Computer Graphics (CS 543)
Lecture 8a: Per-Vertex lighting, Shading and Per-Fragment lighting

Prof Emmanuel Agu

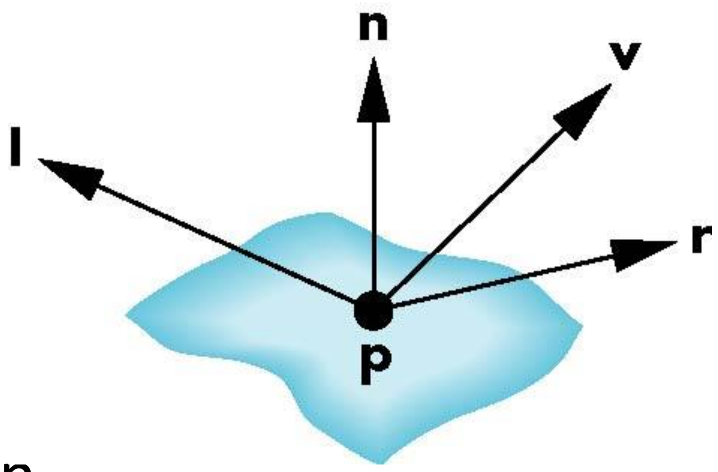
Computer Science Dept.

Worcester Polytechnic Institute (WPI)

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
- **n:** Newell method
- Normalize all vectors!



Specifying a Point Light Source

- For each light source component, set RGBA
- 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

- Light position is in homogeneous coordinates

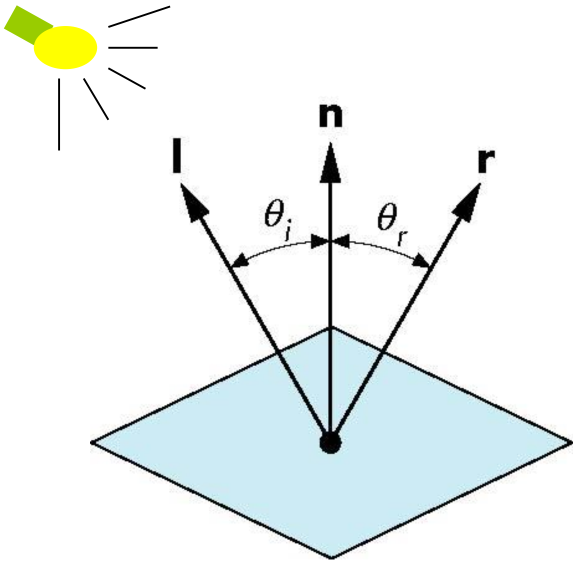
```
vec4 light0_pos =vec4(1.0, 2.0, 3,0, 1.0);
```

x y z w

Recall: Mirror Direction Vector r

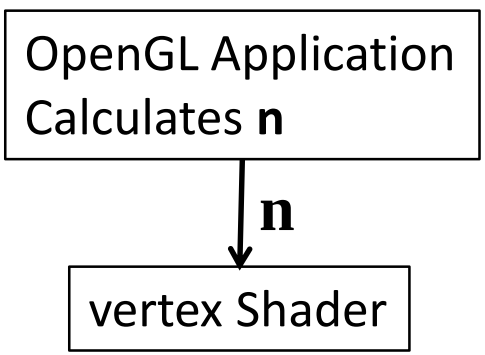
- Can compute $\mathbf{r}$ from $\mathbf{l}$ and $\mathbf{n}$
- $\mathbf{l}$, $\mathbf{n}$ and $\mathbf{r}$ are co-planar

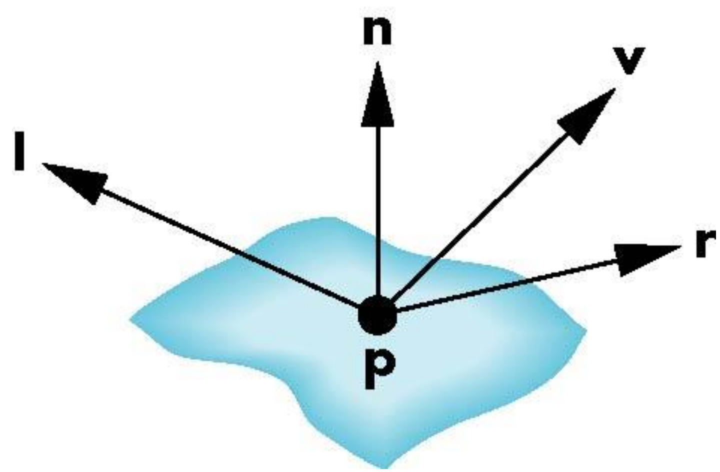
$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n}\,)\,\mathbf{n} - \mathbf{l}$$



Finding Normal, n

- Normal calculation in application. E.g. Newell method
- Passed to vertex shader





Material Properties

- OpenGL Normal, material, shading functions **deprecated**
  - (glNormal, glMaterial, glLight) **deprecated**
- Specify material properties of scene object ambient, diffuse, specular (RGBA)
- w component gives opacity (transparency)
- **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 (alpha in specular)

Recall: CTM Matrix passed into Shader

- Recall: CTM matrix concatenated in application

```
mat4 ctm = ctm  * LookAt(vec4 eye, vec4 at, vec4 up);
```

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

OpenGL Application Builds **CTM**

CTM

vertex Shader

```
in vec4 vPosition;
Uniform mat4 ModelView ;        CTM passed in

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

Per-Vertex Lighting: Declare Variables

Note: Phong lighting calculated at EACH VERTEX!!

```
// 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;
```

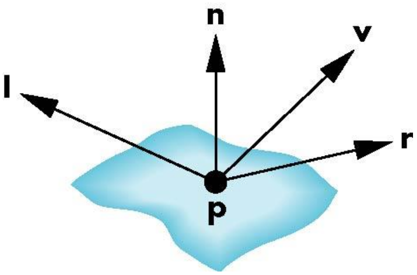
Ambient, diffuse, specular (light * reflectivity) specified by user

$k_a\ I_a$ $\quad$ $k_d\ I_d$ $\quad$ $k_s\ I_s$

exponent of specular term

Per-Vertex Lighting: Compute Vectors

- CTM transforms vertex position into eye coordinates
  - Eye coordinates? Object, light distances measured from eye




```
void main( )
{
    // 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

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

GLSL normalize function

Why not 1.0?

Per-Vertex Lighting: Calculate Components

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

    float cos_theta = max( dot(L, N), 0.0 );
    vec4  diffuse = cos_theta * DiffuseProduct;
    float cos_phi = pow( max(dot(N, H), 0.0), Shininess );
    vec4  specular = cos_phi * 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;
}
```

$\mathbf{k_a\, I_a}$

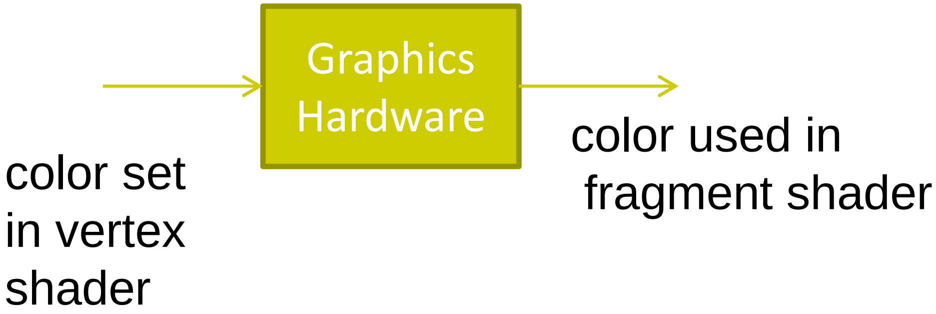
$\mathbf{k_d\, I_d\ \ l \cdot n}$

$k_s\, I_s\, (\mathbf{n} \cdot \mathbf{h})^{\beta}$

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

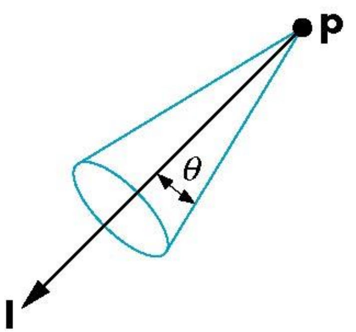
Per-Vertex Lighting Shaders IV

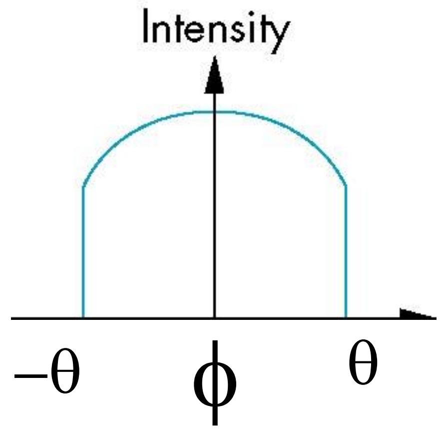
```
// in vertex shader, we declared color as out, set it
    …….
    color = ambient + diffuse + specular;
    color.a = 1.0;
}

// in fragment shader (
in vec4 color;

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



Spotlights

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



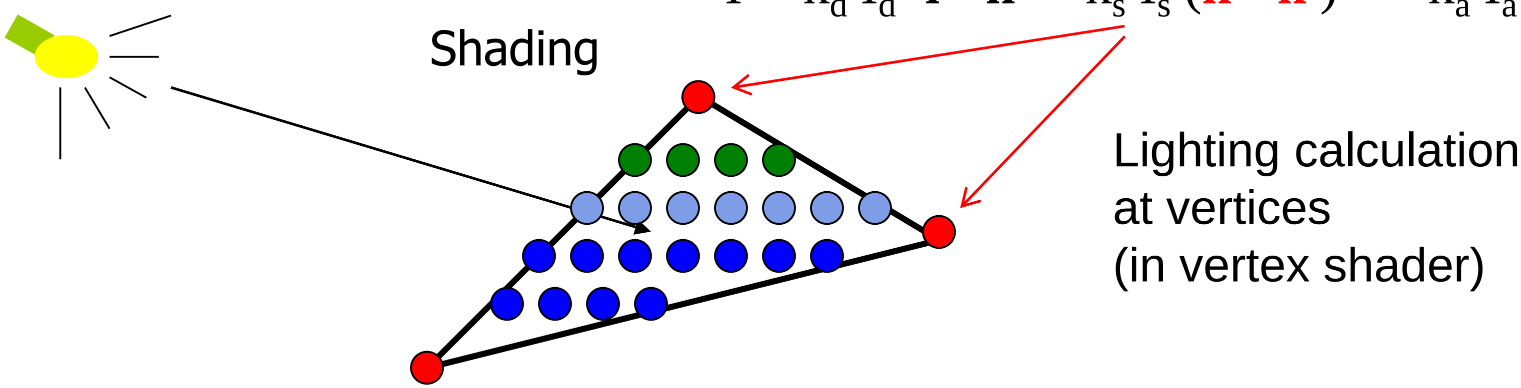


Shading

Shading?

- After triangle is rasterized/drawn
  - Per-vertex lighting calculation means we know color of pixels at vertices (red dots)
- Shading determines color of interior surface pixels

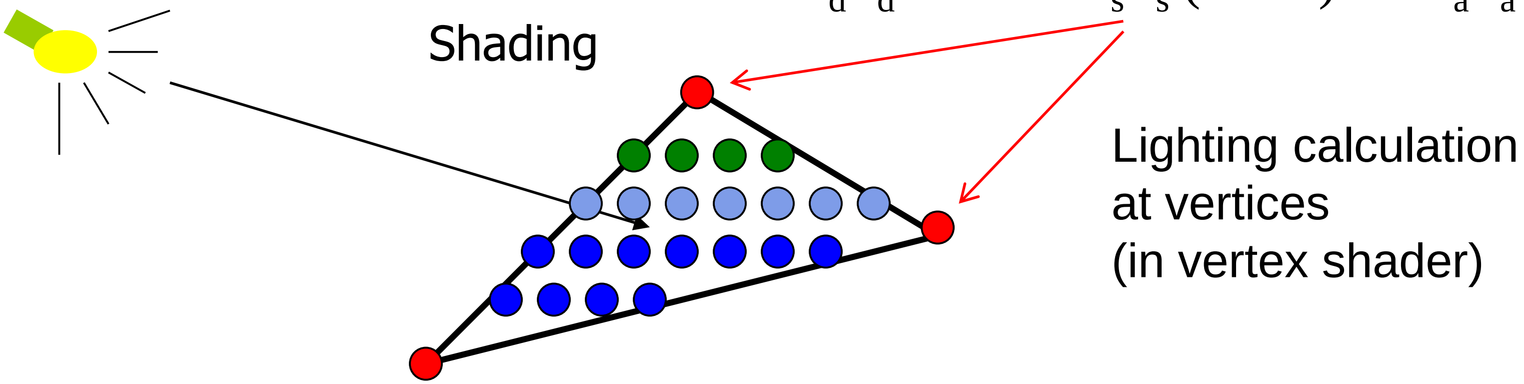
$$I = k_d I_d \; \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$



Shading?

- Two types of shading
  - Assume linear change => interpolate (Smooth shading)
  - No interpolation (Flat shading)

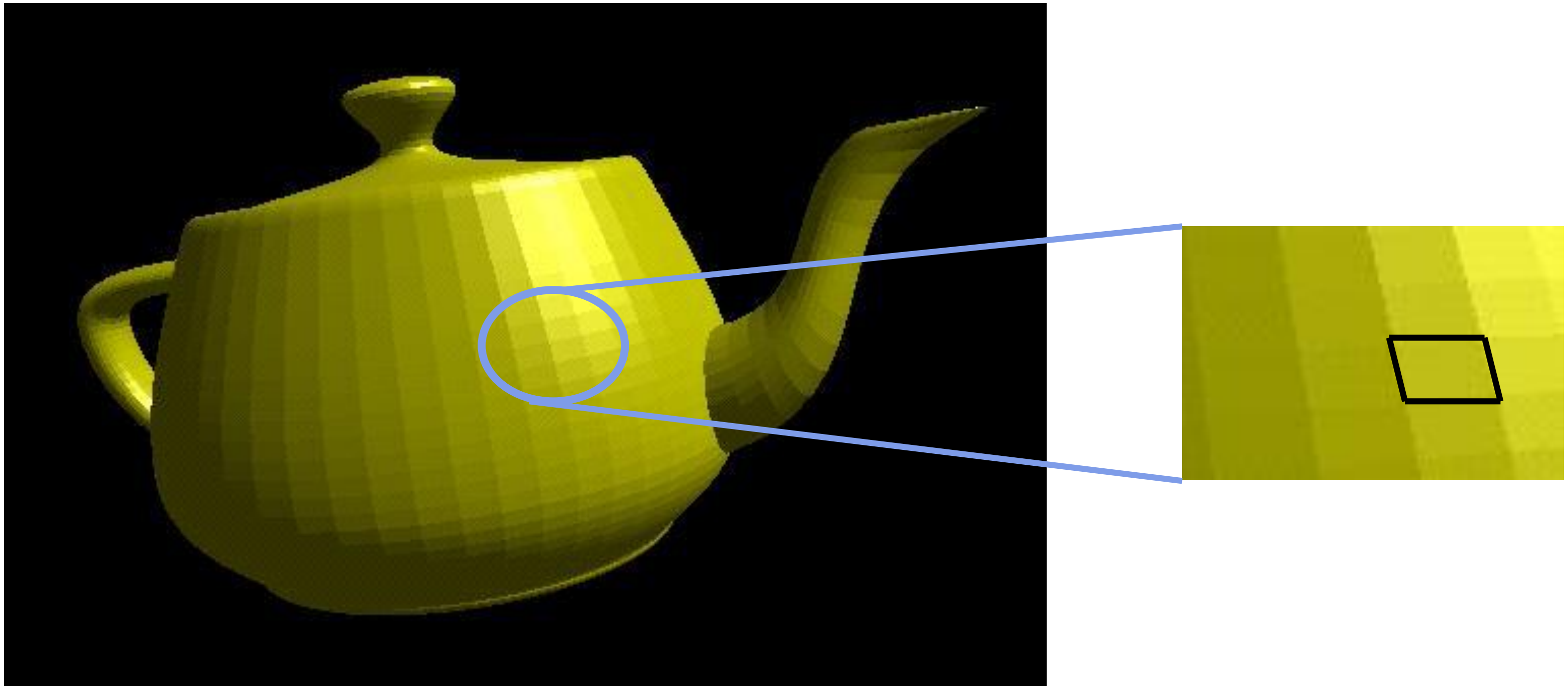
$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$



- Hardware unit between vertex and fragment units does shading

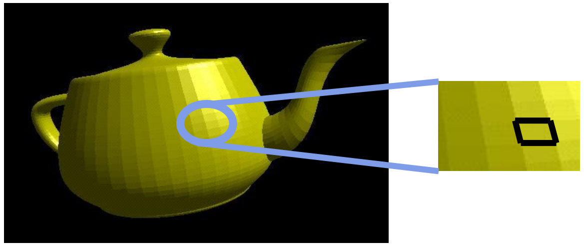
Flat Shading

- compute lighting once for each face, assign color to whole face
- Benefit: Fast!!



Flat shading

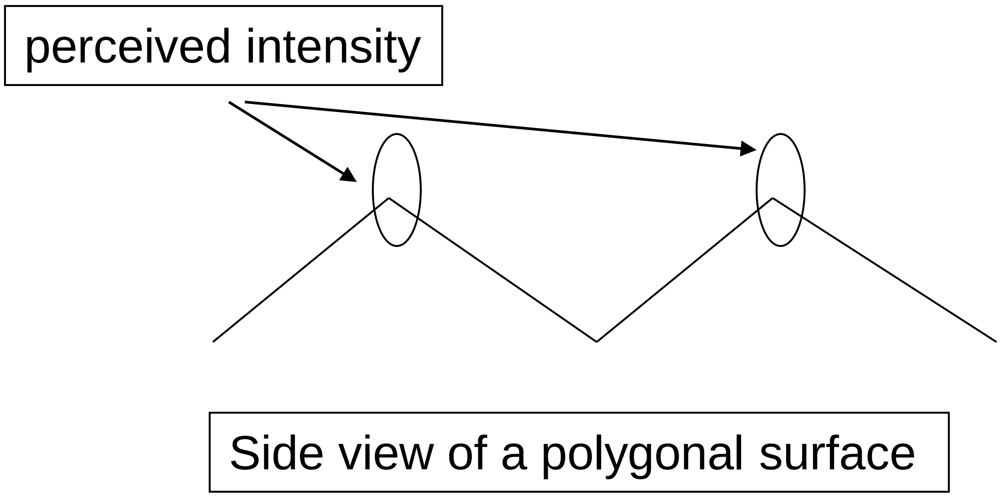
- Used when:
  - Polygon is small enough
  - Light source is far away (why?)
  - Eye is very far away (why?)

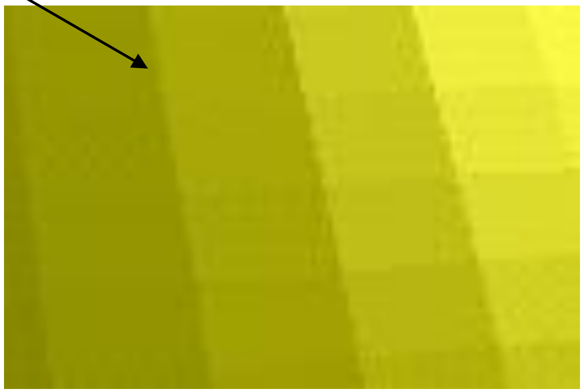


- Previous OpenGL command: glShadeModel(GL_FLAT) **deprecated!**

Mach Band Effect

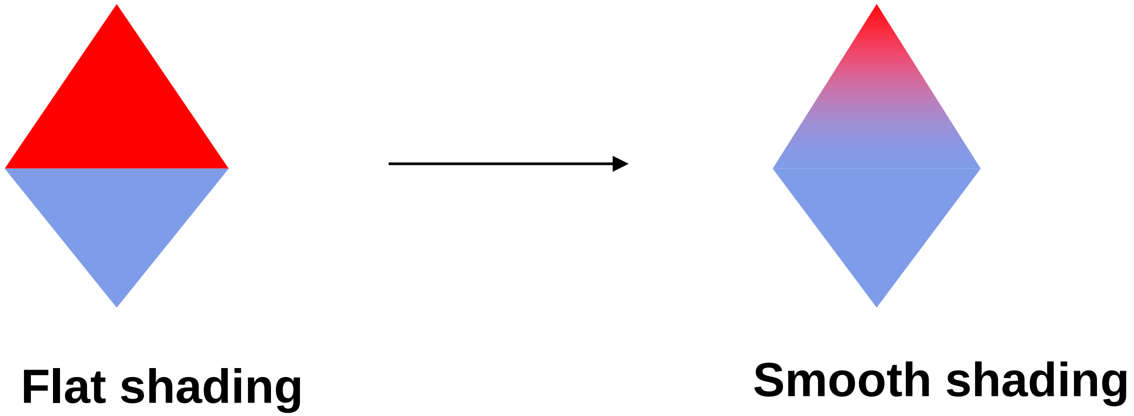
- Flat shading suffers from “mach band effect”
- Mach band effect – human eyes amplify discontinuity at the boundary





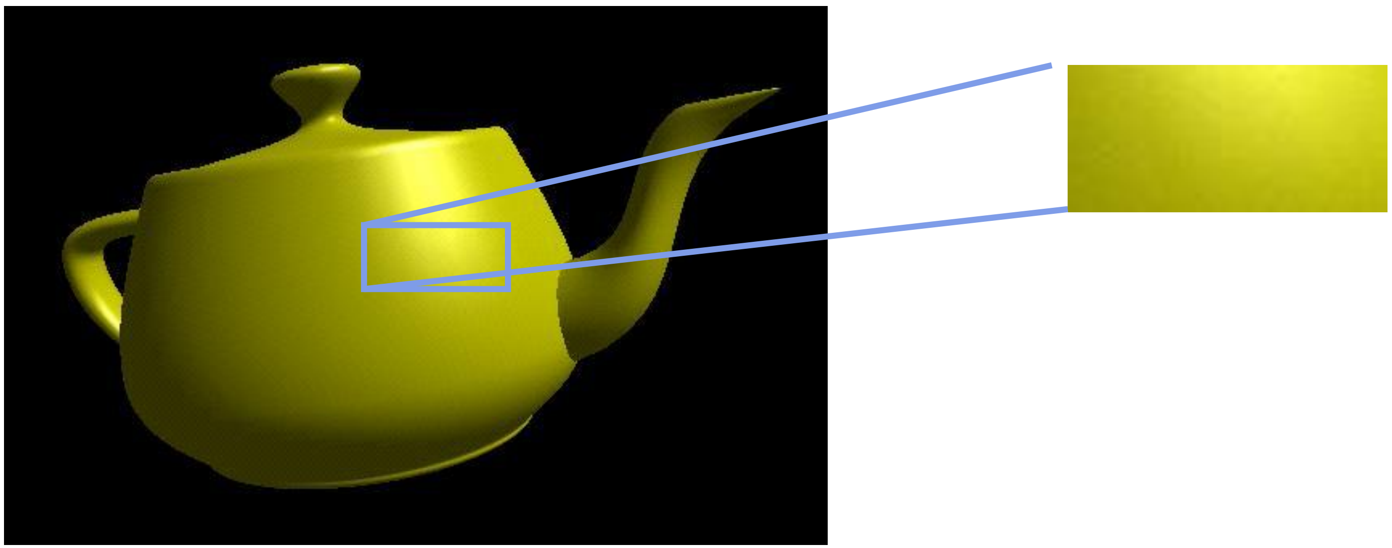
Smooth shading

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



Gouraud Shading

- Lighting calculated for each polygon vertex
- **Colors** are **interpolated** for interior pixels
- Interpolation? Assume linear change across face
- Gouraud shading (interpolation) is OpenGL default



Flat Shading Implementation

- Default is **smooth shading**
- Colors set in vertex shader interpolated
- **Flat shading?** Prevent 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;
```

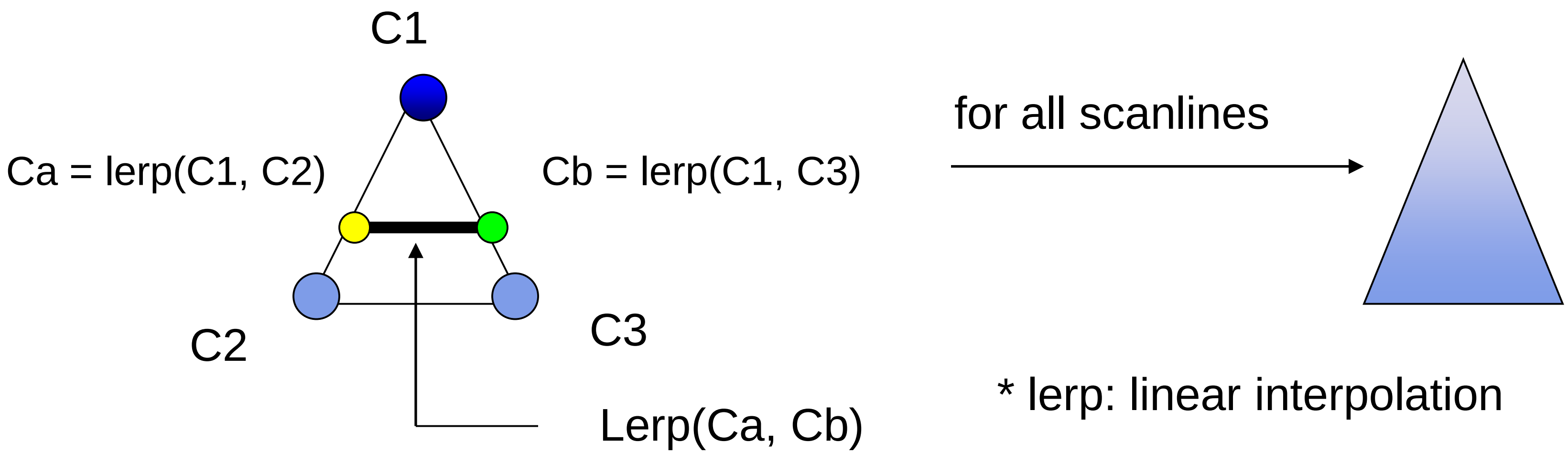
Flat Shading Implementation

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

```
flat in vec4 color;

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

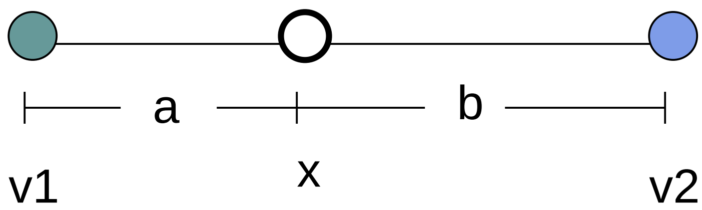
Gouraud Shading

- Compute vertex color in vertex shader
- Shade interior pixels: vertex color **interpolation**



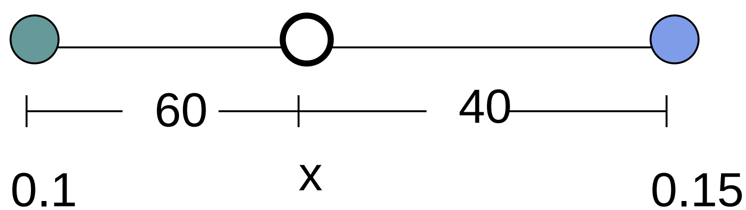
* lerp: linear interpolation

Linear interpolation Example



$$x = \frac{b}{(a+b)} * v1 + \frac{a}{(a+b)} * v2$$

- If a = 60, b = 40
- RGB color at v1 = (0.1, 0.4, 0.2)
- RGB color at v2 = (0.15, 0.3, 0.5)
- Red value of v1 = 0.1, red value of v2 = 0.15

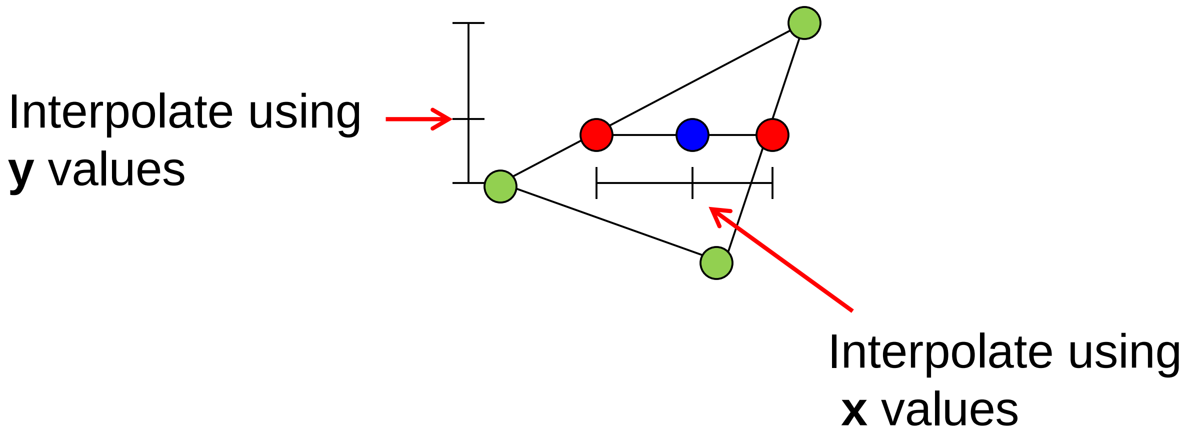


Red value of x = 40 /100 * 0.1 + 60/100 * 0.15
= 0.04 + 0.09 = 0.13

Similar calculations for Green and Blue values

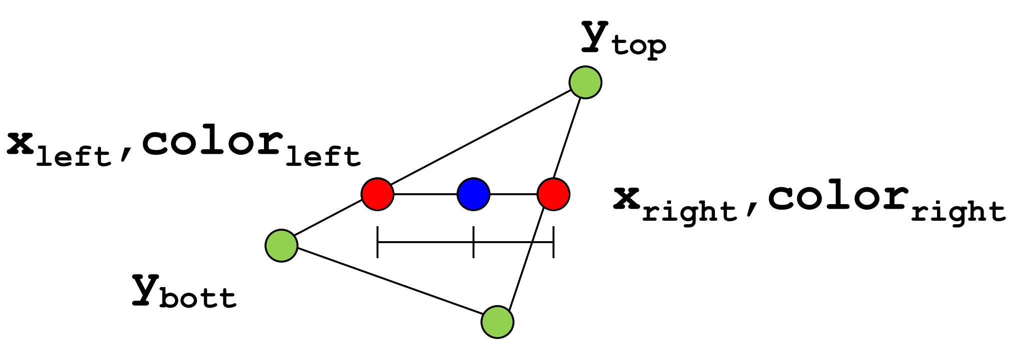
Gouraud Shading

- Interpolate triangle color
  1. Interpolate using **y distance** of end points (green dots) to get color of two end points in scanline (red dots)
  2. Interpolate using **x distance** of two ends of scanline (red dots) to get color of pixel (blue dot)



Gouraud Shading Function (Pg. 433 of Hill)

```
for(int y = y_bott; y < y_top; y++) // for each scan line
{
   find x_left  and x_right
   find color_left  and color_right
   color_inc = (color_right - color_left)/ (x_right - x_left)
   for(int x = x_left, c = color_left; x < x_right; x++, c+ = color_inc)
   {
       put c into the pixel at (x, y)
   }
}
```

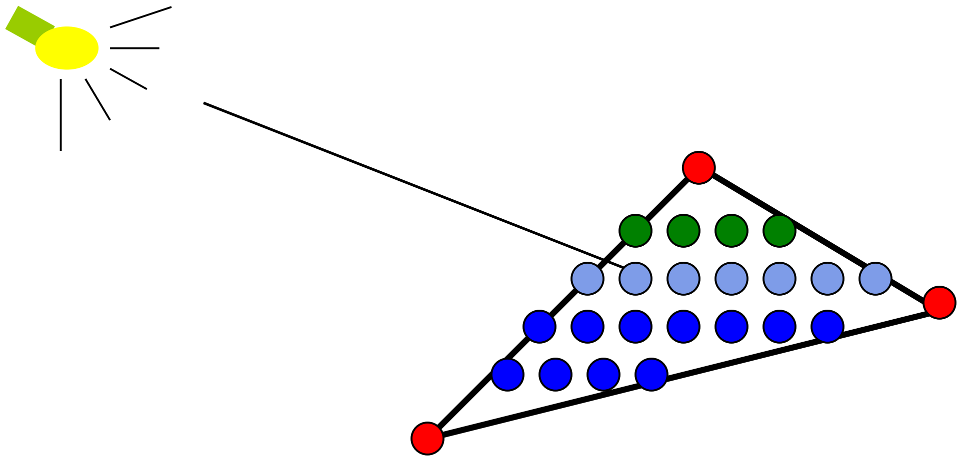


Gouraud Shading Implemenation

- Vertex lighting interpolated across entire face pixels if passed to fragment shader in following way
  1. **Vertex shader:** Calculate output color in vertex shader, Declare output vertex color as **out**

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

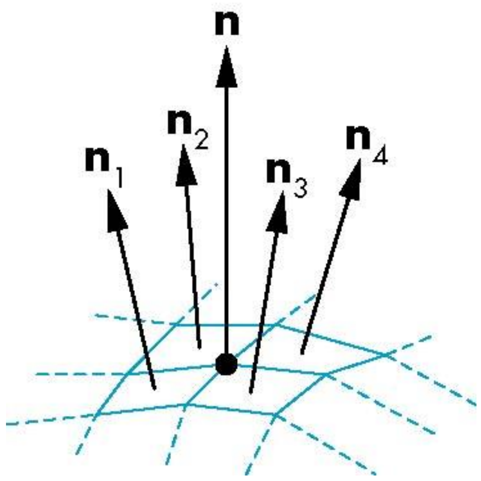
  2. **Fragment shader:** Declare color as **in**, use it, already interpolated!!



Calculating Normals for Meshes

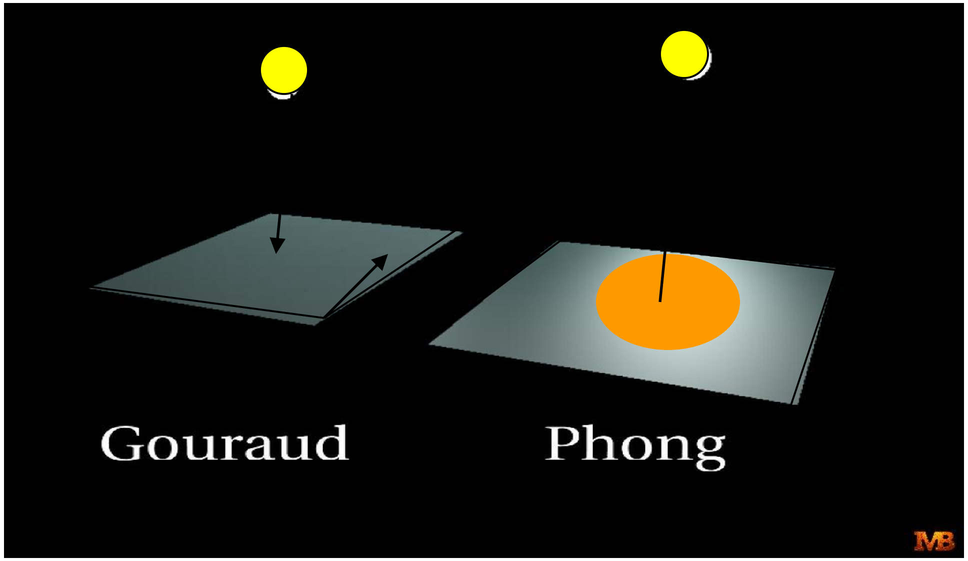
- For meshes, already know how to calculate face normals (e.g. Using Newell method)
- For polygonal models, Gouraud proposed using average of normals around a mesh vertex

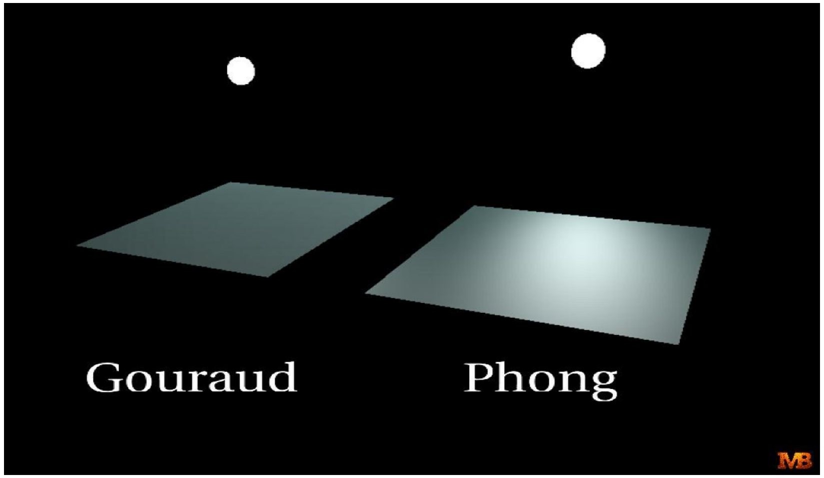
$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4)/\ |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Gouraud Shading Problem

- Assumes linear change across face
- If polygon mesh surfaces have high curvatures, Gouraud shading in polygon interior can be inaccurate
- Phong shading fixes, this, look smooth



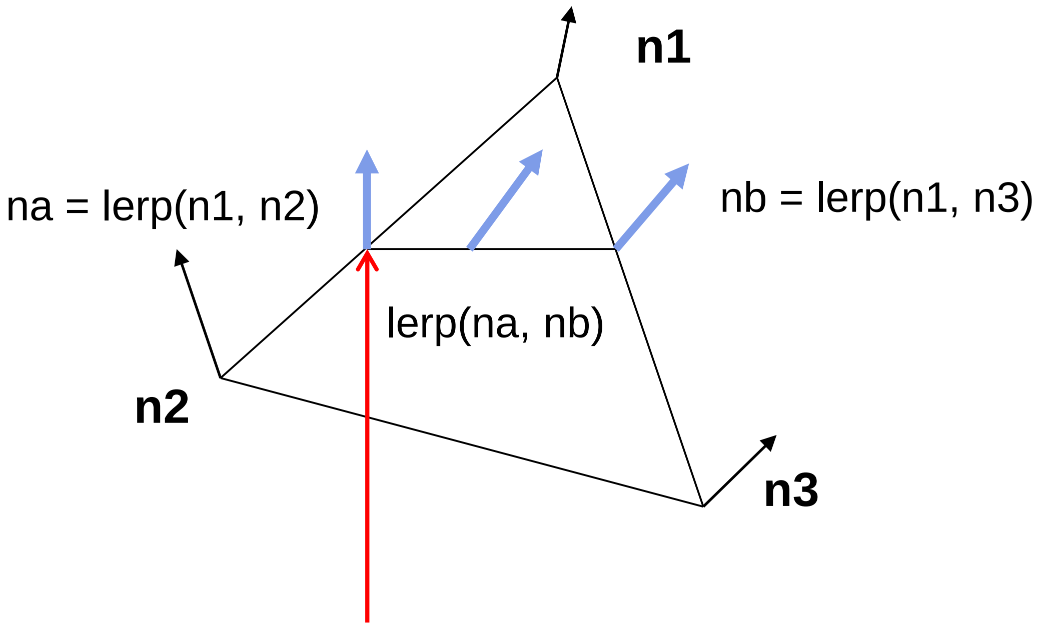


Phong Shading

- Phong shading computes lighting in fragment shader
- Need vectors **n, l, v, r** for each pixels – not provided by user
- Instead of interpolating vertex color
  - Interpolate **vertex normal and vectors**
  - Use pixel **vertex normal and vectors** to calculate Phong lighting at pixel (**per pixel lighting**)

Phong Shading (Per Fragment)

- Normal interpolation (also interpolate l,v)



At each pixel, need to interpolate
Normals (n) and vectors v and l

Gouraud Vs Phong Shading Comparison

- Phong shading:
  - Set up vectors (l,n,v,h) in vertex shader
  - Move lighting calculation to fragment shaders

a. Gouraud Shading

- Set Vectors (l,n,v,h)
- Calculate vertex colors

Hardware Interpolates Vertex color →

- Read/set fragment color
- (Already interpolated)

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

b. Phong Shading

- Set Vectors (l,n,v,h)

Hardware Interpolates Vectors (l,n,v,h) →

- Read in vectors (l,n,v,h)
- (interpolated)
- Calculate fragment lighting

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

Per-Fragment Lighting Shaders I

// vertex shader

```
in vec4 vPosition;
in vec3 vNormal;

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

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

Declare variables **n, v, l** as **out** in vertex shader

Per-Fragment Lighting Shaders II

```
void main()
{
    fN = vNormal;
    fE = -vPosition.xyz;
    fL = LightPosition.xyz;

    if( LightPosition.w != 0.0 ) {
        fL = LightPosition.xyz - vPosition.xyz;
    }

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

Set variables **n, v, l** in vertex shader

Per-Fragment Lighting Shaders III

// 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;
```

Declare vectors n, v, l as **in** in fragment shader
(**Hardware interpolates these vectors**)

Per=Fragment Lighting Shaders IV

```
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;
```

Use interpolated variables n, v, l in fragment shader

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

Per-Fragment Lighting Shaders V

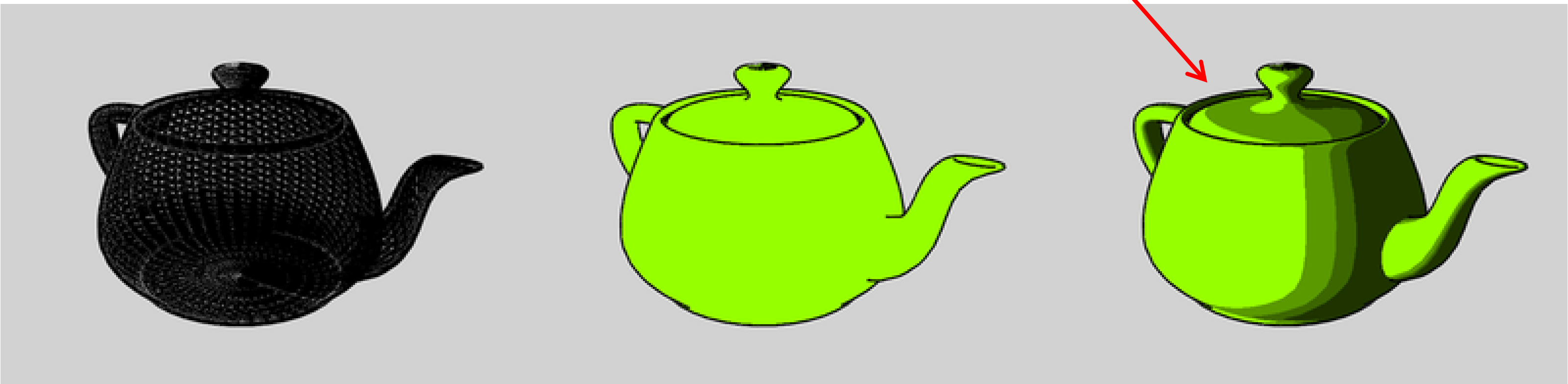
```
float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd*DiffuseProduct;

    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;
}
```

Use interpolated variables n, v, l in fragment shader

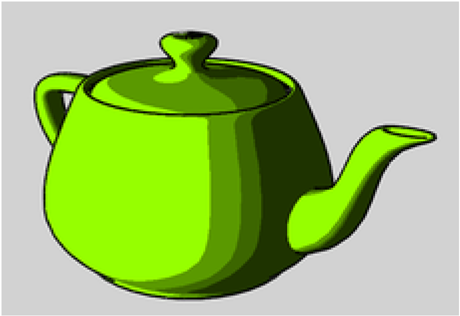
$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

Toon (or Cel) Shading

- Non-Photorealistic (NPR) effect
- Shade in bands of color



Toon (or Cel) Shading



- How?
- Consider **(l ·n)** diffuse term (or cos Θ) term

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

- Clamp values to **min value of ranges** to get toon shading effect

$\mathbf{l} \cdot \mathbf{n}$	Value used
Between 0.75 and 1	0.75
Between 0.5 and 0.75	0.5
Between 0.25 and 0.5	0.25
Between 0.0 and 0.25	0.0

References

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