Lighting and Shading

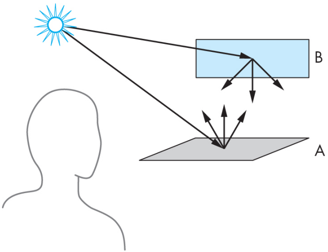
Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

From 2D to 3D

- 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

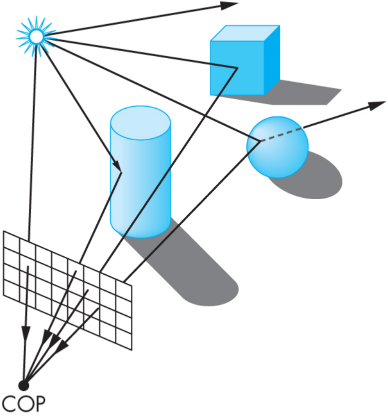
- We are going to render our image based on physics
- A surface can either emits light, reflect right, or both
- The color of a point of an object is a result of multiple interactions



- Light can reflex 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

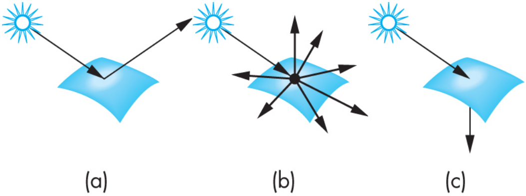
Light and Matter

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



Light-Material Interaction

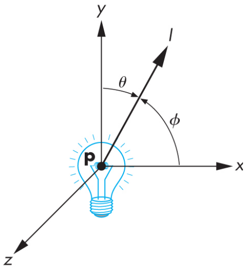
- Light interacts with materials differently:



(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

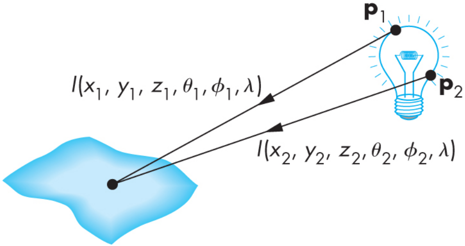
- A light source emits light
- Every point (x, y, z) on the surface of a light source has its own characteristic:
  - intensity of energy emitted at each wave length λ, and
  - direction of emission θ and ϕ



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

Light Sources

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



Color 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

$$\mathbf{I} = \begin{bmatrix} I_{\mathsf{r}} \\ I_{\mathsf{g}} \\ I_{\mathsf{b}} \end{bmatrix}$$

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:

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

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

$$\mathbf{I}(\mathbf{p}_0) = \begin{bmatrix} I_{\mathsf{r}}(\mathbf{p}_0) \\ I_{\mathsf{g}}(\mathbf{p}_0) \\ I_{\mathsf{b}}(\mathbf{p}_0) \end{bmatrix}$$

- 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

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

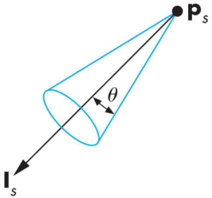
- For a softer effect, we use

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

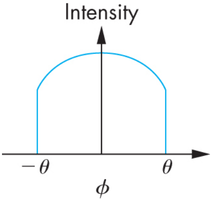
where d is the distance between $\mathbf{p}$ and $\mathbf{p}_0$

Spotlights

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



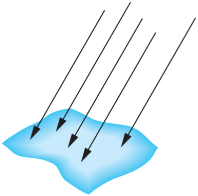
- Ideally, the intensity of a spotlight is 0 when a point is outside the cone:



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

Distance Light Sources

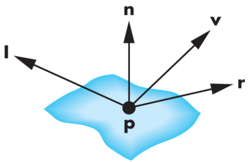
- The sun is an example of a distance light source
- Light from the sun hit all objects in the same direction
- For simplicity, we can use one vector to represent the direction of the light for every point on every object



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

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

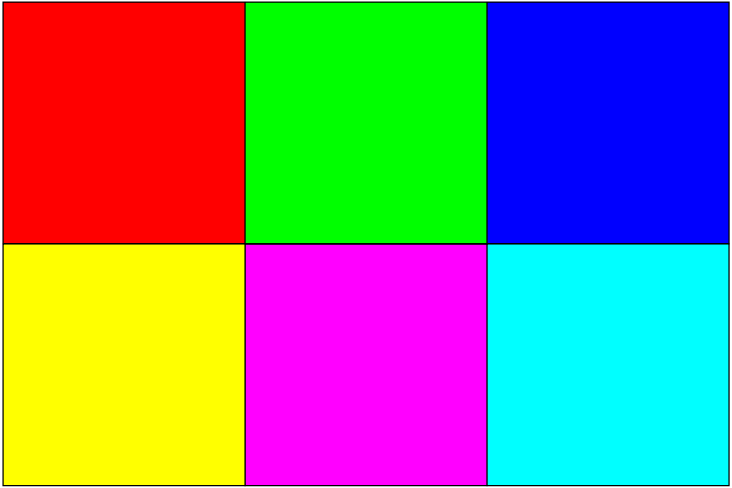
The Phong Reflection Model



- Uses four vectors to calculate a color for an arbitrary point $\mathbf{p}$ on a surface
  - $\mathbf{n}$ is the normal 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 light source
  - $\mathbf{r}$ is in the direction that a perfectly reflected ray from $\mathbf{l}$ 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 3×3 illumination matrix for each light source i:

$$\mathbf{L}_i = \begin{bmatrix} L_{i\mathsf{ra}} & L_{i\mathsf{ga}} & L_{i\mathsf{ba}} \\ L_{i\mathsf{rd}} & L_{i\mathsf{gd}} & L_{i\mathsf{bd}} \\ L_{i\mathsf{rs}} & L_{i\mathsf{gs}} & L_{i\mathsf{bs}} \end{bmatrix}$$

  - 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 causes 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 $L_{i\mathsf{x}}$ contributes to the intensity at a point $\mathbf{p}$ by $R_{i\mathsf{x}}L_{i\mathsf{x}}$.
- $R_{i\mathsf{x}}$ 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

$$\mathbf{R}_i = \begin{bmatrix} R_{i\mathsf{ra}} & R_{i\mathsf{ga}} & R_{i\mathsf{ba}} \\ R_{i\mathsf{rd}} & R_{i\mathsf{gd}} & R_{i\mathsf{bd}} \\ R_{i\mathsf{rs}} & R_{i\mathsf{gs}} & R_{i\mathsf{bs}} \end{bmatrix}$$

- Code

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

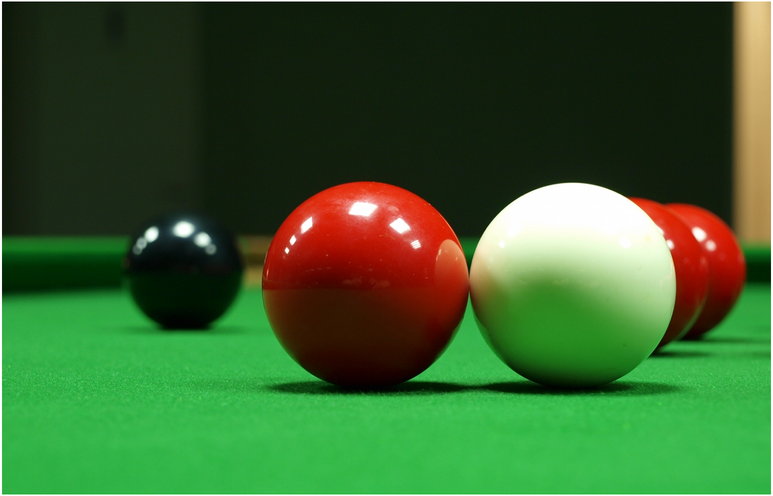
- Note that the reflective properties depend on the surface material

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

$$\begin{aligned} I_{i\mathsf{r}} &= R_{i\mathsf{ra}} L_{i\mathsf{ra}} + R_{i\mathsf{rd}} L_{i\mathsf{rd}} + R_{i\mathsf{rs}} L_{i\mathsf{rs}} \\ &= I_{i\mathsf{ra}} + I_{i\mathsf{rd}} + I_{i\mathsf{rs}} \end{aligned}$$

- The total red intensity (from all sources) is

$$I_{\mathsf{r}} = \sum_i (I_{i\mathsf{ra}} + I_{i\mathsf{rd}} + I_{i\mathsf{rs}}) + I_{\mathsf{ar}}$$

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

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

$$I_{\mathsf{g}} = \sum_i (I_{i\mathsf{ga}} + I_{i\mathsf{gd}} + I_{i\mathsf{gs}}) + I_{\mathsf{ag}}$$

$$I_{\mathsf{b}} = \sum_i (I_{i\mathsf{ba}} + I_{i\mathsf{gb}} + I_{i\mathsf{bs}}) + I_{\mathsf{ab}}$$

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

$$I = I_{\mathsf{a}} + I_{\mathsf{d}} + I_{\mathsf{s}} = L_{\mathsf{a}}R_{\mathsf{a}} + L_{\mathsf{d}}R_{\mathsf{d}} + L_{\mathsf{s}}R_{\mathsf{s}}$$

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

$$0 \le k_a \le 1$$

  - 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 $I_{\mathsf{a}} = k_{\mathsf{a}} L_{\mathsf{a}}$ which is a short notation for:

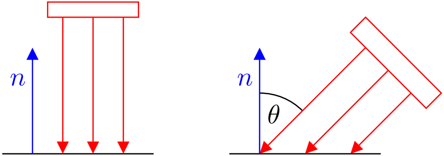
$$I_{\mathsf{ra}} = k_{\mathsf{ra}} L_{\mathsf{ra}} = \sum_i (k_{\mathsf{ira}} L_{\mathsf{ira}})$$

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

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

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

$$I_{\mathsf{d}} = k_{\mathsf{d}}(\mathbf{l} \cdot \mathbf{n}) L_{\mathsf{d}}$$

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

$$I_{\mathsf{d}} = \frac{k_{\mathsf{d}}}{a + bd + cd^2}(\mathbf{l} \cdot \mathbf{n}) L_{\mathsf{d}}$$

  where d is the distance and a, b, and c are constants.
- Note that $\mathbf{l} \cdot \mathbf{n}$ can be negative if the $\theta > 90°$ or $\theta < -90°$.
  - 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:

$$I_{\mathsf{d}} = \frac{k_{\mathsf{d}}}{a + bd + cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_{\mathsf{d}}$$

Diffuse Reflection

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

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

Specular Reflection

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

$$I_{\mathsf{s}} = k_{\mathsf{s}} L_{\mathsf{s}} \cos^{\alpha} \phi$$

  where
  - k_{s} is the coefficient $(0 \leq k_s \leq 1)$
  - ϕ is the angle between the reflection direction $\mathbf{r}$ and the direction of the viewer $\mathbf{v}$
  - α is a shininess coefficient (the larger the shinier)
    - ∞ 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:

$$I_{\mathsf{s}} = \frac{k_{\mathsf{s}}}{a + bd + cd^2} L_{\mathsf{s}} \max(\mathbf{r} \cdot \mathbf{v}, 0)^{\alpha}$$

Specular Reflection

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

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

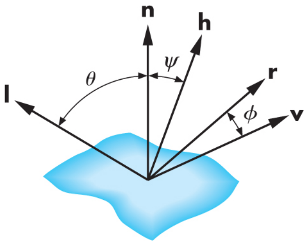
The Phong Reflection Model

- Recall that we have
  - $I_{\mathsf{a}} = k_{\mathsf{a}} L_{\mathsf{a}}$,
  - $I_{\mathsf{d}} = \frac{k_{\mathsf{d}}}{a+bd+cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_{\mathsf{d}}$, and
  - $I_{\mathsf{s}} = \frac{k_{\mathsf{s}}}{a+bd+cd^2} L_{\mathsf{s}} \max(\mathbf{r} \cdot \mathbf{v}, 0)^{\alpha}$.
- Finally we have the Phong reflection model as

$$I = \frac{1}{a+bd+cd^2}(k_{\mathsf{d}} L_{\mathsf{d}} \max(\mathbf{l} \cdot \mathbf{n}, 0) + K_{\mathsf{s}} L_{\mathsf{s}} \max(\mathbf{r} \cdot \mathbf{v}, 0)^{\alpha}) + k_{\mathsf{a}} L_{\mathsf{a}}$$

- 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 sightly different

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

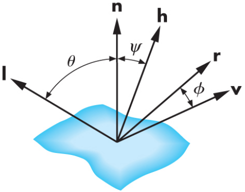


where

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

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

The Modified Phong Model



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

$$\begin{aligned} 2\theta + \phi &= 2(\theta + \psi) \\ &= 2\theta + 2\psi \\ \phi &= 2\psi \end{aligned}$$

The Modified Phong Model

- We can avoid calculating $\mathbf{r} \cot \mathbf{v}$ by simply use $\mathbf{n} \cdot \mathbf{h}$.
    - We do not need to calculate $\mathbf{r}$ since to calculate $\mathbf{n} \cot \mathbf{h}$, 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

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

  - 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, ever 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

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

Normal Vectors

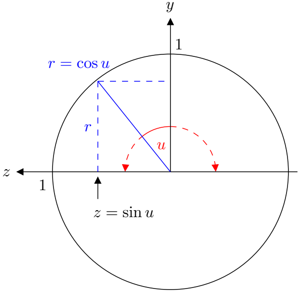
- At $t = t_0$, we have

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

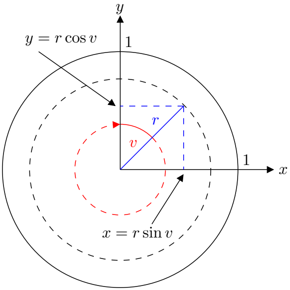
- The above equation tells us that the gradient vector $\nabla f(x_0, y_0, z_0)$ is orthogonal to the tangent vector $g'(t_0)$ to any curve $g(t)$ that passes through P_0 on the surface.
- Therefore, $\nabla f(x_0, y_0, z_0)$ is orthogonal to the surface at point (x_0, y_0, z_0).
- On a curve surface,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

$$
\begin{aligned}
x &= x(u, v) = r \sin v = \cos u \sin v \\
y &= y(u, v) = r \cos v = \cos u \cos v \\
z &= z(u, v) = \sin u
\end{aligned}
$$

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

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

Normal Vectors

- Thus

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

$$= \begin{bmatrix} \cos u \cos u \sin v \\ \cos u \cos u \cos v \\ \sin u \sin v \cos u \sin v + \sin u \cos v \cos u \cos v \end{bmatrix}$$

$$= \cos u \begin{bmatrix} \cos u \sin v \\ \cos u \cos v \\ \sin u (\sin^2 v + \cos^2 v) \end{bmatrix}$$

$$= \cos u \begin{bmatrix} \cos u \sin v \\ \cos u \cos v \\ \sin u \end{bmatrix} = (\cos u)\mathbf{p}$$

Normal Vectors

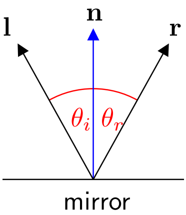
- Recall that we generally draw a plane by drawing a triangle using three vertices $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$.
- Do not forget the right-hand rule:
  - Orders $\mathbf{p}_0 \rightarrow \mathbf{p}_1 \rightarrow \mathbf{p}_2$, $\mathbf{p}_1 \rightarrow \mathbf{p}_2 \rightarrow \mathbf{p}_0$, 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:

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

- 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 $\mathbf{n}$ (θ_i) must be the same as the angle between $\mathbf{n}$ and $\mathbf{r}$ (θ_r)
  - $\mathbf{r}$, $\mathbf{n}$, and $\mathbf{l}$ must lie in the same pane



- 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 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

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

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

$$\begin{aligned}
\mathbf{l} \cdot \mathbf{n} &= \alpha\mathbf{l} \cdot \mathbf{n} + \beta \\
\mathbf{l} \cdot \mathbf{n} - \alpha\mathbf{l} \cdot \mathbf{n} &= \beta
\end{aligned}$$

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 β 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:

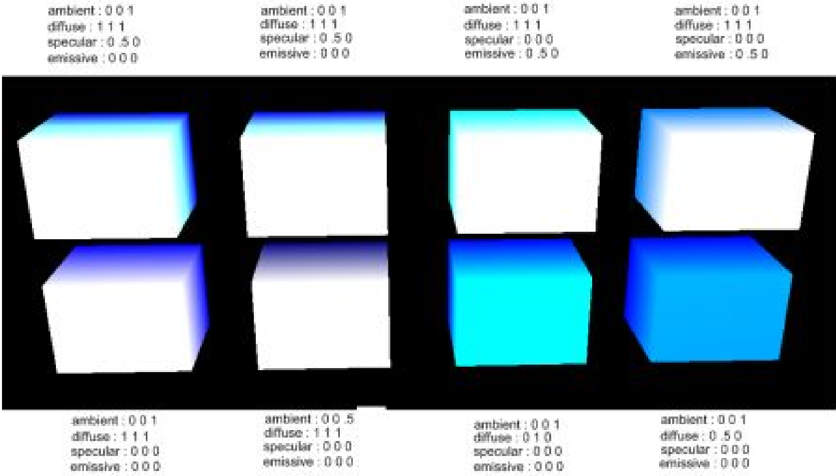
```
GLfloat attenuation_constant;
GLfloat attenuation_linear;
GLfloat attenuation_quadratic;
```

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
- **Emissive** color is the self-illumination color of an object

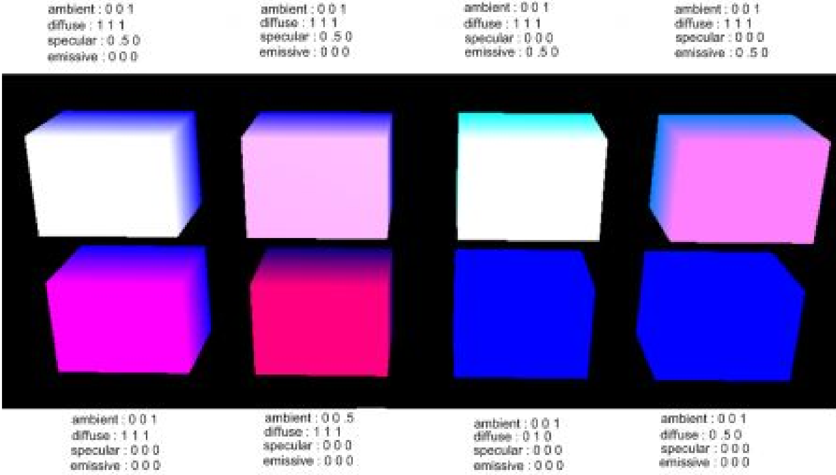
Material

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

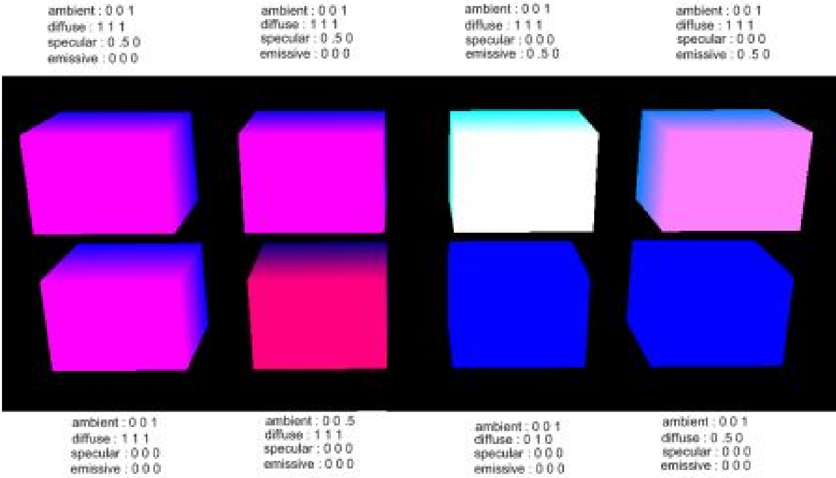


Material

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



- 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 it 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 relecct_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 ith 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

- Recall the ambient for each color are as follows:

$$I_{\mathsf{ra}} = k_{\mathsf{ra}} L_{\mathsf{ra}}$$
$$I_{\mathsf{ga}} = k_{\mathsf{ga}} L_{\mathsf{ga}}$$
$$I_{\mathsf{ba}} = k_{\mathsf{ba}} L_{\mathsf{ba}}$$

- 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:

$$I_{\mathsf{d}} = \frac{k_{\mathsf{d}}}{a + bd + cd^2} \max(\mathbf{l} \cdot \mathbf{n}, 0) L_{\mathsf{d}}$$

- Thus, we need the normal of each vertex
- Each triangle consists of three vertices, `p0`, `p1`, `p2`
- All of these vertices create a flat plane (same normal vectors)
- Recall that `p1 - p0` and `p2 - p1` 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 $\dfrac{1}{a + bd + cd^2}$ yet

Applying the Lighting Model in the Application

- Suppose all three vertices of a triangle have the same normal:
  - For a distance light source:
    - $\mathbf{l}$ are the same for all three vertices
    - $\mathbf{l} \cdot \mathbf{n}$ 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:
    - $\mathbf{l}$ are not the same for all three vertices
    - $\mathbf{l} \cdot \mathbf{n}$ 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:

$$I_{\mathsf{s}} = \frac{k_{\mathsf{s}}}{a + bd + cd^2} L_{\mathsf{s}} \max(\mathbf{r} \cdot \mathbf{v}, 0)^{\alpha}$$

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

$$I_{\mathsf{s}} = \frac{k_{\mathsf{s}}}{a + bd + cd^2} L_{\mathsf{s}} \max(\mathbf{n} \cdot \mathbf{h}, 0)^{\alpha'}$$

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

- 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);
```

Efficiency

- 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 130

in vec4 vPosition;
uniform mat4 ModelView;
uniform mat4 Porjection;
void main()
{
    gl_Position = Projection * ModelView * 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

```
out vec4 color
vec4 ambient, diffuse, 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
- The products of light and reflection terms needed to be calculated only once
  - $k_{\mathrm{a}}L_{\mathrm{a}}$, $k_{\mathrm{d}}L_{\mathrm{d}}$, and $k_{\mathrm{s}}L_{\mathrm{s}}$
  - So, we can simply send them all three products in as uniform variables

```
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
```

- In case of ambient, it is straightforward:

```
ambient = AmbientProduct;
```

Lighting in the Vertex Shader

- For the `diffuse` and `specular`, we need to know the normal of each vertex
- We can send the array of normal vectors into the graphics pipeline in the same fashion as when we sent the array of colors

```
in vec4 vNormal;
```

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

```
uniform vec4 LightPosition;
```

- **Note** that vertices will be transformed by the model view matrix 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 matrix
- Thus, the vector $\mathbf{n}$ can be calculcated by

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

- Thus, the `diffuse` term can be calculated by

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

or

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

- The variable `L_temp` will also be used to calculate attenuation

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 position is always at origin $(0.0, 0.0, 0.0)$ according to the camera frame

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

- Now, the `specular` can be calculated by

```
vec3 V = normalize(EyePosition - vPosition);
vec3 H = normalize(L + V);
specular = pow(max(dot(N, H), 0.0), Shininess) * SpecularProduct;
```

- The last thing we need is the attenuation

Lighting in the Vertex Shader

- All three attenuation constants can be sent as uniform variables

```
uniform float AttenuationConstant, AttenuationLinear,
              AttenuationQuadratic;
```

- 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 = model_view * (LightPosition - vPosition);
:
float distance = length(L_temp);
```

or

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

- Therefore, the attenuation factor can be calculated as

```
float attenuation = 1/(AttenuationConstant +
                       (AttenuationLinear * distance) +
                       (AttenuationQuadratic * distance * distance));
```

Lighting in the Vertex Shader

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

```
#version 130

in vec4 vPosition;
in vec4 vNormal;
out vec4 color;
uniform mat4 model_view, projection;
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct, LightPosition;
uniform float shininess, attenuation_constant, attenuation_linear, attenuation_quadratic;
vec4 ambient, diffuse, specular;

void main()
{
    ambient = AmbientProduct;
    vec4 N = normalize(model_view * vNormal);
    vec4 L_temp = model_view * (LightPosition - vPosition);
    vec4 L = normalize(L_temp);
    diffuse = max(dot(L,N), 0.0) * DiffuseProduct;
    vec4 EyePoint = vec4(0.0, 0.0, 0.0, 1.0);
    vec4 V = normalize(EyePoint - (model_view * vPosition));
    vec4 H = normalize(L + V);
    specular = pow(max(dot(N, H), 0.0), shininess) * SpecularProduct;
    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 Shader

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

```
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, and
- `a_matrix` is a variable (most likely global) of type `mat4`.

Lighting in the Shader

- To send a vector (e.g., location, light parameters, etc):

```
void init()
{
    :
    a_v_location = glGetUniformLocation(program, "v_name");
    :
}

void display()
{
    :
    glUniform4fv(an_v_location, 1, GL_FALSE, (GLfloat *) &a_vec);
    :
}
```

where

- `an_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):

```
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 vertex shader, we calculate color on a per-vectex basis
- We can also perform calculation on a per-fragment basis
- Recall that data are interpolate when they passing through from a vertex shader to a fragment shader
- The following data should be interpolated:
  - normal of each vertex
  - the $\mathbf{l}$ vector according to each vertexq
  - the $\mathbf{v}$ vector according to each vertex
- 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 130

in vec4 vPosition;
in vec4 vNormal;

uniform mat4 model_view, projection;
uniform mat4 light_position;

out vec4 N;
out vec4 L;
out vec4 E;
out float distance;

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

-

Lighting in the Fragment Shader

- Fragment Shader

```
in vec4 N;
in vec4 L;
in vec4 E;
in float distance;

uniform mat4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 model_view;
uniform float Shininess;
uniform float attenuation_constant, attenuation_linear, attenuation_quadratic;

vec4 ambient, diffuse, specular;

void main()
{
    vec4 NN = normalize(N);
    vec4 EE = normalize(E);
    vec4 LL = normalize(L);
    ambient = AmbientProduct;
    vec4 H = normalize(LL + EE);
    diffuse = max(dot(LL, NN), 0.0) * DiffuseProduct;
    specular = pow(max(dot(NN, H), 0.0), shininess) * SpecularProduct;
    float attenuation = 1/(attenuation_constant + (attenuation_linear * distance) +
                          (attenuation_quadratic * distance * distance));
    gl_FragColor = ambient + attenuation * (diffuse + specular);
}
```