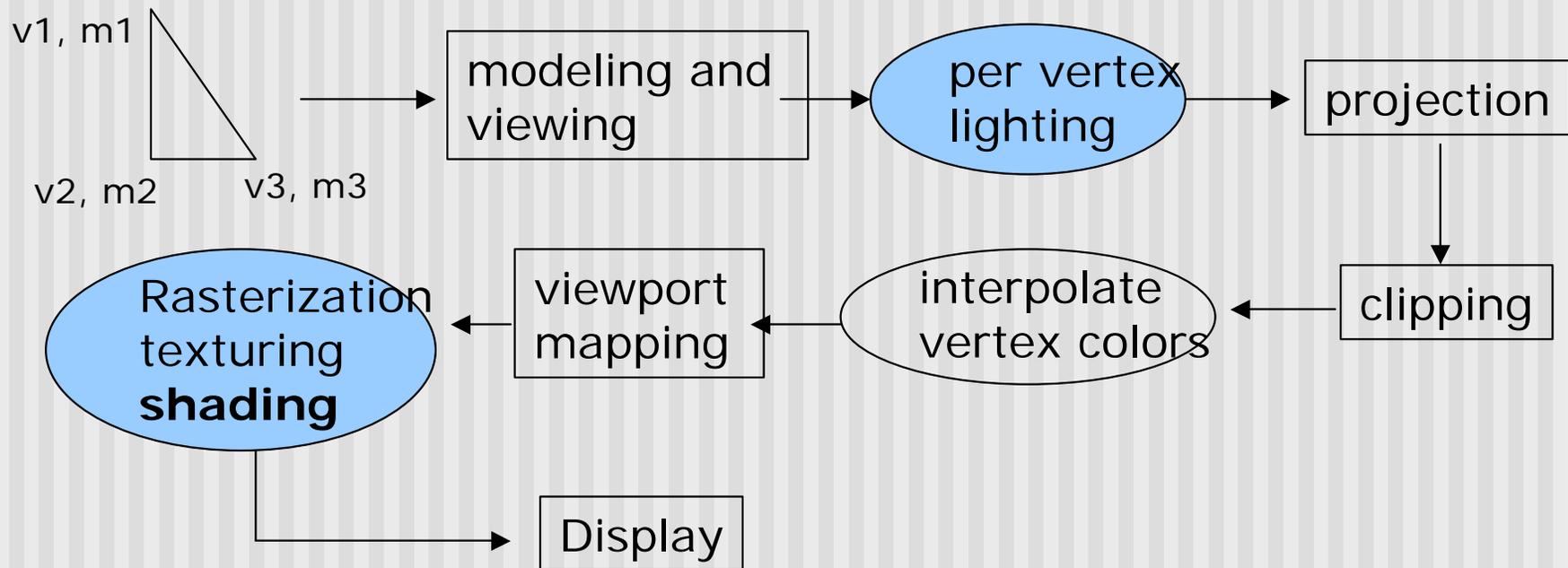
```
glBegin(...)  
    glNormal3f(x,y,z)  
    glVertex3f(x,y,z)  
    ...  
glEnd()
```



- The normals you provide need to have a unit length
 - You can use **glEnable(GL_NORMALIZE)** to have OpenGL normalize all the normals

Lighting revisit

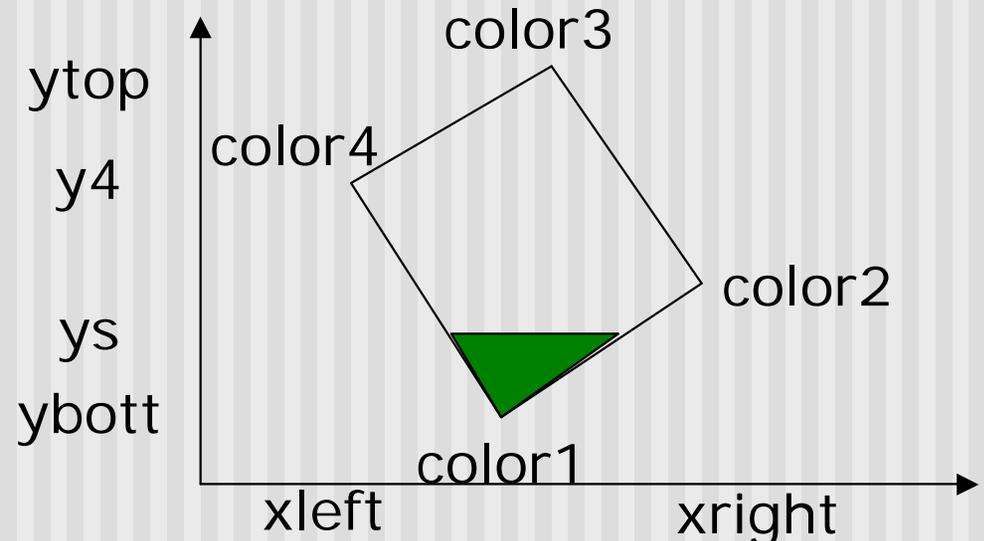
- Light calculation so far is at vertices
- Pixel may not fall right on vertex
- Shading: calculates color to set interior pixel to
- Where are lighting/shading performed in the pipeline?



Example Shading Function (Pg. 432 of Hill)

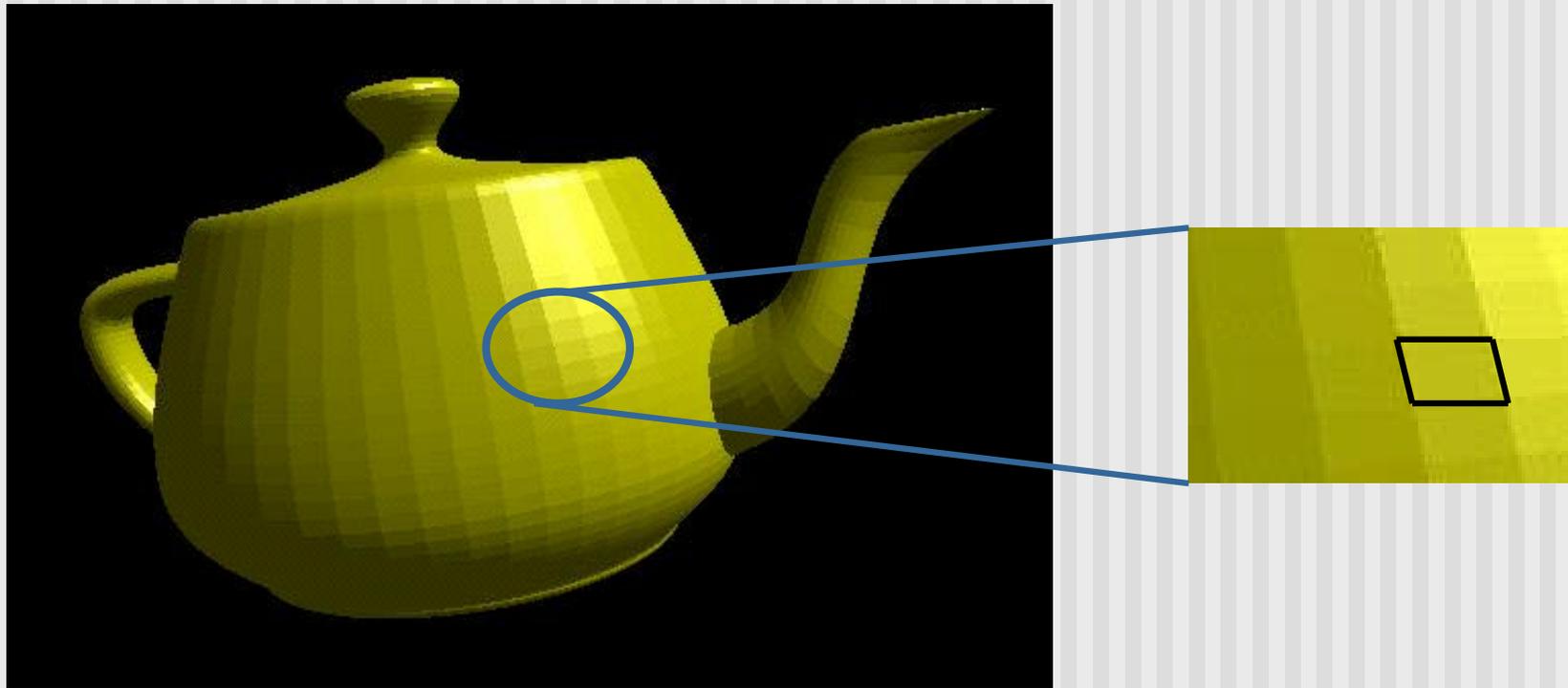
```
for(int y = Ybott; Y < Ytop; Y++)  
{  
    find xleft and xright  
    for(int x = xleft; x < xright; x++)  
    {  
        find the color c for this pixel  
        put c into the pixel at (x, y)  
    }  
}
```

- Scans pixels, row by row, calculating color for each pixel



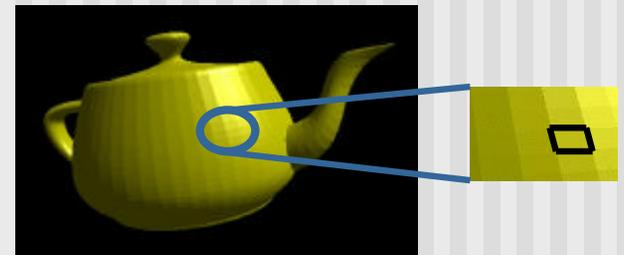
Polygon shading model

- Flat shading - compute lighting once and assign the color to the whole (mesh) polygon



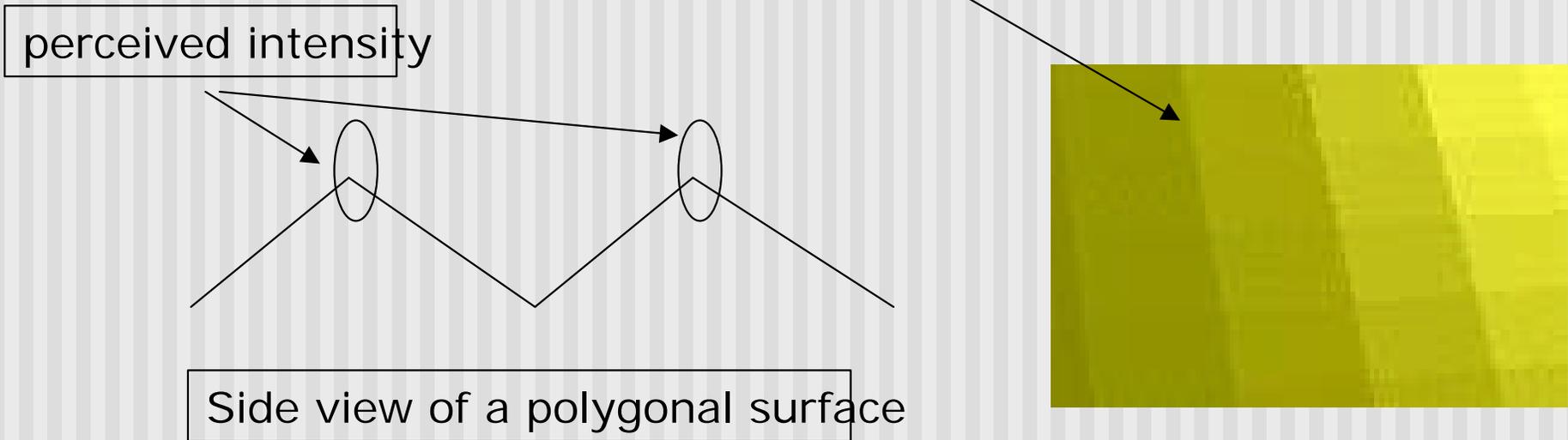
Flat shading

- Only use one vertex normal and material property to compute the color for the polygon
- Benefit: fast to compute
- Used when:
 - Polygon is small enough
 - Light source is far away (why?)
 - Eye is very far away (why?)
- OpenGL command: `glShadeModel(GL_FLAT)`



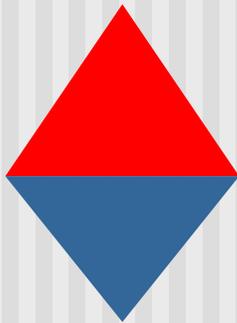
Mach Band Effect

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

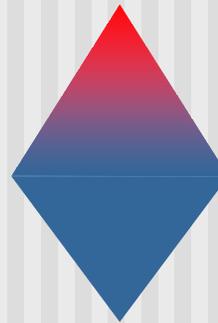


Smooth shading

- Fix the mach band effect – remove edge discontinuity
- Compute lighting for more points on each face



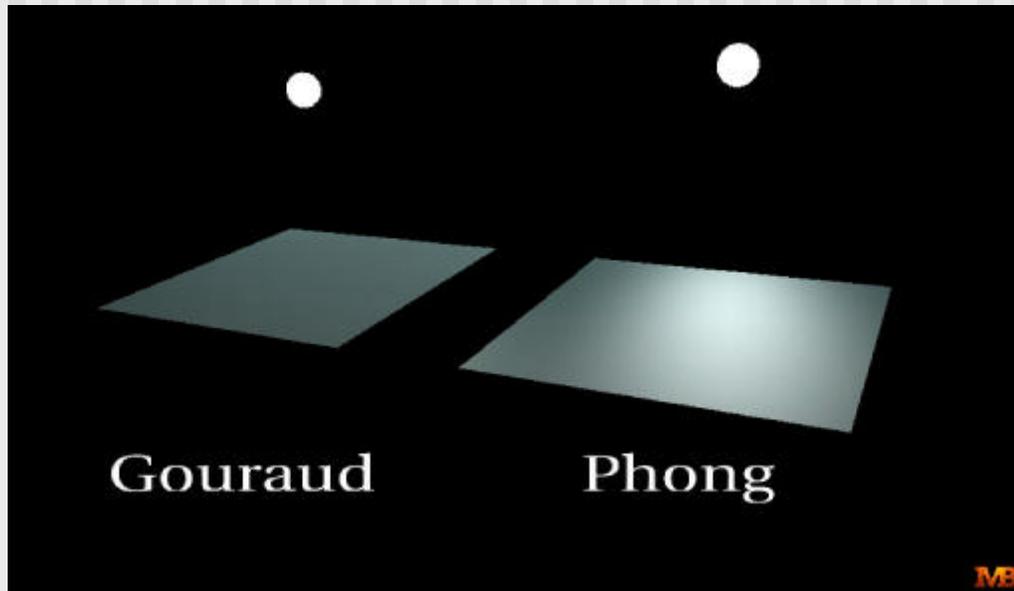
Flat shading



Smooth shading

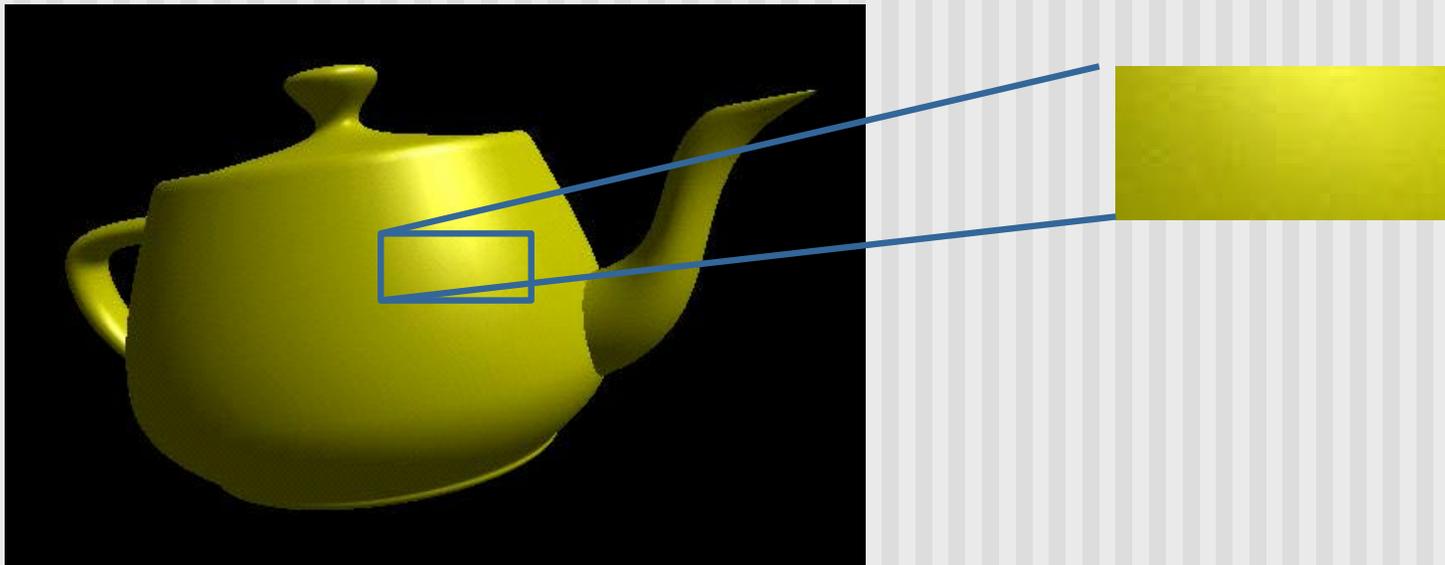
Smooth shading

- Two popular methods:
 - Gouraud shading (used by OpenGL)
 - Phong shading (better specular highlight, not in OpenGL)



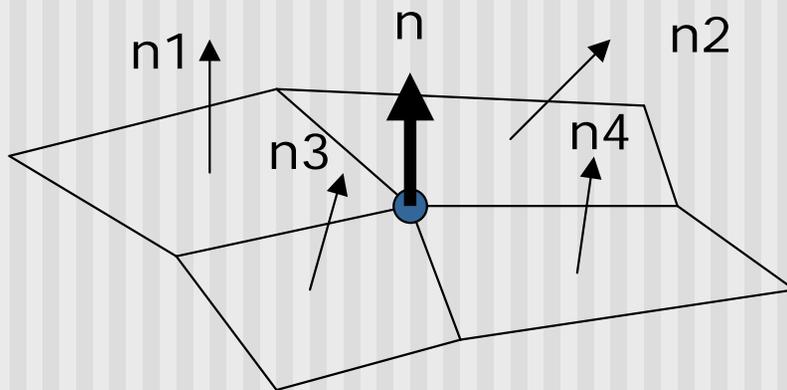
Gouraud Shading

- The smooth shading algorithm used in OpenGL
`glShadeModel(GL_SMOOTH)`
- Lighting is calculated for each of the polygon vertices
- Colors are interpolated for interior pixels



Gouraud Shading

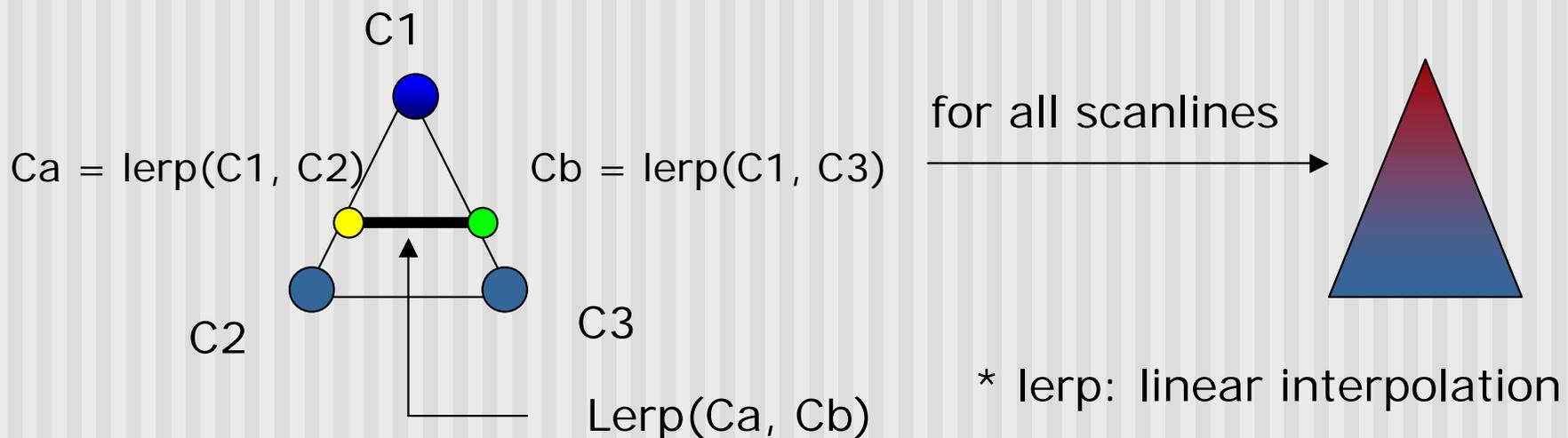
- Per-vertex lighting calculation
- Normal is needed for each vertex
- Per-vertex normal can be computed by averaging the adjacent face normals



$$n = (n1 + n2 + n3 + n4) / 4.0$$

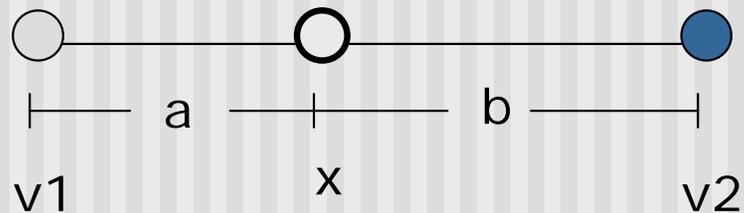
Gouraud Shading

- Compute vertex illumination (color) before the projection transformation
- Shade interior pixels: color interpolation (normals are not needed)



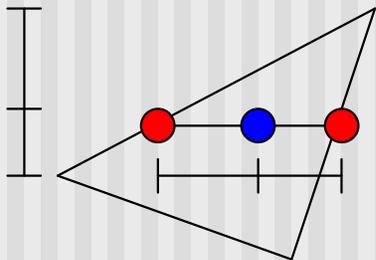
Gouraud Shading

- Linear interpolation



$$x = b / (a+b) * v1 + a / (a+b) * v2$$

- Interpolate triangle color: use y distance to interpolate the two end points in the scanline, and use x distance to interpolate interior pixel colors

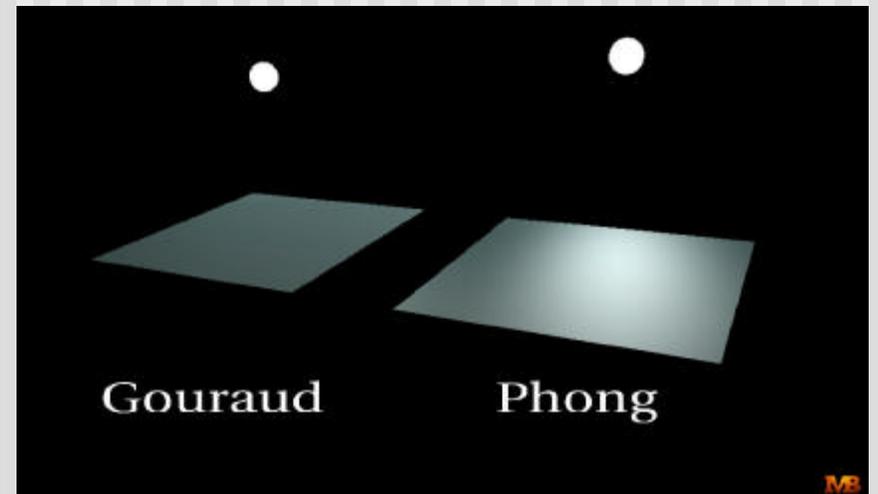
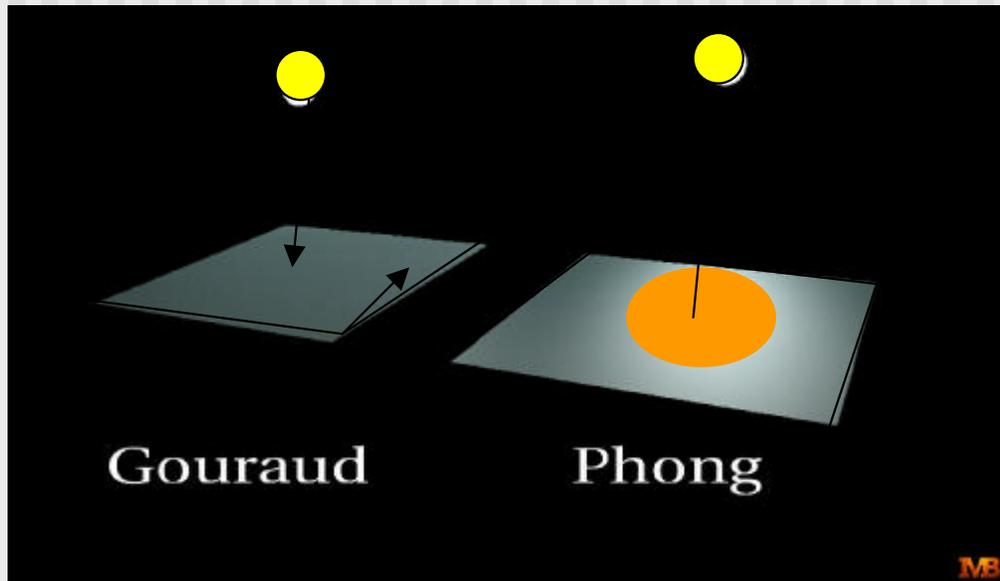


Gouraud Shading Function (Pg. 433 of Hill)

```
for(int y = Ybott; y < Ytop; y++) // for each scan line
{
    find xleft    and xright
    find colorleft    and colorright
    colorinc = (colorright - colorleft) / (xright - xleft)
    for(int x = xleft, c = colorleft; x < xright;
        x++, c+ = colorinc)
    {
        put c into the pixel at (x, y)
    }
}
```

Gouraud Shading Problem

- Lighting in the polygon interior can be inaccurate

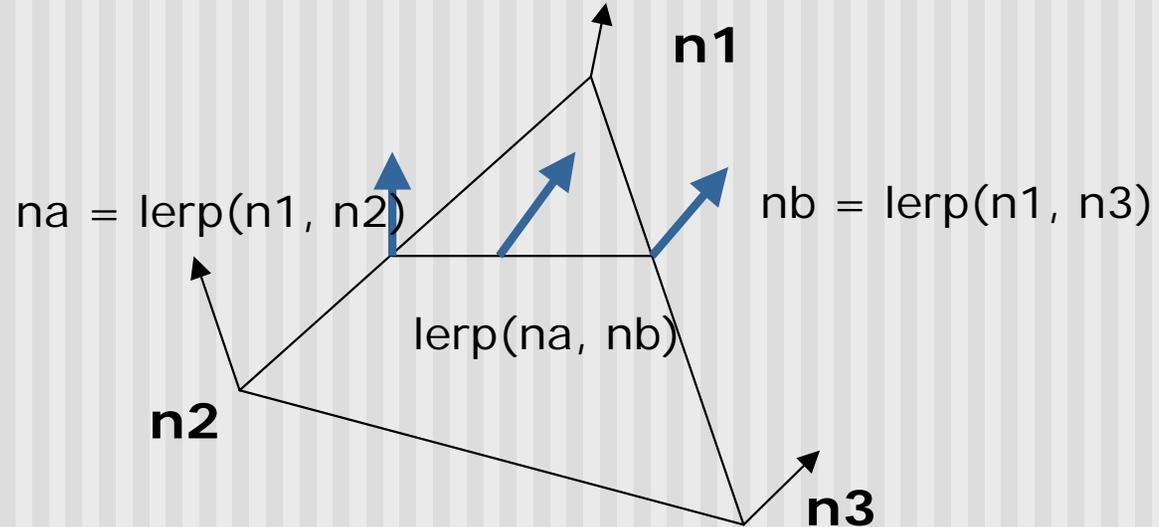


Phong Shading

- Instead of interpolation, we calculate lighting for each pixel inside the polygon (per pixel lighting)
- Need normals for all the pixels – not provided by user
- Phong shading algorithm interpolates the normals and compute lighting during rasterization (need to map the normal back to world or eye space though)

Phong Shading

- Normal interpolation



- Slow – not supported by OpenGL and most graphics hardware

References

- Hill, chapter 8