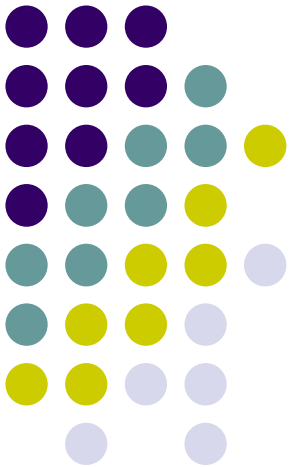
Computer Graphics
CS 543 – Lecture 7 (Part 2)
Lighting, Shading and Materials (Part 2)

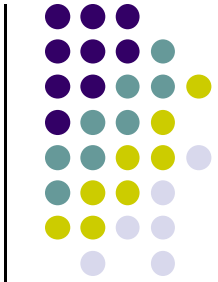
Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)



Modified Phong Model

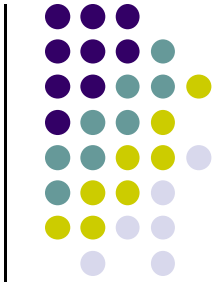


$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$$

- Specular term in Phong model requires calculation new reflection vector **(r)** and view vector **(v)** for each vertex

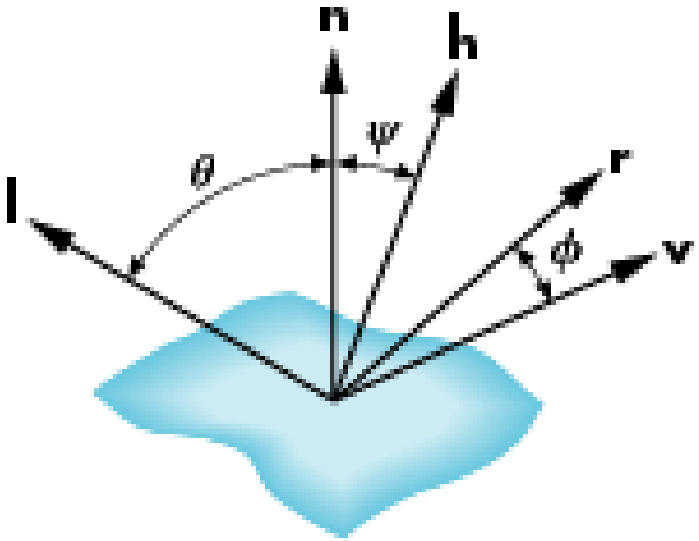
- Blinn suggested approximation using **halfway vector** that is more efficient

The Halfway Vector

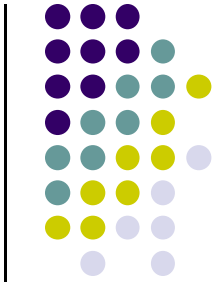


- **h** is normalized vector halfway between **l** and **v**

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

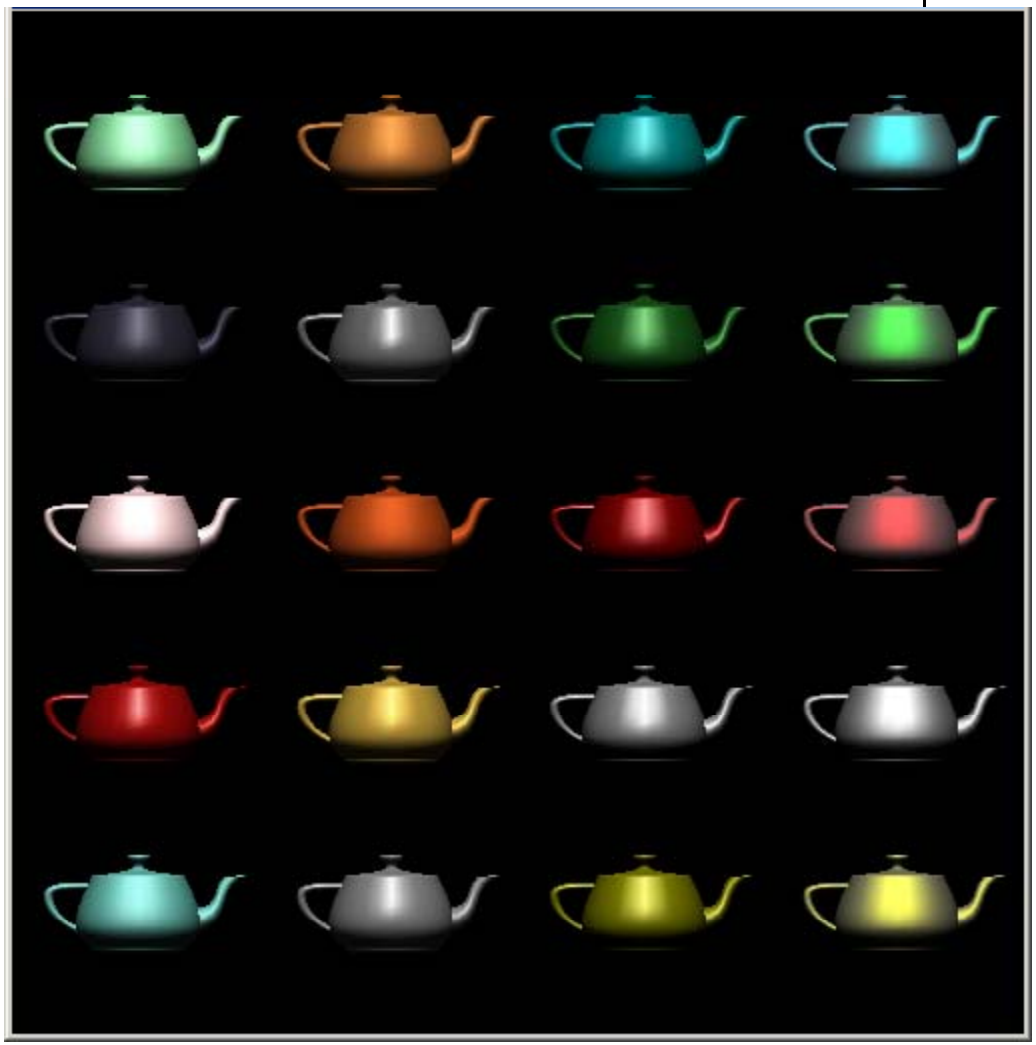


Using the halfway vector



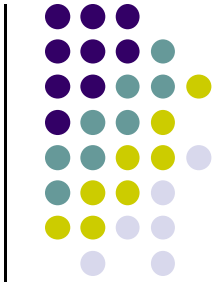
- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- β is chosen to match shininess
- Note that halfway angle is half of angle between $\mathbf{l}$ and $\mathbf{v}$ if vectors are coplanar
- Resulting model is known as the **modified Phong or Blinn lighting model**
  - Specified in OpenGL standard

Example



Only differences in these teapots are the parameters in the modified Phong model

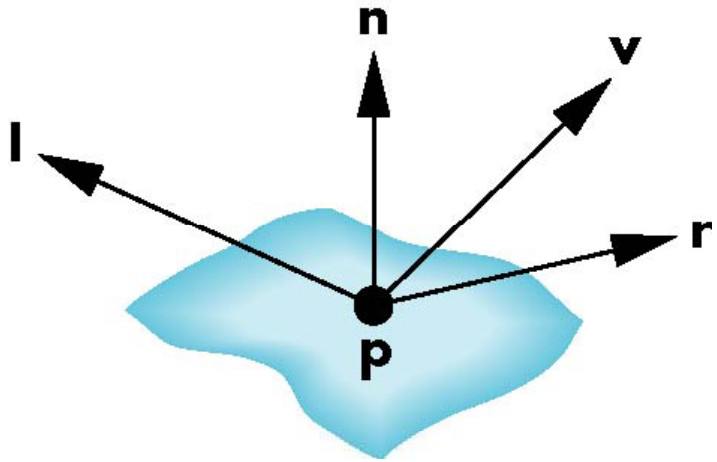
Computation of Vectors



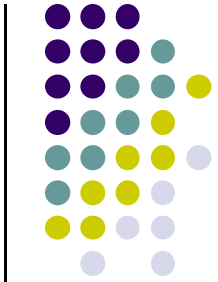
- To calculate lighting at vertex P

  Need **l, n, r** and **v** vector 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!

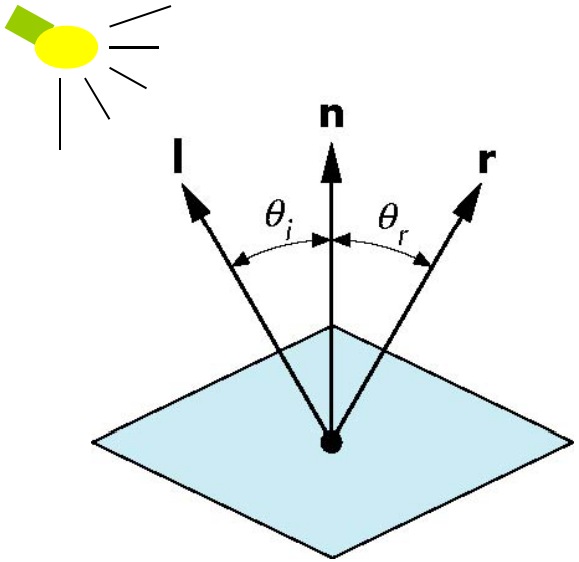


Calculating Mirror Direction Vector r

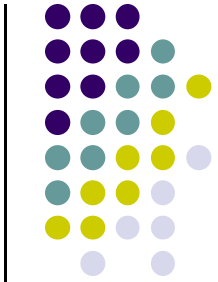


- Can compute **r** from **l** and **n**
- **l**, **n** and **r** are co-planar
- Problem is determining **n**

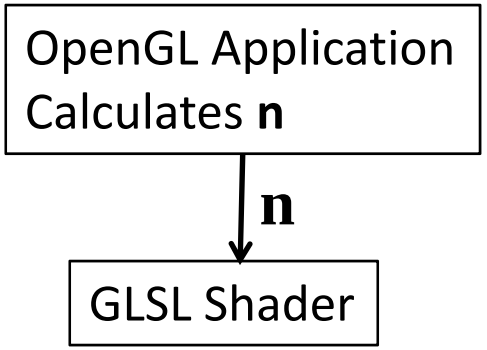
$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n})\,\mathbf{n} - \mathbf{l}$$

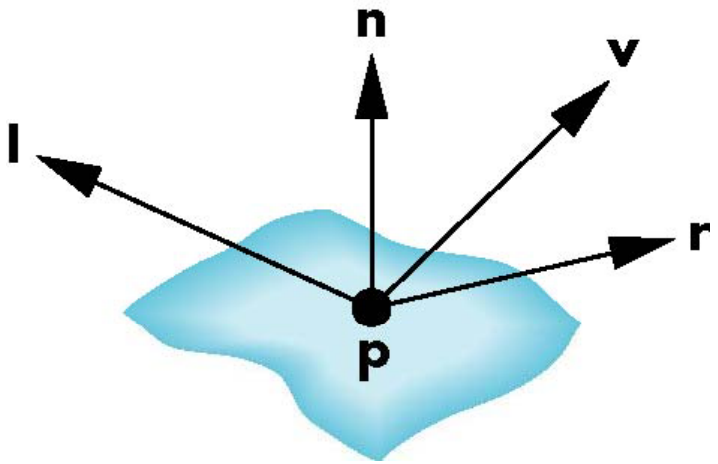


Finding Normal, n

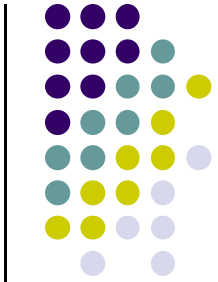


- OpenGL leaves determination of normal to application
  - OpenGL previously calculated normal for GLU quadrics and Bezier surfaces. Now deprecated
- **n** calculation differs depending on surface representation

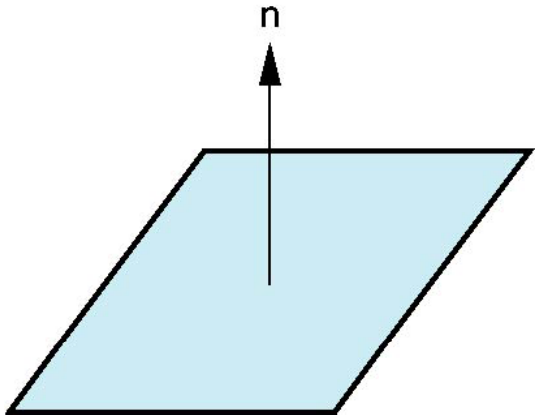




Plane Normals

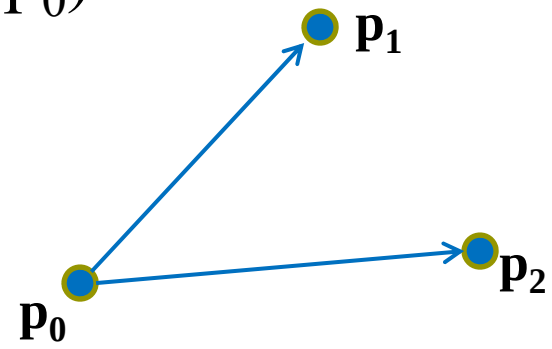


- Equation of plane: $ax+by+cz+d = 0$
- Plane is determined by either
  - three points p_0, p_2, p_3 (on plane)
  - or normal $\mathbf{n}$ and 1 point p_0
- Normal can be obtained by

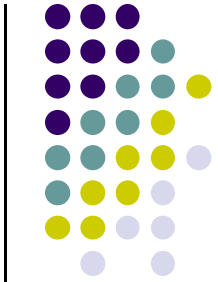


$$\mathbf{n} = (p_2-p_0) \times (p_1-p_0)$$

Cross product method



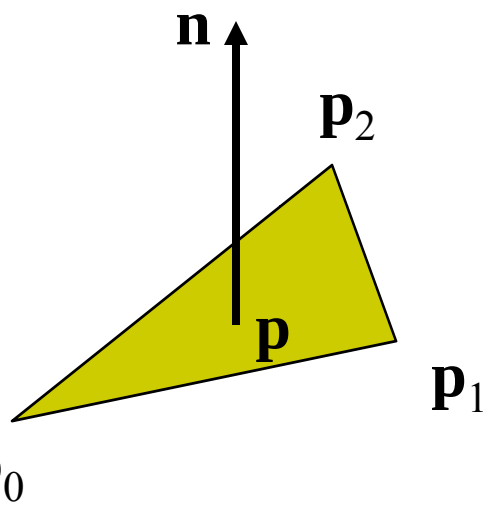
Normal for Triangle



plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

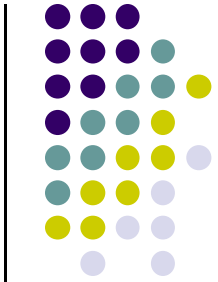
$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$

normalize $\mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$

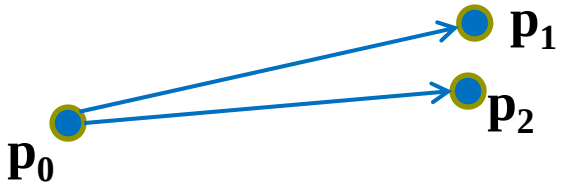


Note that right-hand rule determines outward face

Newell Method for Normal Vectors

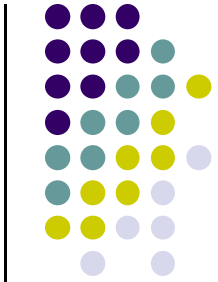


- Problems with cross product method:
  - calculation difficult by hand, tedious
  - If 2 vectors almost parallel, cross product is small
  - Numerical inaccuracy may result



- Proposed by Martin Newell at Utah (teapot guy)
  - Uses formulae, suitable for computer
  - Compute during mesh generation
  - Robust!

Newell Method for Normal Vectors



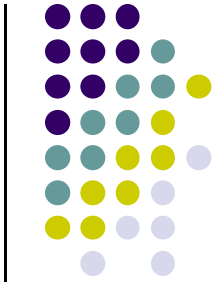
- Formulae: Normal N = (mx, my, mz)

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

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

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

Newell Method Example

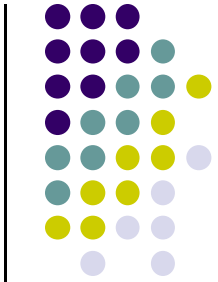


- Example: Find normal of polygon with vertices
  P0 = (6,1,4), P1=(7,0,9) and P2 = (1,1,2)

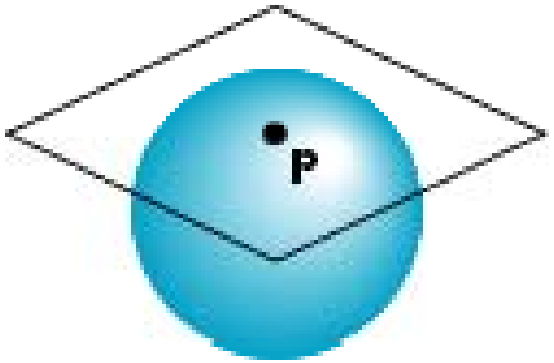
- Using simple cross product:
  ((7,0,9)-(6,1,4)) X ((1,1,2)-(6,1,4)) = (2,-23,-5)

  Using Newell method, plug in values result is same:
  Normal is (2, -23, -5)

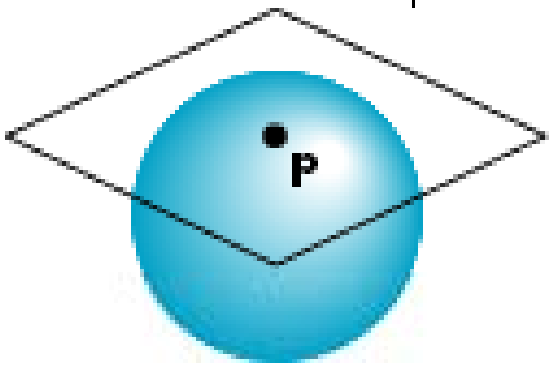
Normal to Sphere



- Implicit function $f(x,y.z)=0$
- Normal given by gradient
- Sphere $f(\mathbf{p})=\mathbf{p}\cdot\mathbf{p}-1$
- $n = [\partial f/\partial x, \partial f/\partial y, \partial f/\partial z]^T=\mathbf{p}$



Parametric Form



- For sphere

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

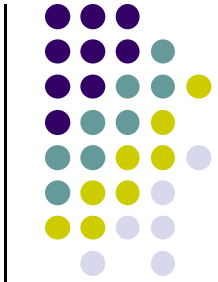
- Tangent plane determined by vectors

$$\partial \mathbf{p}/\partial u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]T$$
$$\partial \mathbf{p}/\partial v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]T$$

- Normal given by cross product

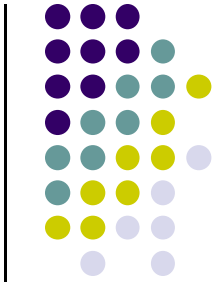
$$\mathbf{n} = \partial \mathbf{p}/\partial u \times \partial \mathbf{p}/\partial v$$

OpenGL shading

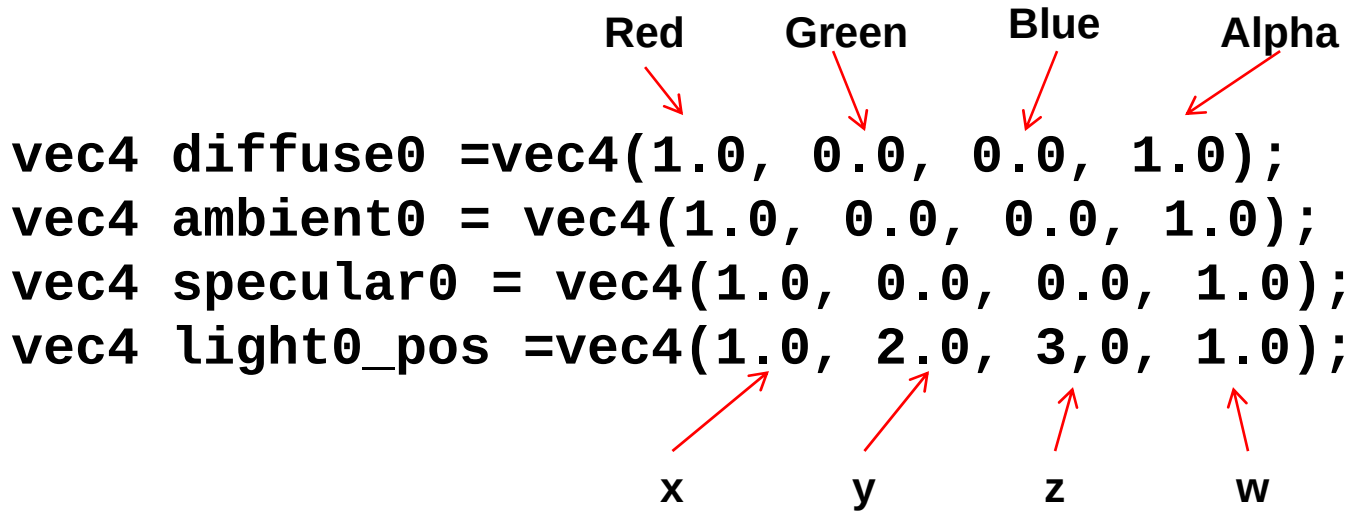


- Need
  - Normals
  - material properties
  - Lights
- State-based shading functions (glNormal, glMaterial, glLight) have been deprecated
- 2 options:
  - Compute lighting in application
  - or send attributes to shaders

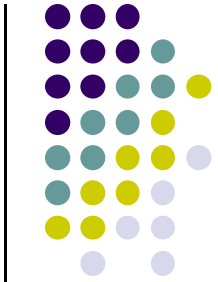
Specifying a Point Light Source



- For each light source, we set RGBA for diffuse, specular, and ambient components, and its position
- Alpha = transparency



Distance and Direction

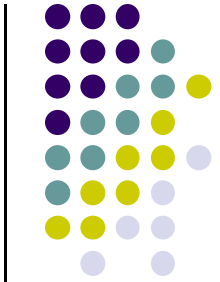


```
vec4 light0_pos =vec4(1.0, 2.0, 3,0, 1.0);
```

- 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

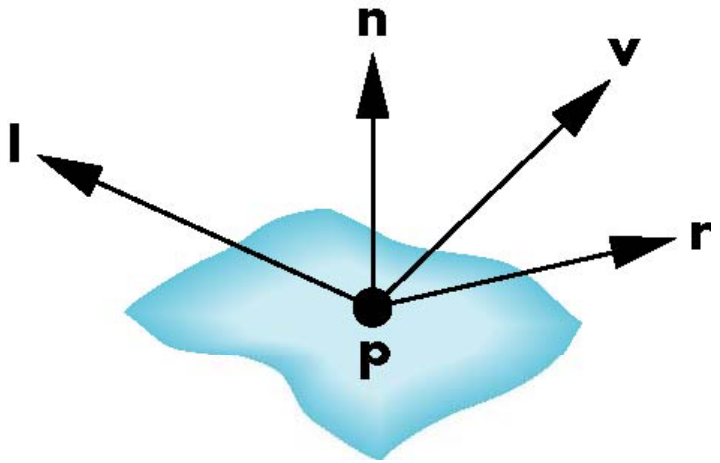
    (x/w = infinity if w = 0)
- Distance term coefficients usually quadratic (1/(a+b*d+c*d*d)) where d is distance from vertex to the light source

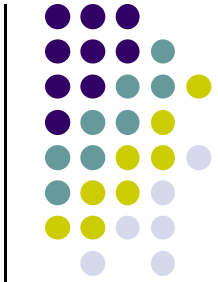
Computation of Vectors



- To calculate lighting at vertex P

  Need **l, n, r** and **v** vector at vertex P
- **l:** Light position – Vertex position
- **v:** Viewer position – vertex position





CTM Matrix passed into Shader

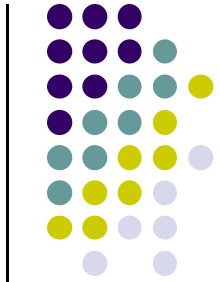
- **Recall: CTM** matrix concatenated in application

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

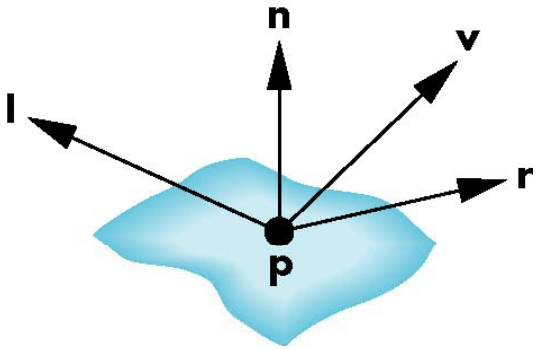
- Connected to matrix **ModelView** in shader
- Recall: CTM matrix contains object transform + Camera

```
in vec4 vPosition;
Uniform mat4 ModelView ;

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

Computation of Vectors

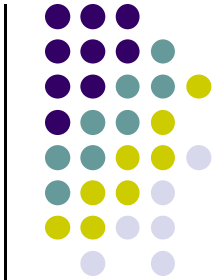


- CTM transforms vertex position into eye coordinates
  - Eye coordinates? Object, light distances measured from eye
- Normalize all vectors! (magnitude = 1)
- GLSL has a **normalize** function
- **Note:** vector lengths affected by scaling

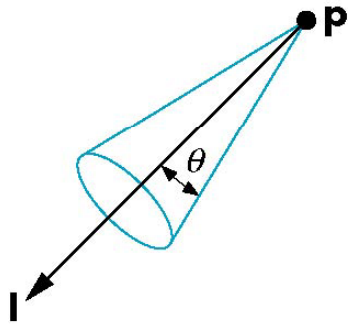


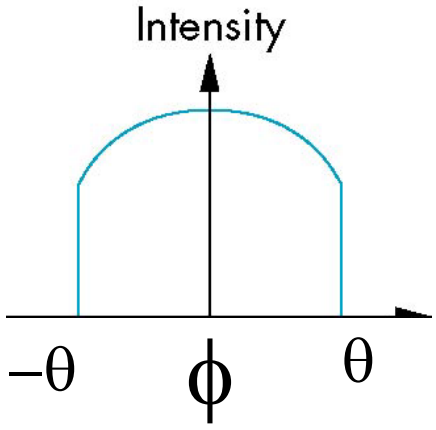

```
// Transform vertex  position into eye coordinates
   vec3 pos = (ModelView * vPosition).xyz;

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

Spotlights

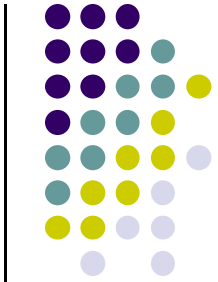


- Derive from point source
  - **Direction l** (of lobe center)
  - **Cutoff:** No light outside θ
  - **Attenuation:** Proportional to $\cos^{\alpha}\phi$



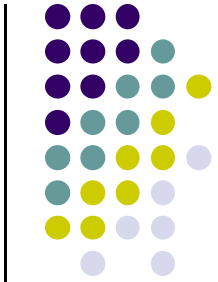


Global Ambient Light



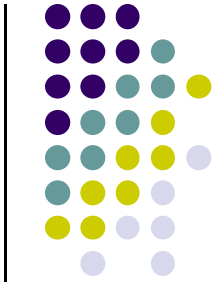
- Ambient light depends on light color
  - Red light in white room will cause a red ambient term
- Previous ambient component added at vertices
- Global ambient term may be added separately **globally**
- Often helpful for testing

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) transformation command, we can
  - Move light source(s) with object(s)
  - Fix object(s) and move light source(s)
  - Fix light source(s) and move object(s)
  - Move light source(s) and object(s) independently

Material Properties



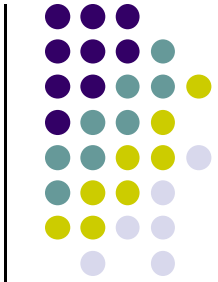
- Material properties also has ambient, diffuse, specular
- Material properties specified as RGBA
- Reflectivities
- w component gives opacity
- **Default?** all surfaces are opaque

```
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
```

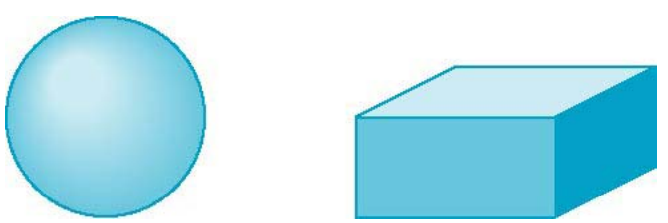
Red Green Blue Opacity

Material Shininess

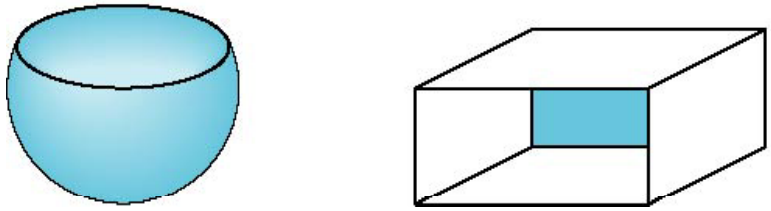
Front and Back Faces



- 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

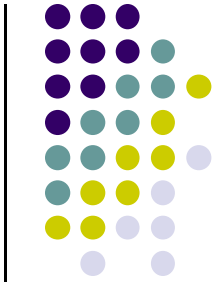


back faces not visible



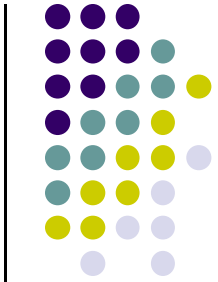
back faces visible

Emissive Term

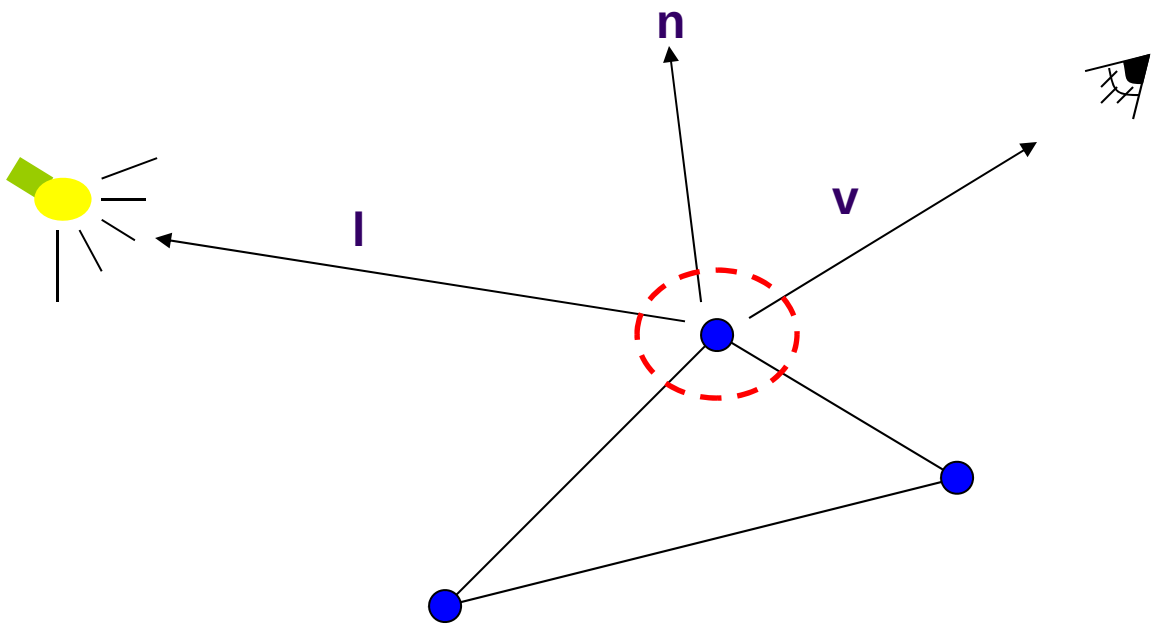


- Some materials glow
- Simulate in OpenGL using emissive component
- This component is unaffected by any sources or transformations

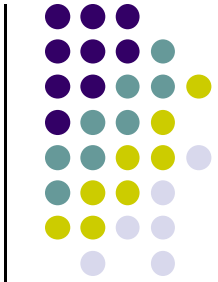
Lighting Calculated Per Vertex



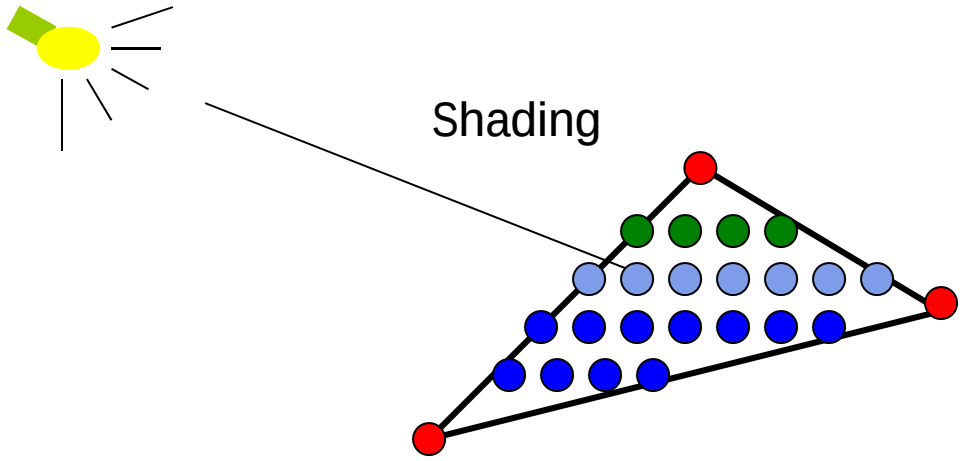
- Phong model (ambient+diffuse+specular) calculated at each vertex to determine vertex color
- Per vertex calculation? Usually done in vertex shader



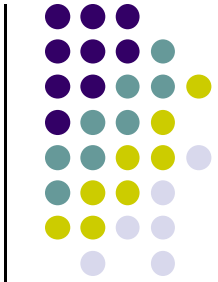
Shading?



- 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
- How? Assume linear change => interpolate



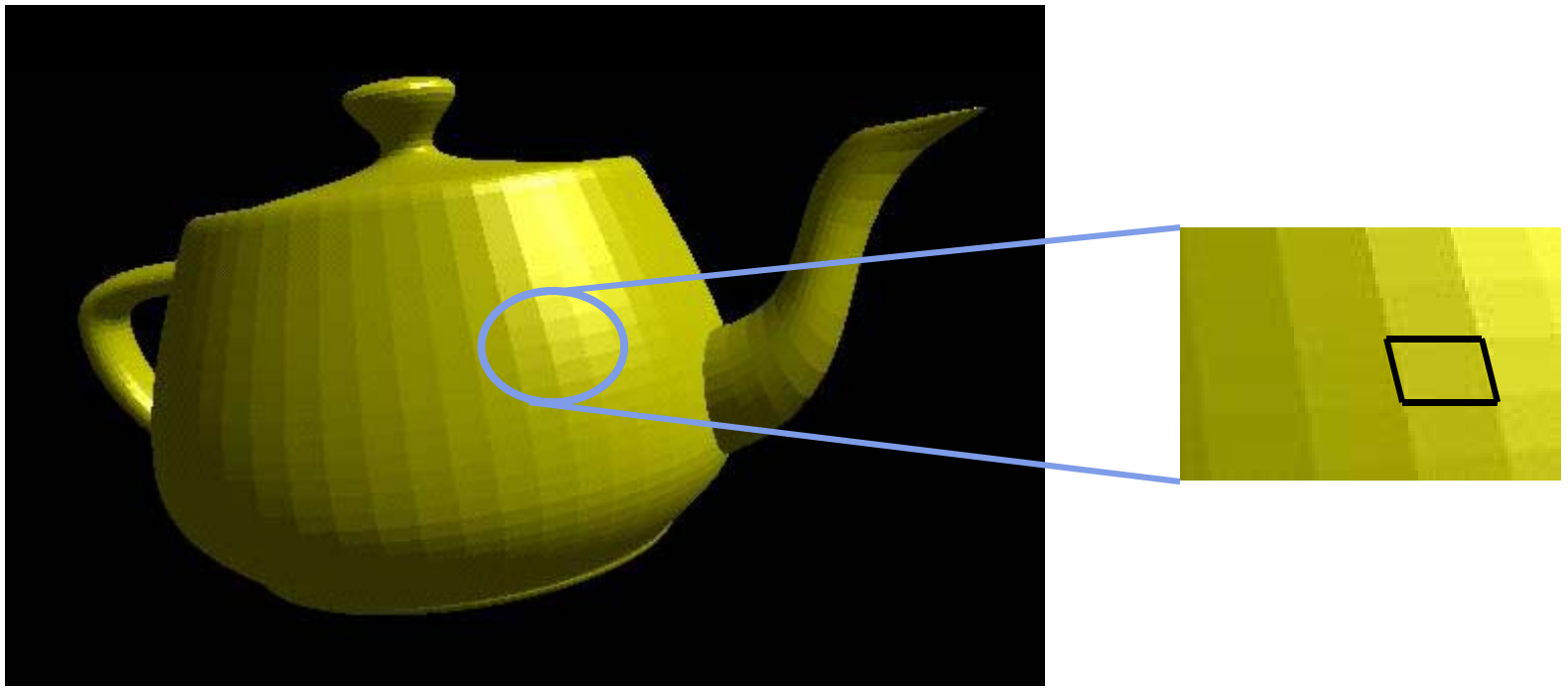
Implementing Polygonal Lighting



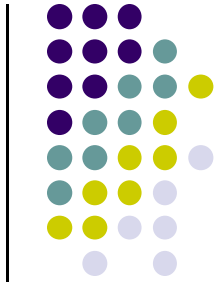
- Per vertex **lighting** calculations can be done either
  - **In application:** Vertex colors become vertex shades and can be sent to vertex shader as vertex attribute
  - **In shader:** send parameters to vertex shader, computer lighting

Flat Shading

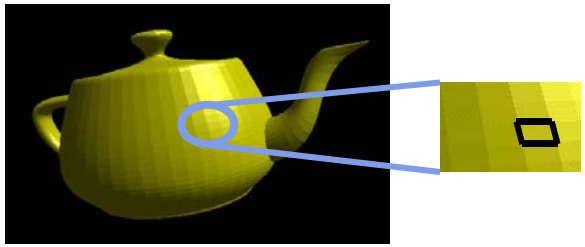
- **2 types of Shading:**
  - Flat shading
  - Smooth shading
- Flat shading - compute lighting once for each face, assign color to whole face



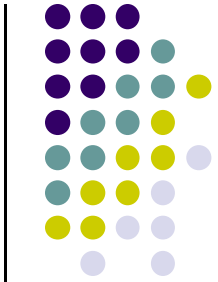
Flat shading



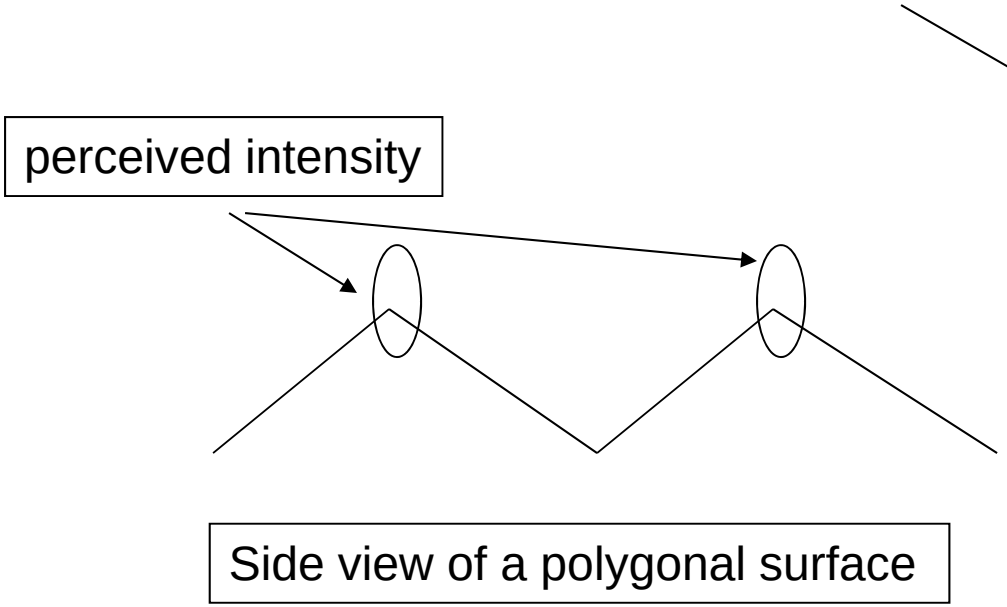
- Only use face normal for all vertices in face and material property to compute color for face
- Benefit: Fast!
- Used when:
  - Polygon is small enough
  - Light source is far away (why?)
  - Eye is very far away (why?)
- Previous OpenGL command: **deprecated!**
  glShadeModel(GL_FLAT)

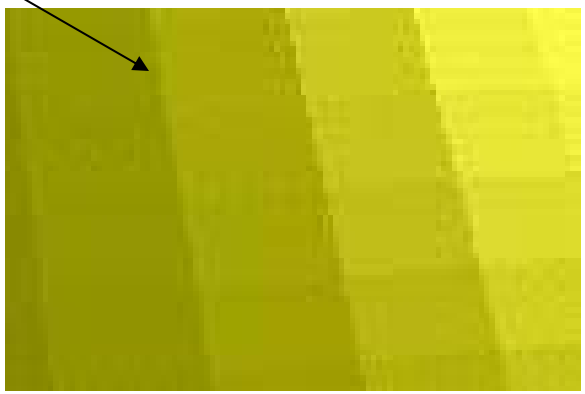


Mach Band Effect

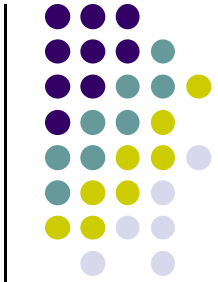


- Flat shading suffers from "mach band effect"
- Mach band effect – human eyes accentuate the discontinuity at the boundary

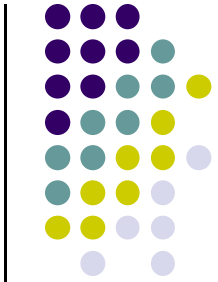




Flat Shading Implementation



- **Flat shading implementation:** Use uniform variable to shade with single shade



References

- Angel and Shreiner
- Hill and Kelley, chapter 8