

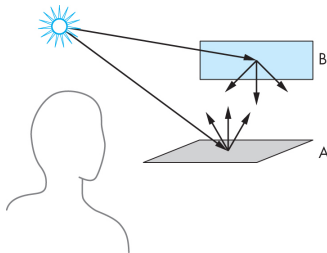
# Lighting and Shading

Thumrongsak Kosiyatrakul  
tkosiyat@cs.pitt.edu

- Without lighting effect, a sphere looks flat regardless of type of projection
- A lighting effect gives two-dimensional images an effect that make it looks like three-dimensional images
- We need to model the following:
  - light sources
  - light-material interactions
- Lighting model can be applied in various parts:
  - application,
  - vertex shader, or
  - fragment shader

# Light and Matter

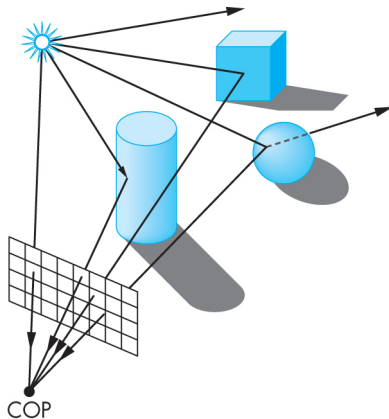
- We are going to render our image based on physics
- A surface can either emits light, reflect light, or both
- The color of a point of an object is a result of multiple interactions



- Light can reflect back and forth among objects multiple times before it reaches the viewer
  - Requires a lot of calculations which is not suitable for real-time rendering
  - Need some approximate approaches

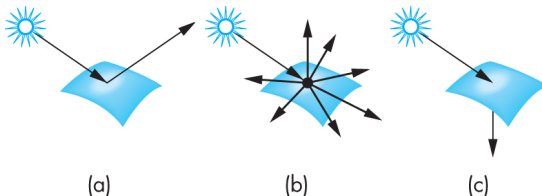
# Light and Matter

- We are only interested in light that enter the viewer (projection) plane



# Light-Material Interaction

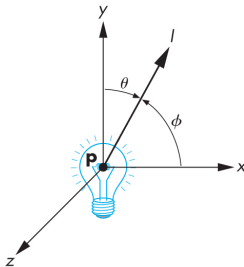
- Light interacts with materials differently:



- (a) **Specular surfaces:** Shiny — reflect light in a narrow angle
- (b) **Diffuse surfaces:** Matte — reflect light equally in all direction
- (c) **Translucent surfaces:** Transparent — let light pass through (reflect some)

# Light Sources

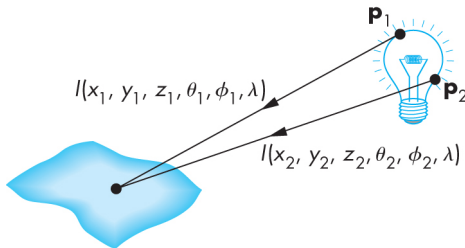
- A light source emits light
- Every point  $(x, y, z)$  on the surface of a light source has its own characteristic:
  - intensity of energy emitted at each wave length  $\lambda$ , and
  - direction of emission  $\theta$  and  $\phi$



- Illumination function  $I(x, y, z, \theta, \phi, \lambda)$

# Light Sources

- For multiple light sources, we need to integrate all light sources



- We are going to use human visual system which is based on three colors, red, green, and blue
- Each color has its own characteristic
- Thus each light source can be defined as **luminance** function as

$$\mathbf{I} = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}$$



# Ambient Light

- Ambient light gives a uniform illumination
- For simplicity, assuming that intensity of ambient light is identical at every point in the scene
- An ambient source also contains three color components:

$$\mathbf{I}_a = \begin{bmatrix} I_{ar} \\ I_{ag} \\ I_{ab} \end{bmatrix}$$

# Point Sources

- A point source emits light evenly in all directions
- The characteristic of a point source at  $\mathbf{p}_0$  can be defined as

$$\mathbf{I}(\mathbf{p}_0) = \begin{bmatrix} I_r(\mathbf{p}_0) \\ I_g(\mathbf{p}_0) \\ I_b(\mathbf{p}_0) \end{bmatrix}$$

- Intensity of light decreases over the distance (inverse square)
- At point  $\mathbf{p}$ , the intensity of light from the point source at  $\mathbf{p}_0$  is given by

$$\mathbf{i}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{|\mathbf{p} - \mathbf{p}_0|^2} \mathbf{I}(\mathbf{p}_0)$$

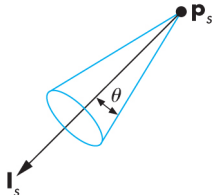
- For a softer effect, we use

$$\mathbf{i}(\mathbf{p}, \mathbf{p}_0) = \frac{1}{a + bd + cd^2} \mathbf{I}(\mathbf{p}_0)$$

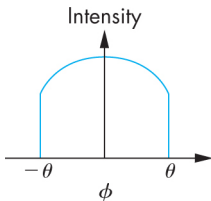
where  $d$  is the distance between  $\mathbf{p}$  and  $\mathbf{p}_0$

# Spotlights

- A spotlight create a narrow range of angle of illumination:



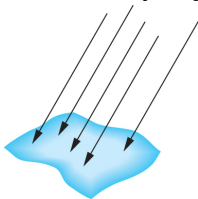
- Ideally, the intensity of a spotlight is 0 when a point is outside the cone:



- Generally use  $\cos^e \theta$  where  $e$  determines how rapidly the light intensity drops off

# Distance Light Sources

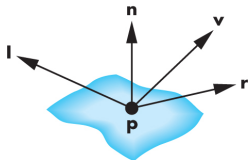
- The sun is an example of a distance light source
- Light from the sun hit all objects in the same direction
- For simplicity, we can use one vector to represent the direction of the light for every point on every object



- This can easily be done by simply change the location of the distance light source to a vector

$$\mathbf{p}_0 = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightsquigarrow \mathbf{p}_0 = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

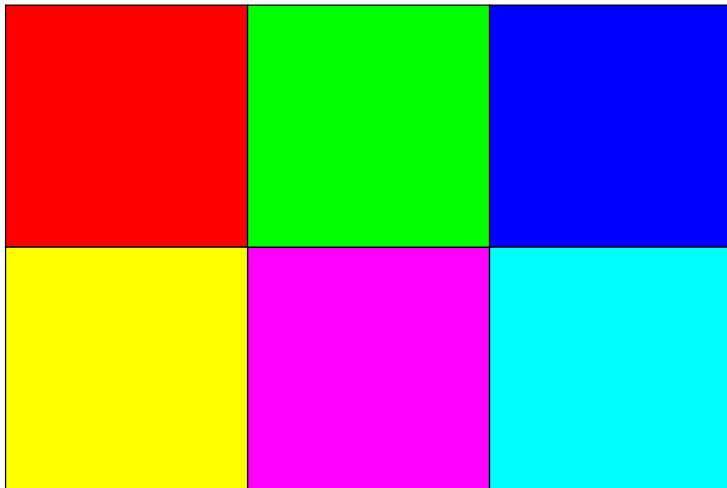
# The Phong Reflection Model



- Uses four vectors to calculate a color for an arbitrary point  $p$  on a surface
  - $n$  is the normal at  $p$
  - $v$  is in the direction from  $p$  to the viewer (COP)
  - $l$  is in the direction of a line from  $p$  to light source
  - $r$  is in the direction that a perfectly reflected ray from  $l$  would take
- Support three types of material-light interactions
  - Ambient
  - Diffuse
  - Specular

# Light Colors

- A light source can have different color



# The Phong Reflection Model

- A light source has ambient, diffuse, and specular terms
- Each term contains three colors, red, green, and blue
- For any point  $\mathbf{p}$  on a surface need a  $3 \times 3$  illumination matrix for each light source  $i$ :

$$\mathbf{L}_i = \begin{bmatrix} L_{ira} & L_{iga} & L_{iba} \\ L_{ird} & L_{igd} & L_{ibd} \\ L_{irs} & L_{igs} & L_{ibs} \end{bmatrix}$$

- the first row represents ambient intensities of red, green, and blue
  - the second row represents diffuse intensities, and
  - the last row represents specular intensities.
- Code

```
vec4 light_i_ambient, light_i_diffuse, light_i_specular;
```

Use vec4 to support translucent surface

# Ambient Component of a Light Source

- **Ambient** component indicates the ambient light caused by a light source
- **Diffuse** component indicates the direct light
- **Specular** component indicates the light that will be used for reflection
- Generally diffuse and specular components are the same



# The Phone Reflection Model

- For each term  $L_{ix}$  contributes to the intensity at a point  $\mathbf{p}$  by  $R_{ix}L_{ix}$ .
- $R_{ix}$  depends on the following:
  - the material properties,
  - the orientation of the surface,
  - the direction of the light source, and
  - the distance between the light source and the viewer
- So, for each point, we need nine coefficients

$$\mathbf{R}_i = \begin{bmatrix} R_{ira} & R_{iga} & R_{iba} \\ R_{ird} & R_{igd} & R_{ibd} \\ R_{irs} & R_{igs} & R_{ibs} \end{bmatrix}$$

- Code

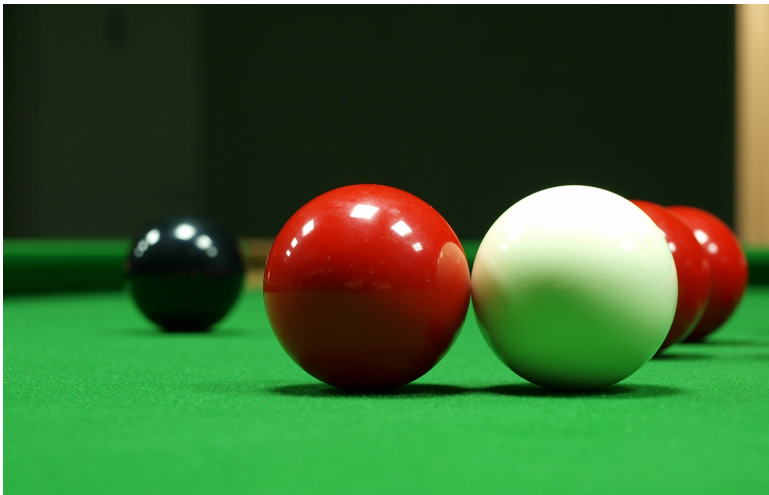
```
vec4 reflect_i_ambient, reflect_i_diffuse, reflect_i_specular;
```

- Note that the reflective properties depend on the surface material

- **Ambient** color is the color of an object where it is in shadow. This color is what the object reflect when illuminated by ambient light rather than direct light
- **Diffuse** color is the color of the object under pure white light
- **Specular** color is the color of the light of a specular reflection

# Lighting Model

- Let's look at a photo



# The Phone Reflection Model

- For each source  $i$ , the red intensity that we see at  $\mathbf{p}$  is

$$\begin{aligned} I_{ir} &= R_{ira} L_{ira} + R_{ird} L_{ird} + R_{irs} L_{irs} \\ &= I_{ira} + I_{ird} + I_{irs} \end{aligned}$$

- The total red intensity (from all sources) is

$$I_r = \sum_i (I_{ira} + I_{ird} + I_{irs}) + I_{ar}$$

where  $I_{ar}$  is the red component of the global ambient light

- Similarly, total green and blue intensity from all sources are

$$\begin{aligned} I_g &= \sum_i (I_{iga} + I_{igd} + I_{igs}) + I_{ag} \\ I_b &= \sum_i (I_{iba} + I_{igb} + I_{ibs}) + I_{ab} \end{aligned}$$

# The Phone Reflection Model

- Since computation are the same for each source and each primary color, we can omit subscripts  $i$ ,  $r$ ,  $g$ , and  $b$  for simplicity

$$I = I_a + I_d + I_s = L_a R_a + L_d R_d + L_s R_s$$

# Ambient Reflection

- The intensity of ambient light is the same at every point on the surface
- The ambient reflection  $R_a$  can simply be  $k_a$  where

$$0 \leq k_a \leq 1$$

- If  $k_a = 1$ , it reflects everything back
- If  $0 < k_a < 1$ , some get absorbed by the surface
- If  $k_a = 0$ , it absorbs all the light
- Thus the intensity is  $I_a = k_a L_a$  which is a short notation for:

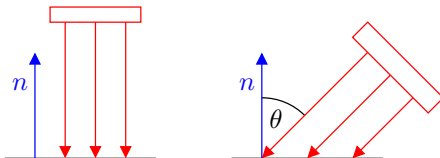
$$I_{ra} = k_{ra} L_{ra} = \sum_i (k_{ira} L_{ira})$$

$$I_{ga} = k_{ga} L_{ga} = \sum_i (k_{iga} L_{iga})$$

$$I_{ba} = k_{ba} L_{ba} = \sum_i (k_{iba} L_{iba})$$

# Diffuse Reflection

- A diffuse surface scatters the light equally in all direction
- So, we need to consider the angle that the light hit the surface and the distance of the light source
- If the light source coming in an angle, it needs to spread over a larger area.



- The diffuse reflection  $R_d$  is proportional to  $\cos \theta$
- According to the dot product, we have

$$\cos \theta = \mathbf{l} \cdot \mathbf{n}$$

where  $\mathbf{n}$  is the plane normal (unit length) and  $\mathbf{l}$  is the direction of the light source (unit length)

# Diffuse Reflection

- We can also add in a reflection factor  $k_d$  which gives us

$$I_d = k_d(\mathbf{l} \cdot \mathbf{n})L_d$$

- The distance term can be incorporated using the modified inverse-square

$$I_d = \frac{k_d}{a + bd + cd^2}(\mathbf{l} \cdot \mathbf{n})L_d$$

where  $d$  is the distance and  $a$ ,  $b$ , and  $c$  are constants.

- Note that  $\mathbf{l} \cdot \mathbf{n}$  can be negative if the  $\theta > 90^\circ$  or  $\theta < -90^\circ$ .
  - We need to set  $\mathbf{l} \cdot \mathbf{n}$  to 0 if it happens using  $\max(\mathbf{l} \cdot \mathbf{n}, 0)$
- So, the final equation:

$$I_d = \frac{k_d}{a + bd + cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0)L_d$$



- Thus, for each color and all light sources, we have

$$\begin{aligned} I_{\text{rd}} &= \frac{k_{\text{rd}}}{a + bd + cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_{\text{rd}} \\ &= \sum_i \left( \frac{k_{i\text{rd}}}{a + bd_i + cd_i^2} \max(\mathbf{l}_i \cdot \mathbf{n}, 0) L_{i\text{rd}} \right) \end{aligned}$$

$$\begin{aligned} I_{\text{gd}} &= \frac{k_{\text{gd}}}{a + bd + cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_{\text{gd}} \\ &= \sum_i \left( \frac{k_{i\text{gd}}}{a + bd_i + cd_i^2} \max(\mathbf{l}_i \cdot \mathbf{n}, 0) L_{i\text{gd}} \right) \end{aligned}$$

$$\begin{aligned} I_{\text{bd}} &= \frac{k_{\text{bd}}}{a + bd + cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_{\text{bd}} \\ &= \sum_i \left( \frac{k_{i\text{bd}}}{a + bd_i + cd_i^2} \max(\mathbf{l}_i \cdot \mathbf{n}, 0) L_{i\text{bd}} \right) \end{aligned}$$

# Specular Reflection

- Without specular reflection, a surface will look dull (not shiny)
- We use Phong model

$$I_s = k_s L_s \cos^\alpha \phi$$

where

- $k_s$  is the coefficient ( $0 \leq k_s \leq 1$ )
- $\phi$  is the angle between the reflection direction  $\mathbf{r}$  and the direction of the viewer  $\mathbf{v}$
- $\alpha$  is a shininess coefficient (the larger the shinier)
  - $\infty$  represents a perfect mirror
  - 100 – 500 for metallic
- As usual, if  $\mathbf{r}$  and  $\mathbf{v}$  are unit length vector,  $\cos \phi = \mathbf{r} \cdot \mathbf{v}$
- Note that  $\mathbf{r} \cdot \mathbf{v}$  can be negative and the distance term can be incorporated
- The final equation:

$$I_s = \frac{k_s}{a + bd + cd^2} L_s \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha$$

# Specular Reflection

- Thus, for each color and all light sources, we have

$$\begin{aligned} I_{rs} &= \frac{k_{rs}}{a + bd + cd^2} L_{rs} \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha \\ &= \sum_i \left( \frac{k_{irs}}{a + bd_i + cd_i^2} L_{irs} \max(\mathbf{r}_i \cdot \mathbf{v}, 0)^\alpha \right) \\ I_{gs} &= \frac{k_{gs}}{a + bd + cd^2} L_{gs} \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha \\ &= \sum_i \left( \frac{k_{igs}}{a + bd_i + cd_i^2} L_{igs} \max(\mathbf{r}_i \cdot \mathbf{v}, 0)^\alpha \right) \\ I_{bs} &= \frac{k_{bs}}{a + bd + cd^2} L_{bs} \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha \\ &= \sum_i \left( \frac{k_{ibs}}{a + bd_i + cd_i^2} L_{ibs} \max(\mathbf{r}_i \cdot \mathbf{v}, 0)^\alpha \right) \end{aligned}$$

# The Phong Reflection Model

- Recall that we have

- $I_a = k_a L_a,$
- $I_d = \frac{k_d}{a+bd+cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_d,$  and
- $I_s = \frac{k_s}{a+bd+cd^2} L_s \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha.$

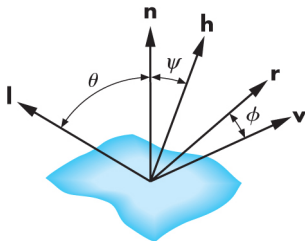
- Finally we have the Phong reflection model as

$$I = \frac{1}{a + bd + cd^2} (k_d L_d \max(\mathbf{l} \cdot \mathbf{n}, 0) + K_s L_s \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha) + k_a L_a$$

- Note that the above formula is computed for each light source and for each primary color.
- Drawback:  $\mathbf{r} \cdot \mathbf{v}$  must be recalculated for every point on the surface:
  - $\mathbf{r}$ : reflection vector is slightly different
  - $\mathbf{v}$ : vector to the viewer is slightly different

# The Modified Phong Model

- Modified Phong model use the unit vector halfway between the viewer vector and the light-source vector:

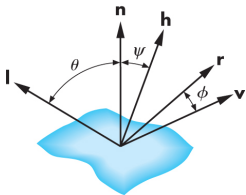


where

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{|\mathbf{l} + \mathbf{v}|}$$

- Adding two unit vectors results in a vector halfway between the two

# The Modified Phong Model



- The angle between  $\mathbf{l}$  and  $\mathbf{n}$  ( $\theta$ ) must be the same as the angle between  $\mathbf{n}$  and  $\mathbf{r}$  since  $\mathbf{r}$  is the perfect reflection.
- Thus, the angle from  $\mathbf{l}$  to  $\mathbf{v}$  is  $2\theta + \phi$ .
- The angle from  $\mathbf{l}$  to  $\mathbf{h}$  ( $\theta + \psi$ ), is the same as the angle from  $\mathbf{h}$  to  $\mathbf{v}$  (halfway)
- Thus, the angle from  $\mathbf{l}$  to  $\mathbf{v}$  is  $2(\theta + \psi)$
- So, we have

$$2\theta + \phi = 2(\theta + \psi)$$

$$= 2\theta + 2\psi$$

$$\phi = 2\psi$$

# The Modified Phong Model

- We can avoid calculating  $\mathbf{r} \cdot \mathbf{v}$  by simply use  $\mathbf{n} \cdot \mathbf{h}$ .
  - We do not need to calculate  $\mathbf{r}$  since to calculate  $\mathbf{n} \cdot \mathbf{h}$ , we only need  $\mathbf{l}$  and  $\mathbf{v}$
- However, the angle is smaller ( $2\psi = \phi$ )
- We can adjusted it since we know that  $2\psi = \phi$  but we end up using the same amount of calculation
- One way to reduce the calculation is to replace  $(\mathbf{r} \cdot \mathbf{v})^e$  by  $(\mathbf{n} \cdot \mathbf{h})^{e'}$  where  $(\mathbf{n} \cdot \mathbf{h})^{e'} \approx (\mathbf{r} \cdot \mathbf{v})^e$

# Computation of Vectors

- To use Phong reflection model, we need to know the following vectors for each point  $\mathbf{p}$  on a surface:
  - $\mathbf{n}$  is the normal vector at  $\mathbf{p}$
  - $\mathbf{v}$  is in the direction from  $\mathbf{p}$  to the viewer (COP)
  - $\mathbf{l}$  is in the direction of a line from  $\mathbf{p}$  to a light source
  - $\mathbf{r}$  is in the direction that a perfectly reflected ray from  $\mathbf{l}$  would take
- We should normalize these vectors to unit vectors for simplicity



# Normal Vectors

- A normal vector  $\mathbf{n}$  of a plane is a vector perpendicular to the plane
  - Perpendicular to every point on a flat plane
- Suppose  $\mathbf{p}_0$  is a known point on the plane and  $\mathbf{p}$  is a point on a plane
  - A vector  $\mathbf{p} - \mathbf{p}_0$  is also perpendicular to  $\mathbf{n}$ . From dot product, we have

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

- Let  $n = (a, b, c)$ ,  $p = (x, y, z)$ , and  $p_0 = (x_0, y_0, z_0)$ , we have

$$\begin{aligned}\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) &= 0 = (a, b, c) \cdot ((x, y, z) - (x_0, y_0, z_0)) \\ &= (a, b, c) \cdot (x - x_0, y - y_0, z - z_0) \\ &= a(x - x_0) + b(y - y_0) + c(z - z_0) \\ &= ax + by + cz - (ax_0 + by_0 + cz_0) \\ &= ax + by + cz + d\end{aligned}$$

where  $d = -(ax_0 + by_0 + cz_0)$ .

- Formally a plane (surface) is given by  $f(x, y, z) = k$  for some constant  $k$ .

# Normal Vectors

- For example, the equation of the surface of a unit sphere center at the origin is given by

$$x^2 + y^2 + z^2 = 1$$

where  $f(x, y, z) = x^2 + y^2 + z^2$ .

- This surface consists of various points.
  - An example of a point on this sphere is  $P_0 = (1, 0, 0)$
- This surface consists of various lines
  - An example of a line is  $g(t) = (x(t), y(t), z(t))$  where

$$x(t) = \cos t \quad y(t) = \sin t \quad z(t) = 0$$

- Note that this line pass through  $P_0$  since when  $t = 0$ ,  
 $g(t) = (1, 0, 0) = P_0$

# Normal Vectors

- Suppose we have a surface  $f(x, y, z) = k$
- Let  $P = (x_0, y_0, z_0)$  be a point on this surface
- Let  $g(t) = (x(t), y(t), z(t))$  be a line on this surface where  $g(t_0) = (x_0, y_0, z_0) = P$
- Since  $g(t)$  is a line on the surface, every point generated by  $g(t)$  must satisfy the equation of the surface

$$f(x(t), y(t), z(t)) = k$$

- From the Chain Rule, we have

$$\frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt} + \frac{\partial f}{\partial z} \frac{dz}{dt} = 0$$

which is a dot product

$$\nabla f \cdot g'(t) = 0$$

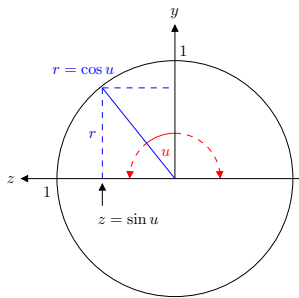
- At  $t = t_0$ , we have

$$\nabla f(x_0, y_0, z_0) \cdot g'(t_0) = 0$$

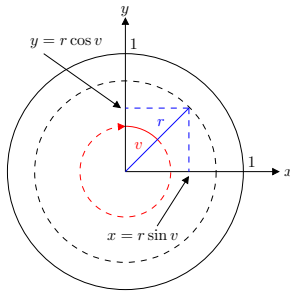
- The above equation tells us that the gradient vector  $\nabla f(x_0, y_0, z_0)$  is orthogonal to the tangent vector  $g'(t_0)$  to any curve  $g(t)$  that passes through  $P_0$  on the surface.
- Therefore,  $\nabla f(x_0, y_0, z_0)$  is orthogonal to the surface at point  $(x_0, y_0, z_0)$ .
- On a curve surface, we can use the gradient vector at a point as the normal vector.

# Normal Vectors

- A unit sphere can also be represented in parametric form  $f(\theta, \phi)$



Side View



Front View

where

$$x = x(u, v) = r \sin v = \cos u \sin v$$

$$y = y(u, v) = r \cos v = \cos u \cos v$$

$$z = z(u, v) = \sin u$$

where  $-\pi/2 < u < \pi/2$  and  $-\pi < v < \pi$ .

# Normal Vectors

- At point  $\mathbf{p}(u, v)$  the normal vector can be calculated by

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

where

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

- In our case

$$\frac{\partial \mathbf{p}}{\partial u} = \begin{bmatrix} -\sin u \sin v \\ -\sin u \cos v \\ \cos u \end{bmatrix} \quad \text{and} \quad \frac{\partial \mathbf{p}}{\partial v} = \begin{bmatrix} \cos u \cos v \\ -\cos u \sin v \\ 0 \end{bmatrix}$$

- Thus

$$\begin{aligned}\frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v} &= \begin{bmatrix} -\sin u \sin v \\ -\sin u \cos v \\ \cos u \end{bmatrix} \times \begin{bmatrix} \cos u \cos v \\ -\cos u \sin v \\ 0 \end{bmatrix} \\&= \begin{bmatrix} \cos u \cos u \sin v \\ \cos u \cos u \cos v \\ \sin u \sin v \cos u \sin v + \sin u \cos v \cos u \cos v \end{bmatrix} \\&= \cos u \begin{bmatrix} \cos u \sin v \\ \cos u \cos v \\ \sin u (\sin^2 v + \cos^2 v) \end{bmatrix} \\&= \cos u \begin{bmatrix} \cos u \sin v \\ \cos u \cos v \\ \sin u \end{bmatrix} = (\cos u) \mathbf{p}\end{aligned}$$

# Normal Vectors

- Recall that we generally draw a plane by drawing a triangle using three vertices  $\mathbf{p}_0$ ,  $\mathbf{p}_1$ , and  $\mathbf{p}_2$ .
- Do not forget the right-hand rule:
  - Orders  $\mathbf{p}_0 \rightarrow \mathbf{p}_1 \rightarrow \mathbf{p}_2$ ,  $\mathbf{p}_1 \rightarrow \mathbf{p}_2 \rightarrow \mathbf{p}_0$ , and  $\mathbf{p}_2 \rightarrow \mathbf{p}_0 \rightarrow \mathbf{p}_1$  give you the same triangle facing the same direction
  - Orders  $\mathbf{p}_0 \rightarrow \mathbf{p}_2 \rightarrow \mathbf{p}_1$ ,  $\mathbf{p}_2 \rightarrow \mathbf{p}_1 \rightarrow \mathbf{p}_0$ , and  $\mathbf{p}_1 \rightarrow \mathbf{p}_0 \rightarrow \mathbf{p}_2$  give you the same triangle as in previous three orders but facing in the opposite direction
- Suppose a triangle is defined by three vertices  $\mathbf{p}_0$ ,  $\mathbf{p}_1$ , and  $\mathbf{p}_2$  and it is drawn in that order, the plane normal can be found using their cross product:

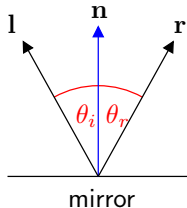
$$\mathbf{n} = \frac{(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)}{|(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)|}$$

- This is a normal vector of a flat surface created by a triangle



# Angle of Reflection

- To calculate the angle of reflection or the direction of reflection ( $\mathbf{r}$ ) of a point, we need
  - the normal at the point ( $\mathbf{n}$ ), and
  - the direction of the light source ( $\mathbf{l}$ ).
- There are conditions that they must satisfy:
  - The angle between  $\mathbf{l}$  and  $\mathbf{n}$  ( $\theta_i$ ) must be the same as the angle between  $\mathbf{n}$  and  $\mathbf{r}$  ( $\theta_r$ )
  - $\mathbf{r}$ ,  $\mathbf{n}$ , and  $\mathbf{l}$  must lie in the same plane



- Note that there are two possible  $\mathbf{r}$ :
  - if  $\mathbf{r} = \mathbf{l}$ , it also satisfies above condition but it is not what we want.

# Angle of Reflection

- For simplicity, assume that  $\mathbf{l}$ ,  $\mathbf{n}$ , and  $\mathbf{r}$  are unit vector
- According to dot product
  - $\mathbf{l} \cdot \mathbf{n} = |\mathbf{l}||\mathbf{n}| \cos \theta_i = \cos \theta_i$
  - $\mathbf{n} \cdot \mathbf{r} = |\mathbf{n}||\mathbf{r}| \cos \theta_r = \cos \theta_r$
- Since  $\theta_i = \theta_r$ , we have  $\mathbf{l} \cdot \mathbf{n} = \mathbf{n} \cdot \mathbf{r}$ .
- Since  $\mathbf{r}$  lies in the same plane as  $\mathbf{l}$  and  $\mathbf{n}$ , we can express  $\mathbf{r}$  as a linear combination of  $\mathbf{l}$  and  $\mathbf{n}$  as

$$\mathbf{r} = \alpha \mathbf{l} + \beta \mathbf{n}$$

$$\mathbf{n} \cdot \mathbf{r} = (\alpha \mathbf{l} + \beta \mathbf{n}) \cdot \mathbf{n}$$

$$\mathbf{n} \cdot \mathbf{r} = \alpha \mathbf{l} \cdot \mathbf{n} + \beta \mathbf{n} \cdot \mathbf{n}$$

$$\mathbf{n} \cdot \mathbf{r} = \alpha \mathbf{l} \cdot \mathbf{n} + \beta$$

- Since  $\mathbf{l} \cdot \mathbf{n} = \mathbf{n} \cdot \mathbf{r}$ , we have

$$\mathbf{l} \cdot \mathbf{n} = \alpha \mathbf{l} \cdot \mathbf{n} + \beta$$

$$\mathbf{l} \cdot \mathbf{n} - \alpha \mathbf{l} \cdot \mathbf{n} = \beta$$

# Angle of Reflection

- $\mathbf{r}$  must be a unit length:

$$\begin{aligned}1 &= \mathbf{r} \cdot \mathbf{r} \\&= (\alpha \mathbf{l} + \beta \mathbf{n}) \cdot (\alpha \mathbf{l} + \beta \mathbf{n}) \\&= \alpha^2 \mathbf{l} \cdot \mathbf{l} + 2\alpha\beta \mathbf{l} \cdot \mathbf{n} + \beta^2 \mathbf{n} \cdot \mathbf{n} \\&= \alpha^2 + 2\alpha\beta \mathbf{l} \cdot \mathbf{n} + \beta^2\end{aligned}$$

- Substitute  $\beta$  by  $\mathbf{l} \cdot \mathbf{n} - \alpha \mathbf{l} \cdot \mathbf{n}$ , we have

$$\begin{aligned}1 &= \alpha^2 + 2\alpha(\mathbf{l} \cdot \mathbf{n} - \alpha \mathbf{l} \cdot \mathbf{n})\mathbf{l} \cdot \mathbf{n} + (\mathbf{l} \cdot \mathbf{n} - \alpha \mathbf{l} \cdot \mathbf{n})^2 \\1 &= \alpha^2 + 2\alpha(\mathbf{l} \cdot \mathbf{n})^2 - 2\alpha^2(\mathbf{l} \cdot \mathbf{n})^2 + (\mathbf{l} \cdot \mathbf{n})^2 - 2\alpha(\mathbf{l} \cdot \mathbf{n})^2 + \alpha^2(\mathbf{l} \cdot \mathbf{n})^2 \\1 &= \alpha^2 - \alpha^2(\mathbf{l} \cdot \mathbf{n})^2 + (\mathbf{l} \cdot \mathbf{n})^2 \\1 &= \alpha^2(1 - (\mathbf{l} \cdot \mathbf{n})^2) + (\mathbf{l} \cdot \mathbf{n})^2 \\1 - (\mathbf{l} \cdot \mathbf{n})^2 &= \alpha^2(1 - (\mathbf{l} \cdot \mathbf{n})^2) \\1 &= \alpha^2\end{aligned}$$

- Thus,  $\alpha = 1$  or  $\alpha = -1$ .

# Angle of Reflection

- If  $\alpha = 1$ , we have

$$\beta = \mathbf{l} \cdot \mathbf{n} - \alpha \mathbf{l} \cdot \mathbf{n} = \mathbf{l} \cdot \mathbf{n} - (1) \mathbf{l} \cdot \mathbf{n} = 0$$

- Thus, we have

$$\mathbf{r} = \alpha \mathbf{l} + \beta \mathbf{n} = (1) \mathbf{l} + (0) \mathbf{n} = \mathbf{l}$$

- The result satisfies all conditions but  $\mathbf{r}$  is not what we want
- If  $\alpha = -1$ , we have

$$\beta = \mathbf{l} \cdot \mathbf{n} - \alpha \mathbf{l} \cdot \mathbf{n} = \mathbf{l} \cdot \mathbf{n} - (-1) \mathbf{l} \cdot \mathbf{n} = 2 \mathbf{l} \cdot \mathbf{n}$$

- In this case, we have

$$\begin{aligned} \mathbf{r} &= \alpha \mathbf{l} + \beta \mathbf{n} = (-1) \mathbf{l} + (2 \mathbf{l} \cdot \mathbf{n}) \mathbf{n} \\ &= 2(\mathbf{l} \cdot \mathbf{n}) \mathbf{n} - \mathbf{l} \end{aligned}$$

# Specifying Lighting Parameter

- There are four type of light sources:
  - Ambient
  - Point
  - Spotlight
  - Distance
- Note that a spotlight and a distance light sources are simply a point source
  - Spotlight: In stead of emitting light in all direction, simply limit the direction
  - Distance: Simply move the point source to infinite so the position becomes vector (direction)

# Specifying Lighting Parameter

- For every light source, we need color and either location or direction
- Each light source need three components, ambient, diffuse, and specular:

```
vec4 light_ambient = {0.1, 0.1, 0.1, 1.0};  
vec4 light_diffuse = {1.0, 1.0, 1.0, 1.0};  
vec4 light_specular = {1.0, 1.0, 1.0, 1.0};
```

- For a point source, the position is a point:

```
vec4 light_position = {1.0, 2.0, 3.0, 1.0};
```

- For a distance source, the position is a direction (vector):

```
vec4 light_position = {1.0, 2.0, 3.0, 0.0};
```

# Specifying Lighting Parameter

- We also need distance-attenuation model

$$f(d) = \frac{1}{a + bd + cd^2}$$

which can be defined by three floating-point values:

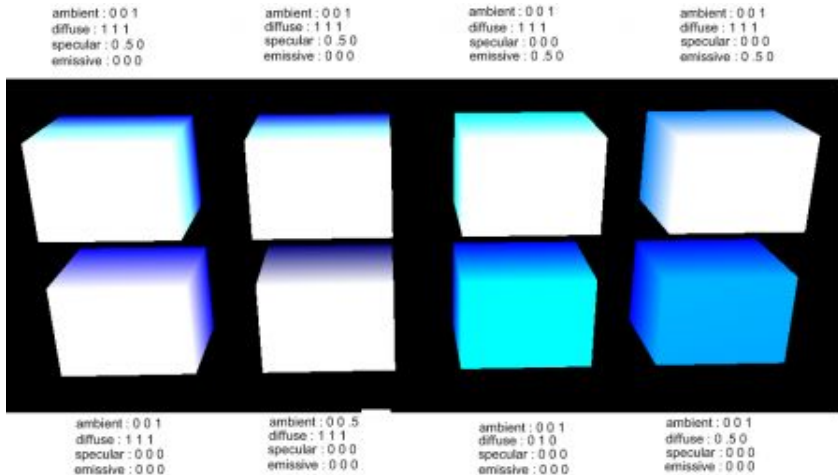
```
GLfloat attenuation_constant;  
GLfloat attenuation_linear;  
GLfloat attenuation_quadratic;
```

- **Ambient** color is the color of an object where it is in shadow. This color is what the object reflect when illuminated by ambient light rather than direct light
- **Diffuse** color is the color of the object under pure white light
- **Specular** color is the color of the light of a specular reflection
- **Emissive** color is the self-illumination color of an object



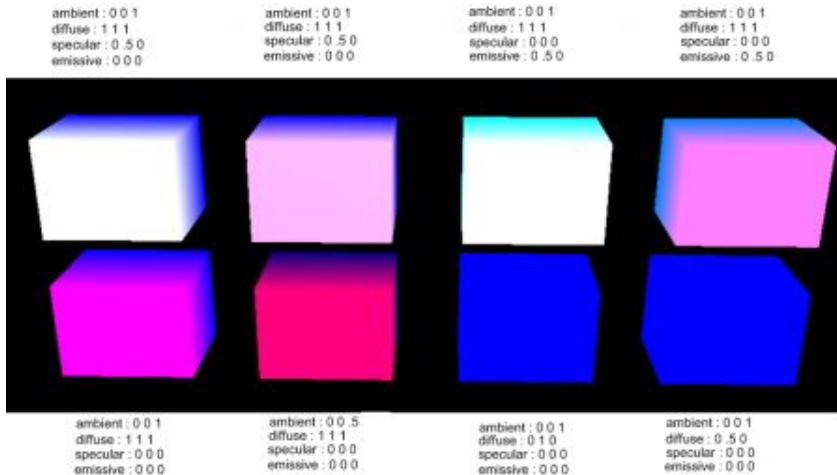
# Material

- Light Parameters: Ambient color is white and diffuse color is also white



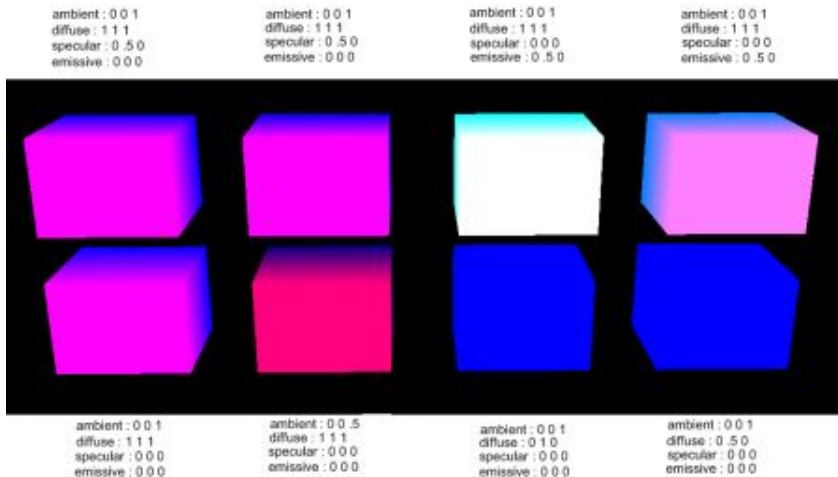
# Material

- Light Parameters: Ambient color is red and diffuse color is white



# Material

- Light Parameters: Ambient color is red and diffuse color is also red



# Materials

- For each material, it contains ambient, diffuse, and specular reflectivity coefficient for each primary color
- For example,

```
vec4 reflect_ambient = {0.2, 0.2, 0.2, 1.0};  
vec4 reflect_diffuse = {1.0, 0.8, 0.0, 1.0};  
vec4 reflect_specular = {1.0, 1.0, 1.0, 1.0};
```

Small amount of white ambient, yellow diffuse, and white specular.

- We also need to specify a shininess for specular

```
GLfloat shininess = ...;
```

- If the reflectivity properties of front and back are different, simply specify for the back

```
vec4 back_ambient, back_diffuse, back_specular;
```

# Materials

- You may have various object and each object has different materials
- It is a good idea to create a structure for each material

```
typedef struct
{
    vec4 reflect_ambient;
    vec4 reflect_diffuse;
    vec4 reflect_specular;
    vec4 emission;
    float shininess;
} material;
```

- This will allow us to define reflection properties of each material

```
materialStruct brassMaterials = {
    {0.33, 0.22, 0.03, 1.0},    // ambient
    {0.78, 0.57, 0.11, 1.0},    // diffuse
    {0.99, 0.91, 0.81, 1.0},    // specular
    {0.0, 0.0, 0.0, 1.0},       // emission
    27.8                         // shininess
};
```

# C Programming Trick

- Suppose you have 5 different materials, table, ball, chair, door, and carpet.
  - You may want to create five variables:

```
material table_material, ball_material, chair_material,  
        door_material, carpet_material;
```

But you cannot use a loop to iterate through material

- You may want to use an array of materials

```
material materials[5];
```

But you cannot refer to each material by a familiar name

# C Programming Trick

- In C, you can use a combination of enum type and array

```
typedef enum material_name
    {TABLE, BALL, CHAIR, DOOR, CARPET, NUM_MATERIALS};

material materials[NUM_MATERIALS];

materials[DOOR].reflect_ambient = ...;

for(i = 0; i < NUM_MATERIALS; i++)
{
    materials[i]....
}
```

- If you have a new material, simply insert a new name in the enum statement before NUM\_MATERIALS

# Applying the Lighting Model

- Lighting model can be applied in various stages:
  - Application
  - Vertex Shader
  - Fragment Shader
- Each stage has its own advantages and drawbacks



# Applying the Lighting Model in the Application

- What we have done so far, we generate an array of vertices and an array of colors
- Each vertex has its own specific color
- If two vertices of the same triangle have different colors, the fragment shader interpolate the color of each fragment for us
- To apply the lighting model in the application:
  - Each vertex is assigned a color as usual
  - However, the color of each vertex comes from lighting model instead of predefined color

# Applying the Lighting Model in the Application

- First, we need to define the light source (let's create a single point light source)

```
vec4 light_ambient = {L_ra, L_ga, L_ba, 1.0};  
vec4 light_diffuse = {L_rd, L_gd, L_bd, 1.0};  
vec4 light_specular = {L_rs, L_gs, L_bs, 1.0};  
vec4 light_position = ...;
```

- Assume we have only one single material:

```
vec4 reflect_ambient = {k_ra, k_ga, k_ba, 1.0};  
vec4 reflect_diffuse = {k_rd, k_gd, k_bd, 1.0};  
vec4 reflect_specular = {k_rs, k_gs, k_bs, 1.0};  
GLfloat shininess = ...;
```

- Suppose the color of  $i$ th vertex is stored in `colors[i]`, what we need to do is to assign a value to `colors[i]` based on lighting model

# Applying the Lighting Model in the Application

- For simplicity, create three variables of `vec4` for ambient, diffuse, and specular
- Suppose we use the normal of a triangle (from three vertices)

```
vec4 ambient, diffuse, specular;

for(i = 0; i < num_vertices; i = i + 3)
{
    vec4 p0 = vertices[i];
    vec4 p1 = vertices[i + 1];
    vec4 p2 = vertices[i + 2];

    // Calculate normal for all three vertices

    // Calculate color for the first vertex

    colors[i] = ambient + diffuse + specular;
    colors[i].w = 1.0;

    // Calculate color for the second vertex

    colors[i + 1] = ambient + diffuse + specular;
    colors[i + 1].w = 1.0;

    // Calculate color for the third vertex

    colors[i + 2] = ambient + diffuse + specular;
    colors[i + 2].w = 1.0;
}
```

# Applying the Lighting Model in the Application

- To calculate values of variables ambient, diffuse, and specular, we need to define a special **product** operation:

```
vec4 product(vec4 u, vec4 v)
{
    vec4 result;
    result.x = u.x * v.x;
    result.y = u.y * v.y;
    result.z = u.z * v.z;
    result.w = u.w * v.w;
}
```

- Note that is is not a matrix multiplication
- This will allow use to calculate all red, green, and blue, component in one function

# Applying the Lighting Model in the Application

- Recall the ambient for each color are as follows:

$$I_{ra} = k_{ra} L_{ra}$$

$$I_{ga} = k_{ga} L_{ga}$$

$$I_{ba} = k_{ba} L_{ba}$$

- With the `product()` function defined previously, we can simply use

```
ambient = product(light_ambient, reflect_ambient);
```

- Now the ambient becomes

```
{k_ra * L_ra, k_ga * L_ga, k_ba * L_ba, 1.0}
```

# Applying the Lighting Model in the Application

- Recall the diffuse reflection:

$$I_d = \frac{k_d}{a + bd + cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_d$$

- Thus, we need the normal of each vertex
- Each triangle consists of three vertices,  $p_0$ ,  $p_1$ ,  $p_2$
- All of these vertices create a flat plane (same normal vectors)
- Recall that  $p_1 - p_0$  and  $p_2 - p_1$  are vectors
- A vector perpendicular with two vectors can be calculated by their cross product
- Note that **the order matter**

```
vec4 n = normalize(cross(p1 - p0, p2 - p1));
```

Replace `normalize`, `cross`, and `-` with your functions

# Applying the Lighting Model in the Application

- If the light is a distance light source, the vector  $\mathbf{l}$  are the same for every vertex

```
vec4 l = light_position - {0.0, 0.0, 0.0, 1.0};  
vec4 diffuse = product(light_diffuse, reflect_diffuse) *  
                dot(n, normalize(l));
```

- For a finite light source  $\mathbf{l}$  depends on the position of light source and the vertex of interest:

```
vec4 l = light_position - vertex_position;  
vec4 diffuse = product(light_diffuse, reflect_diffuse) *  
                dot(n, normalize(l));
```

- Recall that we need to use  $\max(\mathbf{l} \cdot \mathbf{n}, 0)$ , thus

```
vec4 diffuse = {0.0, 0.0, 0.0, 1.0};  
GLfloat d = dot(n, normalize(l));  
if(d > 0.0)  
    diffuse = product(light_diffuse, reflect_diffuse) * d;
```

- Note that we did not include the term  $\frac{1}{a + bd + cd^2}$  yet

# Applying the Lighting Model in the Application

- Suppose all three vertices of a triangle have the same normal:
  - For a distance light source:
    - $l$  are the same for all three vertices
    - $l \cdot n$  are the same for all three vertices
    - Interpolate and constant diffuse shading are the same
    - Only need one calculation for all three vertices
  - For a finite light source:
    - $l$  are not the same for all three vertices
    - $l \cdot n$  are not the same
    - Results are different between interpolate and constant diffuse shading
    - Need one calculation for each vertex



# Applying the Lighting Model in the Application

- Recall the specular reflection term:

$$I_s = \frac{k_s}{a + bd + cd^2} L_s \max(\mathbf{r} \cdot \mathbf{v}, 0)^\alpha$$

- If we use modified Phong model, the specular reflection term becomes:

$$I_s = \frac{k_s}{a + bd + cd^2} L_s \max(\mathbf{n} \cdot \mathbf{h}, 0)^{\alpha'}$$

where  $\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{|\mathbf{l} + \mathbf{v}|}$

- First, create the half vector  $\mathbf{h}$  which requires vectors  $\mathbf{l}$  and  $\mathbf{v}$

# Applying the Lighting Model in the Application

- The vector  $\mathbf{l}$  are the same as in the diffuse reflection
- The vector  $\mathbf{v}$  is the vector from a vertex to the eye point at  $(0.0, 0.0, 0.0)$

```
vec4 v = normalize({0.0, 0.0, 0.0, 1.0} - vertex_position);
```

- Thus, our half vector  $\mathbf{h}$  becomes:

```
vec4 half = normalize(l + v);
```

- Finally, our specular reflection term becomes:

```
vec4 specular = {0.0, 0.0, 0.0, 1.0};  
GLfloat s = dot(half, n);  
if(s > 0.0)  
{  
    specular = pow(s, shininess) *  
               product(light_specular, material_specular);  
}
```

# Applying the Lighting Model in the Application

- We need to include attenuation factor for diffuse and specular

$$f(d) = \frac{1}{a + bd + cd^2}$$

where  $d$  is the distance from the light source to a vertex of interest

```
GLfloat attenuation_constant;  
GLfloat attenuation_linear;  
GLfloat attenuation_quadratic;  
:  
GLfloat attenuation(GLfloat d)  
{  
    return 1 / (attenuation_constant + (attenuation_linear * d) +  
                (attenuation_quadratic * d * d));  
}
```

- In our case,

```
d = magnitude(light_position - vertex_position);
```

- Suppose we want to rotate objects with lighting effect
  - Vectors  $\mathbf{l}$  and  $\mathbf{v}$  of a vertex change
  - Color of each vertex must be recalculated every time we rotate
  - Perform lighting effect in application requires the application to send the array of colors to the graphics pipeline every time we rotate
- Suppose we want to rotate an object in front of us where the light position is fixed
  - Vertices position are changed by the model view matrix in the vertex shader
- We can increase performance by performing the lighting model in the vertex shader

# Lighting in the Vertex Shader

- We can also generate color of each vertex in the vertex shader
- Let's start with the following vertex shader program:

```
#version 120

attribute vec4 vPosition;

uniform mat4 model_view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * model_view * vPosition;
}
```

- Note that we do not send the array of colors to the graphic pipeline in this case
  - The color of each vertex will be generated in the vertex shader itself

# Lighting in the Vertex Shader

- We are going to calculate the color of each vertex inside the vertex shader program and send it into the fragment shader

```
#version 120

attribute vec4 vPosition;
varying vec4 color;

uniform mat4 model_view;
uniform mat4 projection;
vec4 ambient, diffuse, specular;

void main()
{
    gl_Position = projection * model_view * vPosition;
    // calculate ambient, diffuse, and specular
    color = ambient + attenuation * (diffuse + specular);
}
```

where attenuation is the term  $\frac{1}{a+bd+cd^2}$  where  $d$  is the distance of the vertex to the light source

# Lighting in the Vertex Shader

- The products of light and reflection terms of a light source and a material only needed to be calculated once for all vertices of the material:
  - Recall that they are  $k_a L_a$ ,  $k_d L_d$ , and  $k_s L_s$
  - So, we can simply send all three products in as uniform variables

```
#version 120

attribute vec4 vPosition;
varying vec4 color;

uniform mat4 model_view;
uniform mat4 projection;
uniform vec4 ambient_product, diffuse_product, specular_product;
vec4 ambient, diffuse, specular;

void main()
{
    gl_Position = projection * model_view * vPosition;
    // calculate ambient, diffuse, and specular
    color = ambient + attenuation * (diffuse + specular);
}
```

# Lighting in the Vertex Shader

- We can send the array of normal vectors into the graphics pipeline in the same fashion as when we sent the array of colors or vertices

```
#version 120

attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec4 color;

uniform mat4 model_view;
uniform mat4 projection;
vec4 ambient, diffuse, specular;
uniform vec4 ambient_product, diffuse_product, specular_product;

void main()
{
    gl_Position = projection * model_view * vPosition;
    // calculate ambient, diffuse, and specular
    color = ambient + attenuation * (diffuse + specular);
}
```



# Lighting in the Vertex Shader

- We also need the position of the light source which can be sent as a uniform variable

```
#version 120

attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec4 color;

uniform mat4 model_view;
uniform mat4 projection;
vec4 ambient, diffuse, specular;
uniform vec4 ambient_product, diffuse_product, specular_product;
uniform vec4 light_position;

void main()
{
    gl_Position = projection * model_view * vPosition;
    // calculate ambient, diffuse, and specular
    color = ambient + attenuation * (diffuse + specular);
}
```

# Lighting in the Vertex Shader

- In case of ambient, it is straightforward:
  - Intensity of color does not depend on location of light, vertex's normal, or viewer position

```
ambient = ambient_product;
```

- For the diffuse and specular, we need to use the normal of each vertex
- **Note** that vertices will be transformed by the model view and projection matrices but what about the light source position?
  - If the light source is fixed based on the camera frame, no need to transform
  - If the light source is fixed based on the object (model) frame, it must be transformed as well

# Lighting in the Vertex Shader

- The normal of each vertex is also transformed by the model view and projection matrices
- Thus, the vector  $\mathbf{n}$  can be calculated by

```
vec4 N = normalize(projection * model_view * vNormal);
```

- A `vNormal` should already be normalized by the application (but not necessary)
- We need to normalize because the `projection` matrix contains shear
  - Rotate and translate will not change the magnitude of a vector
  - But shear does
- The `normalize()` function is a predefined function in shader programs

# Lighting in the Vertex Shader

- Thus, the diffuse term can be calculated by

```
// Light source position is fixed to the object frame
vec4 L_temp = projection * model_view * (light_position - vPosition);
vec4 L = normalize(L_temp);
diffuse = max(dot(L, N), 0.0) * diffuse_product;
```

or

```
// Light source position is fixed to the camera frame
vec4 L_temp = light_position - (projection * model_view * vPosition);
vec4 L = normalize(L_temp);
diffuse = max(dot(L, N), 0.0) * diffuse_product;
```

- The `max()` and `dot()` are predefined functions
- The variable `L_temp` will also be used to calculate attenuation
  - The magnitude of `L_temp` is the distance from the vertex to the light source

# Lighting in the Vertex Shader

- For the specular, we also need to know the shininess factor which can be sent as uniform variables

```
uniform float Shininess
```

- Note that the viewer can be anywhere in the object (world) frame
  - The position is the eye point from model view
  - But after applying the model view matrix, the eye point become the origin
  - The the viewer position is always at origin (0.0, 0.0, 0.0) according to the camera frame

```
vec4 eye_position = vec4(0.0, 0.0, 0.0, 1.0);
```

- `vec4()` is a predefined function

# Lighting in the Vertex Shader

- Now, the specular can be calculated by

```
vec4 V = normalize(eye_position - vPosition);  
vec4 H = normalize(L + V);  
specular = pow(max(dot(N, H), 0.0), Shininess) * specular_product;
```

- The `pow()` function is a predefined function
- The last thing we need is the attenuation
- All three attenuation constants can be sent as uniform variables

```
uniform float attenuation_constant, attenuation_linear,  
             attenuation_quadratic;
```

# Lighting in the Vertex Shader

- The distance is the magnitude of the vector from the vertex position to the light source

```
// Light source position is fixed to the object frame
vec4 L_temp = projection * model_view * (light_position - vPosition);
:
float distance = length(L_temp);
```

or

```
// Light source position is fixed to the camera frame
vec4 L_temp = light_position - (projection * model_view * vPosition);
:
float distance = length(L_temp);
```

- `length()` is another predefined function
- Therefore, the attenuation factor can be calculated as

```
float attenuation = 1/(attenuation_constant +
                      (attenuation_linear * distance) +
                      (attenuation_quadratic * distance * distance));
```

# Lighting in the Vertex Shader

- Finally, our vertex shader (light position is fixed with object frame) becomes:

```
#version 120

attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec4 color;
uniform mat4 model_view, projection;
uniform vec4 ambient_product, diffuse_product, specular_product, light_position;
uniform float shininess, attenuation_constant, attenuation_linear, attenuation_quadratic;
vec4 ambient, diffuse, specular;

void main()
{
    ambient = ambient_product;
    vec4 N = normalize(projection * model_view * vNormal);
    vec4 L_temp = projection * model_view * (light_position - vPosition);
    vec4 L = normalize(L_temp);
    diffuse = max(dot(L,N), 0.0) * diffuse_product;
    vec4 eye_point = vec4(0.0, 0.0, 0.0, 1.0);
    vec4 V = normalize(eye_position - (projection * model_view * vPosition));
    vec4 H = normalize(L + V);
    specular = pow(max(dot(N, H), 0.0), shininess) * specular_product;
    float distance = length(L_temp);
    float attenuation = 1/(attenuation_constant + (attenuation_linear * distance) +
                           (attenuation_quadratic * distance * distance));
    color = ambient + (attenuation * (diffuse + specular));

    gl_Position = projection * model_view * vPosition;
}
```



# Lighting in the Vertex Shader

- For this method, the fragment shader remain unchanged

```
#version 120

varying vec4 color;

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

# Lighting in the Shader

- Currently we send a  $4 \times 4$  matrix (transformation, model view, and projection matrices):

```
GLuint an_m_location;
mat4 a_matrix = identity_matrix;
void init()
{
    :
    an_m_location = glGetUniformLocation(program, "m_name");
    :
}
void display()
{
    :
    glUniformMatrix4fv(an_m_location, 1, GL_FALSE,
                       (GLfloat *) &a_matrix);
    :
}
```

where

- `an_m_location` is a global variable of type `GLuint`
- `m_name` is a uniform variable of type `mat4` in a shader program
- `a_matrix` is a variable (most likely global) of type `mat4`

# Lighting in the Shader

- To send a vector (vec4) such as location, light parameters, etc:

```
GLuint a_v_location;
vec4 a_vec = a_position;
void init()
{
    :
    a_v_location = glGetUniformLocation(program, "v_name");
    :
}
void display()
{
    :
    glUniform4fv(a_v_location, 1, GL_FALSE, (GLfloat *) &a_vec);
    :
}
```

where

- `a_v_location` is a global variable of type `GLuint`,
- `v_name` is a uniform variable of type `vec4` in a shader program, and
- `a_vec` is a variable (most likely global) of type `vec4`.

# Lighting in the Shader

- To send a floating-point (shininess, attenuation, etc):

```
GLuint a_f_location;
float a_float = a_value;
void init()
{
    :
    a_f_location = glGetUniformLocation(program, "f_name");
    :
}
void display()
{
    :
    glUniform1fv(a_f_location, 1, GL_FALSE, (GLfloat *) &a_float);
    :
}
```

where

- `a_f_location` is a global variable of type `GLuint`,
- `f_name` is a uniform variable of type `float` in a shader program, and
- `a_float` is a variable (most likely global) of type `GLfloat`.

# Lighting in the Fragment Shader

- By using the vertex shader, we calculate color on a per-vertex basis
- We can also perform calculation on a per-fragment basis
- Recall that data are interpolated when they pass through from a vertex shader to a fragment shader via *varying* variables
- To calculate color in the fragment shader, following data should be interpolated:
  - Normals of vertices become normal of each fragment
  - The  $\mathbf{l}$  vectors of vertices become  $\mathbf{l}$  of each fragment
  - The  $\mathbf{v}$  vectors of vertices become  $\mathbf{v}$  of each fragment
  - Distances of vertices to the light source become distance of each fragment to the light source
- As usual, the application will send an array of normal vectors

# Lighting in the Fragment Shader

- Our vertex shader file should look like the following:

```
#version 120

attribute vec4 vPosition;
attribute vec4 vNormal;

uniform mat4 model_view, projection;
uniform mat4 light_position;

varying vec4 N;
varying vec4 L;
varying vec4 V;
varying float distance;

void main()
{
    gl_position = projection * model_view * vPosition;
    N = projection * model_view * vNormal;
    L = light_position - (projection * model_view * vPosition);
    vec4 eye_point = vec4(0.0, 0.0, 0.0, 1.0);
    V = eye_point - (projection * model_view * vPosition);
    distance = length(L);
}
```

# Lighting in the Fragment Shader

- Fragment Shader

```
#version 120

varying vec4 N;
varying vec4 L;
varying vec4 V;
varying float distance;

uniform mat4 ambient_product, diffuse_product, specular_product;
uniform float shininess;
uniform float attenuation_constant, attenuation_linear, attenuation_quadratic;

vec4 ambient, diffuse, specular;

void main()
{
    vec4 NN = normalize(N);
    vec4 VV = normalize(V);
    vec4 LL = normalize(L);
    ambient = ambient_product;
    vec4 H = normalize(LL + VV);
    diffuse = max(dot(LL, NN), 0.0) * diffuse_product;
    specular = pow(max(dot(NN, H), 0.0), shininess) * specular_product;
    float attenuation = 1/(attenuation_constant + (attenuation_linear * distance) +
                           (attenuation_quadratic * distance * distance));
    gl_FragColor = ambient + attenuation * (diffuse + specular);
}
```