SHADING I

Lecturer: Asst. Prof. Ufuk Çelikcan

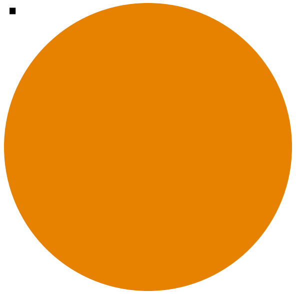
Based on the slides by: E. Angel and D. Shreiner

Objectives

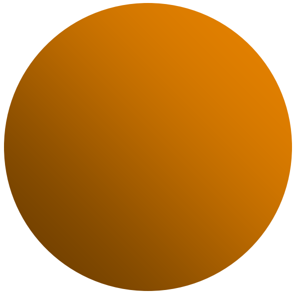
- Learn to shade objects so their images appear three-dimensional
- Introduce the types of light-material interactions
- Build a simple reflection model---the Phong model---that can be used with real time graphics hardware

Why we need shading

- Suppose we build a model of a sphere using many polygons and color it with a simple constant **`Color`**. We get something like :

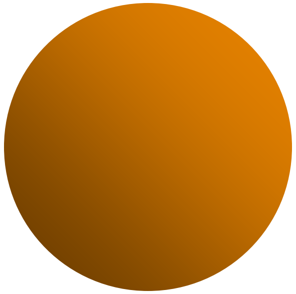


- But we want :



Shading

- Why does the image of a real sphere look like



- Light-material interactions cause each point to have a different color **(shade)**
- Need to consider
  - **Light sources**
  - **Material properties**
  - **Location of viewer**
  - **Surface orientation**

Scattering

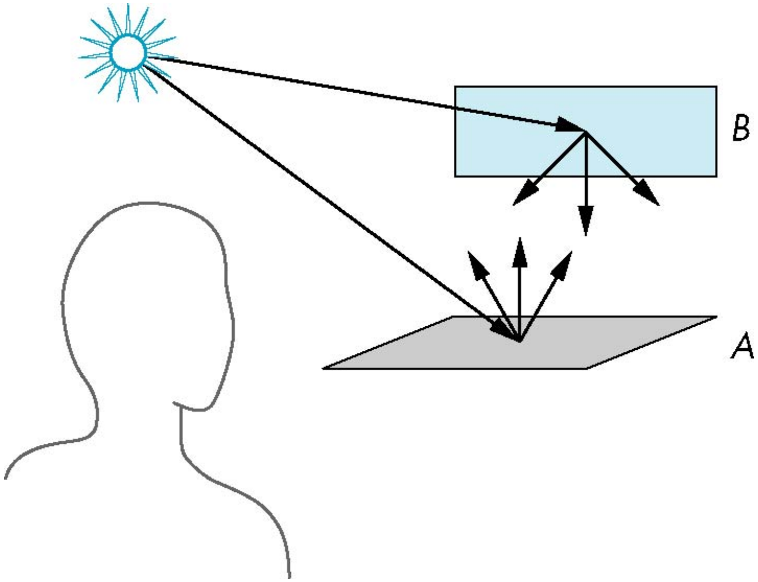
- Light strikes A
  - Some scattered
  - Some absorbed

- Some of scattered light strikes B
  - Some scattered
  - Some absorbed

- Some of this scattered

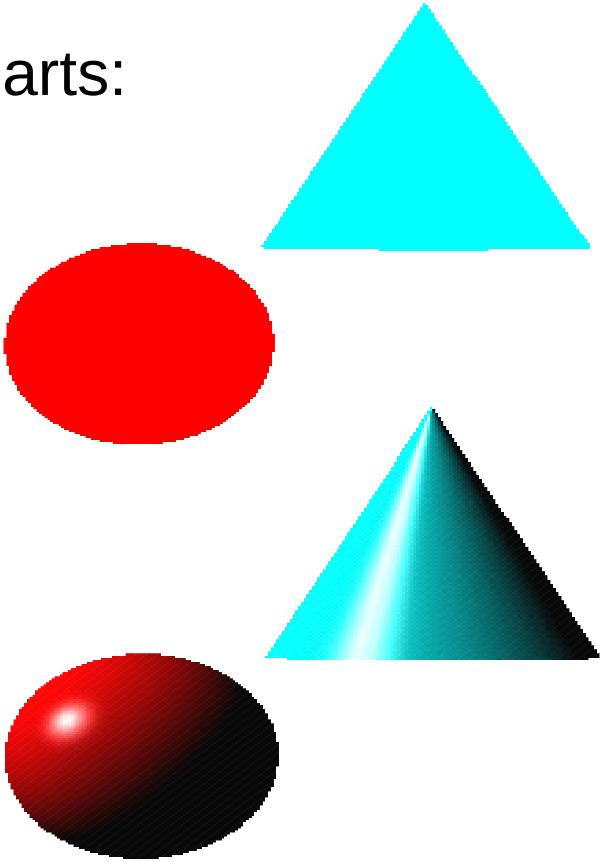
light strikes A

and so on



Lighting Principles

- Lighting simulates how objects reflect light
- Without lighting, objects tend to look like they are made out of plastic.
- OpenGL divides lighting into three parts:
  1. material composition of object
  2. light’s color and position
  3. global lighting parameters
- Can be implemented
  - in vertex shader for faster speed
  - in fragment shader for nicer shading

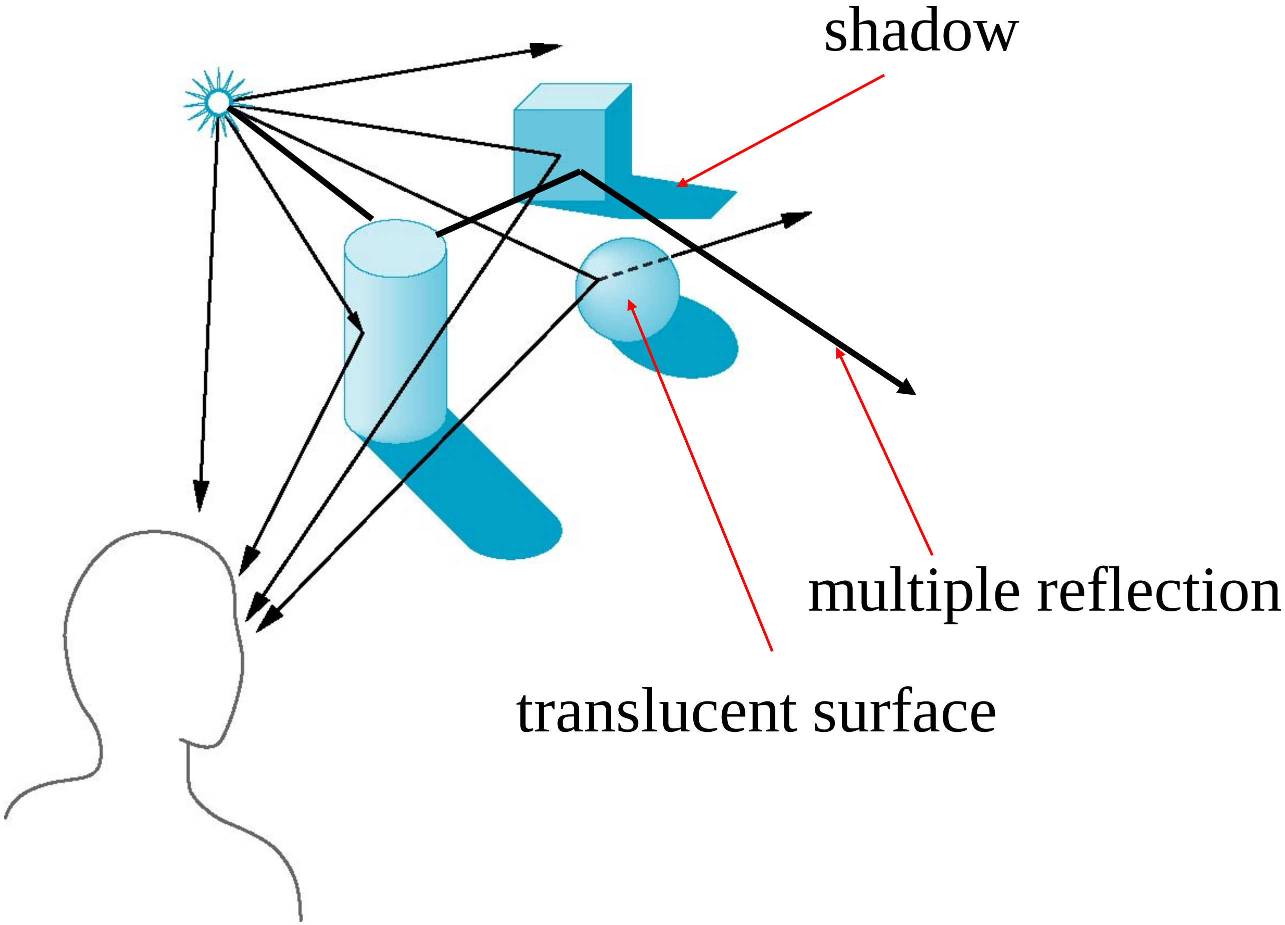


Rendering Equation

- The infinite scattering and absorption of light can be described by the *rendering equation*
  - In general, cannot be solved in finite time
  - Ray tracing is a special case for perfectly reflecting surfaces

- Rendering equation is global and includes
  - Shadows
  - Multiple scattering from object to object

Global Effects



shadow

multiple reflection

translucent surface

Local vs Global Rendering

- Correct shading requires a global calculation involving all objects and light sources
  - Incompatible with pipeline model which shades each polygon independently (local rendering)

- However, in computer graphics, especially in real time graphics, we are happy **if things "<u>look right</u>"**
  - many techniques exist for **approximating** global effects

Light-Material Interaction

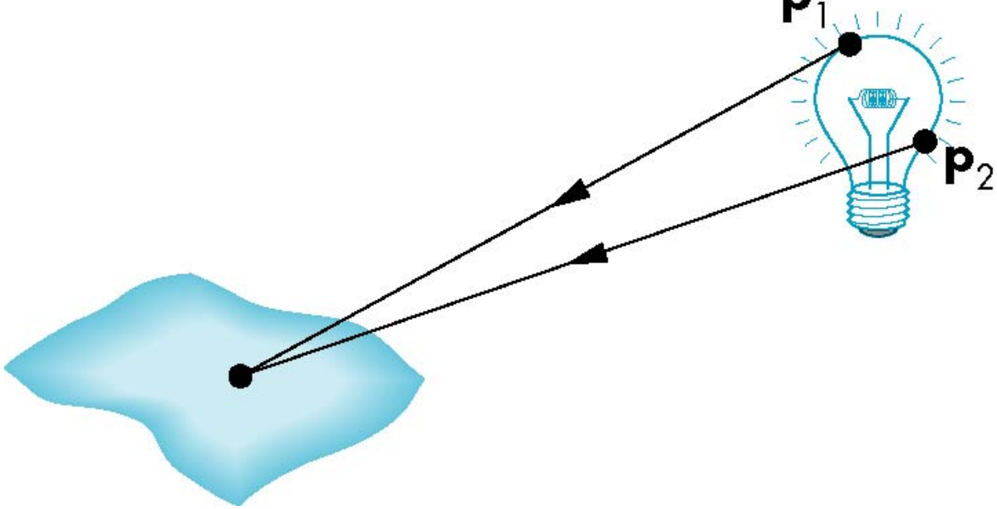
- Light that strikes an object is partially absorbed and partially scattered (reflected)

- The amount reflected determines the color and brightness of the object
  - A surface **appears red** under white light **because the red component of the light is reflected** and the rest is absorbed

- The reflected light is scattered in a manner that depends on the <u>smoothness and orientation</u> of the surface

Light Sources

General light sources are difficult to work with because we must integrate light coming from all points on the source



Simplified Light Sources

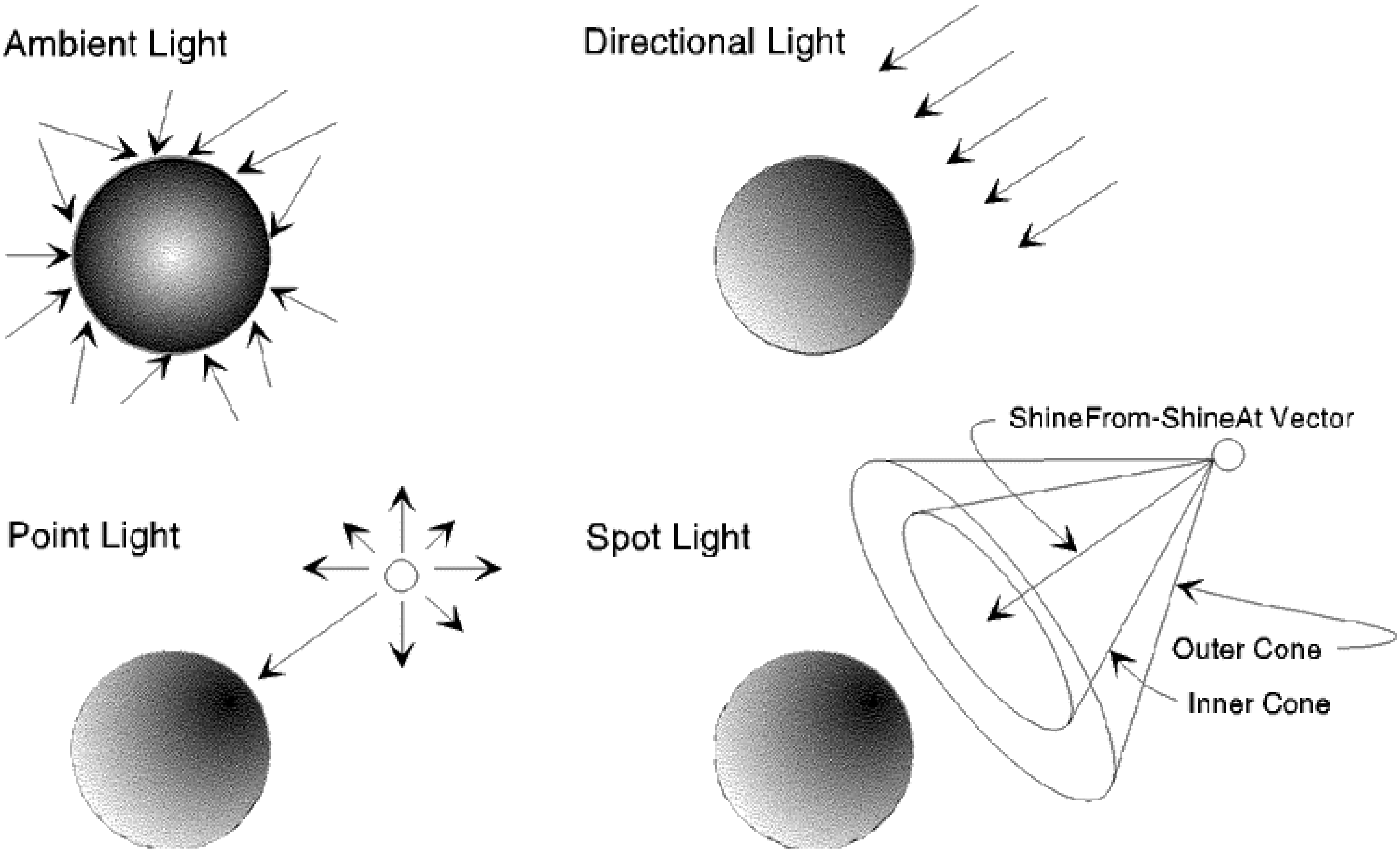
- **Point source**
  - Model with position and color

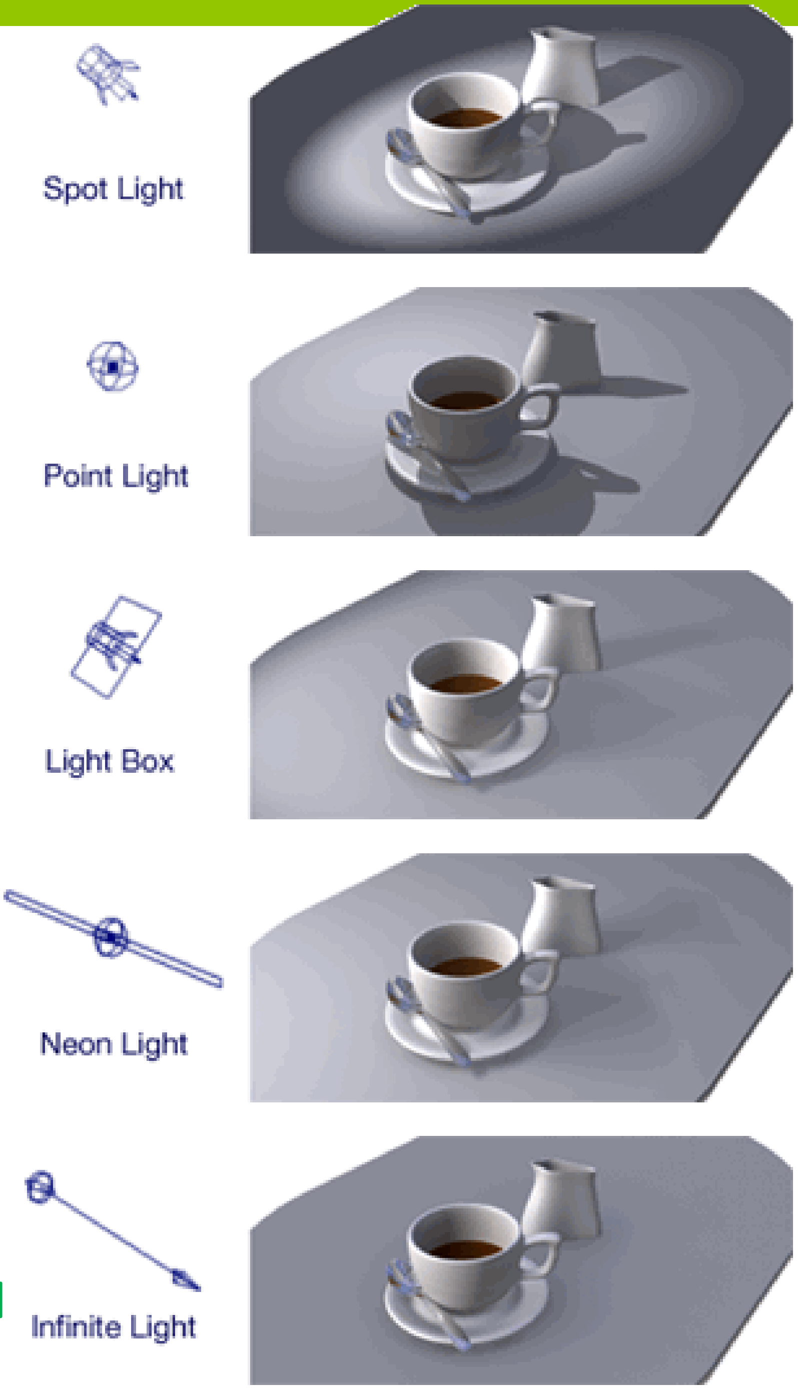
Directional light source

  - Very distant point light source ~ infinite distance away >> rays become parallel >> directional light source

- **Spotlight**
  - Restricts light from ideal point source to a certain range and area

- **Ambient light**
  - Same amount of light everywhere in the scene
  - No direction
  - Can model contribution of many sources and reflecting surfaces


Ambient Light
Directional Light
ShineFrom-ShineAt Vector
Point Light
Spot Light
Outer Cone
Inner Cone

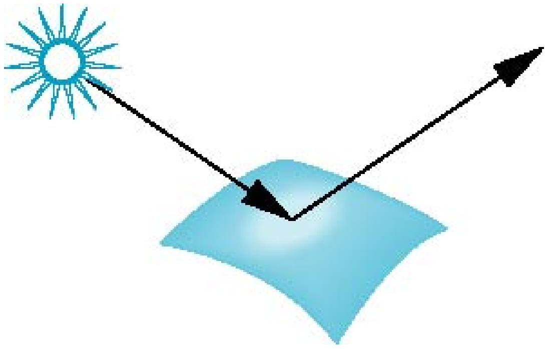


[Directional]

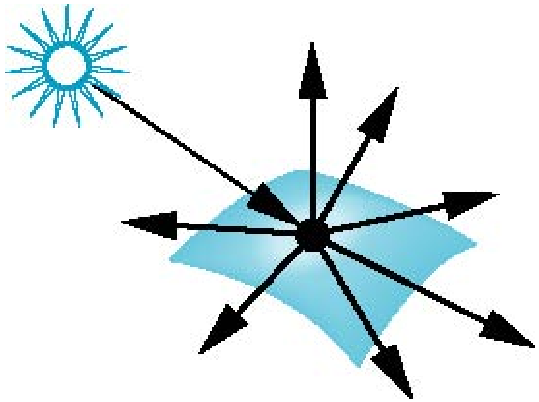
Surface Types

- The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflect the light

- A very rough surface scatters light in all directions



smooth surface



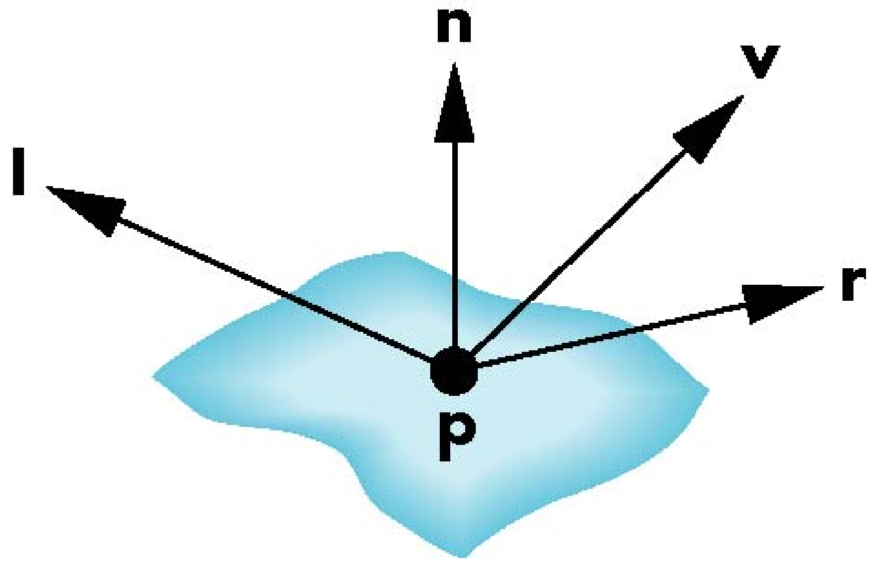
rough surface

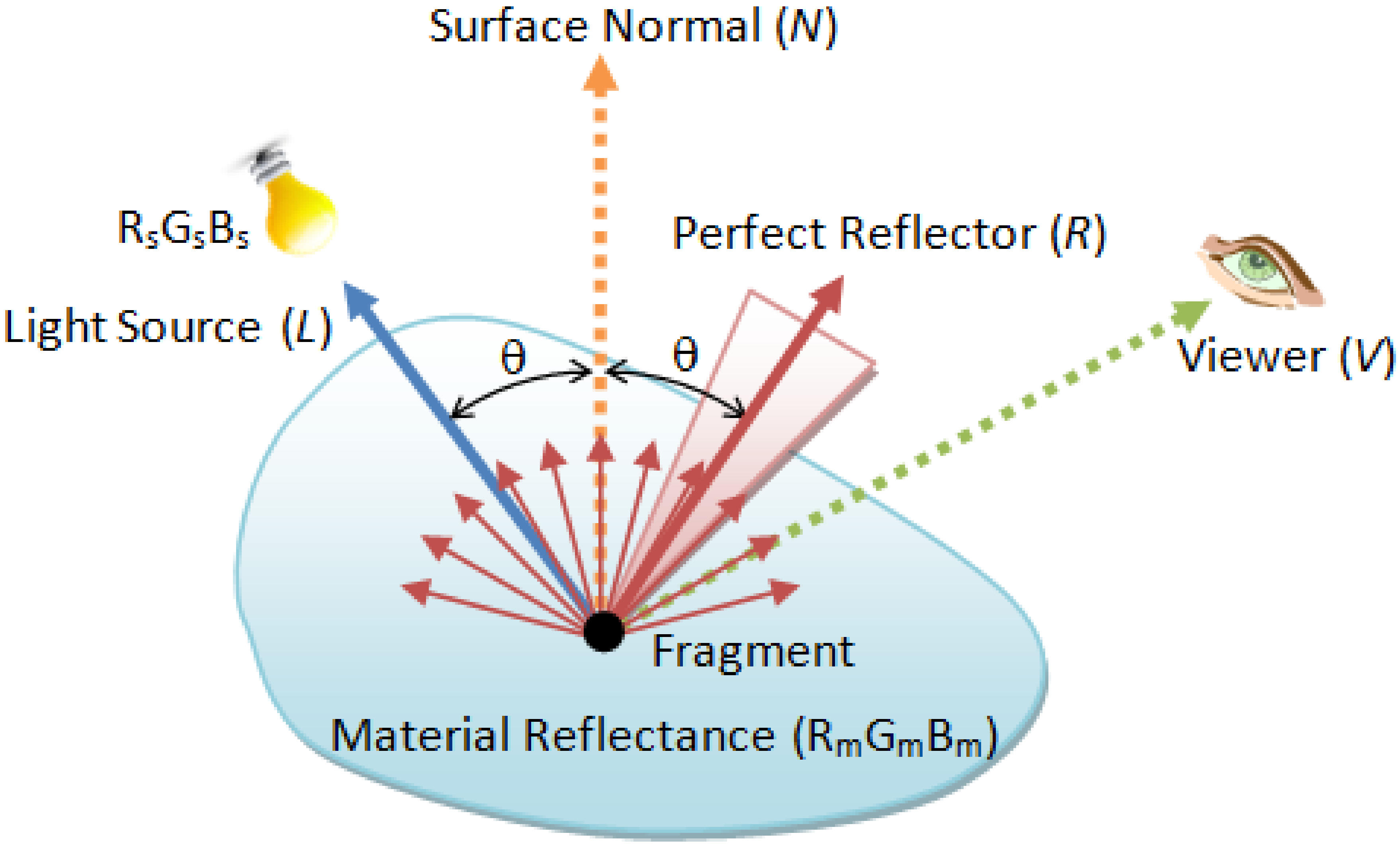
Phong Lighting Model

- A simple model that can be computed rapidly

- Has three components
  - **Diffuse**
  - **Specular**
  - **Ambient**

- Uses 4 vectors
  - To source (**l**)
  - To viewer (**v**)
  - Surface Normal (**n**)
  - Perfect reflector (**r**)

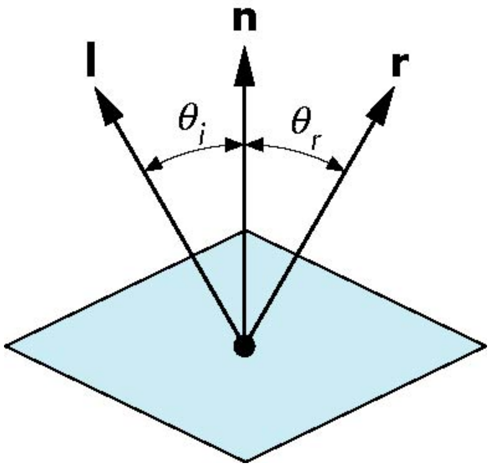



Surface Normal (*N*)
$R_sG_sB_s$
Perfect Reflector (*R*)
Light Source (*L*)
θ
θ
Viewer (*V*)
Fragment
Material Reflectance ($R_mG_mB_m$)

Ideal Reflector

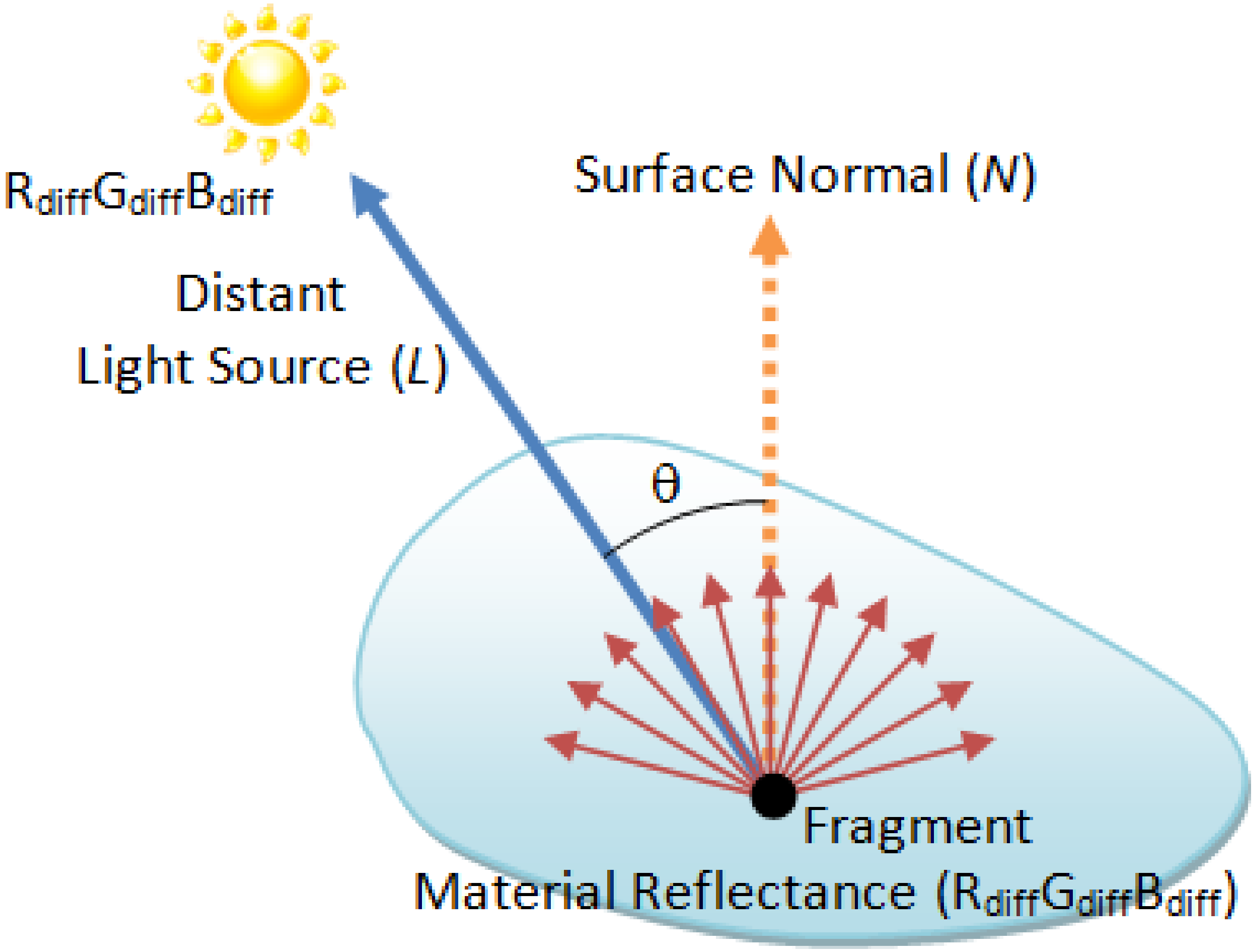
- Normal is determined by local orientation
- angle of incidence θ_i = angle of reflection θ_r
- The three vectors must be coplanar

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



Lambertian Surface

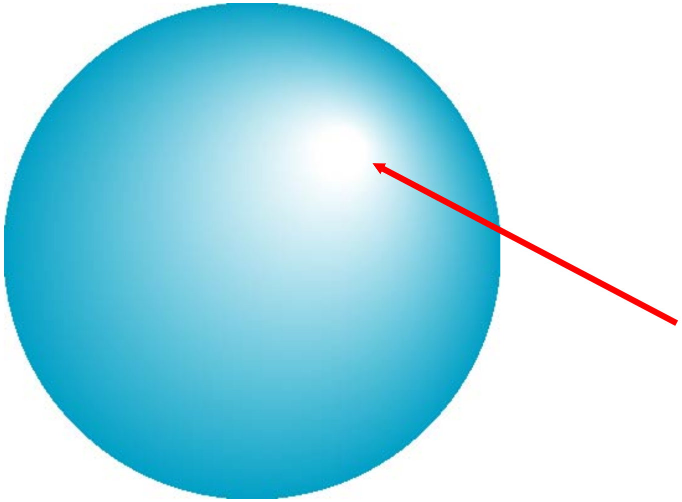
- Lambertian Surface: Perfectly diffuse reflector
- Scatters light equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
  - reflected light $\sim \cos \theta_i$
  - $\cos \theta_i = \mathbf{l} \cdot \mathbf{n}$ if the vectors are normalized
  - There are also three coefficients, k_r , k_b , k_g that show how much of each color component is reflected


$R_{diff}G_{diff}B_{diff}$
Surface Normal (N)
Distant
Light Source (L)
θ
Fragment
Material Reflectance ($R_{diff}G_{diff}B_{diff}$)

Diffuse Light

Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors/mirrors)
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection



specular
highlight

Modeling Specular Relections

- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased

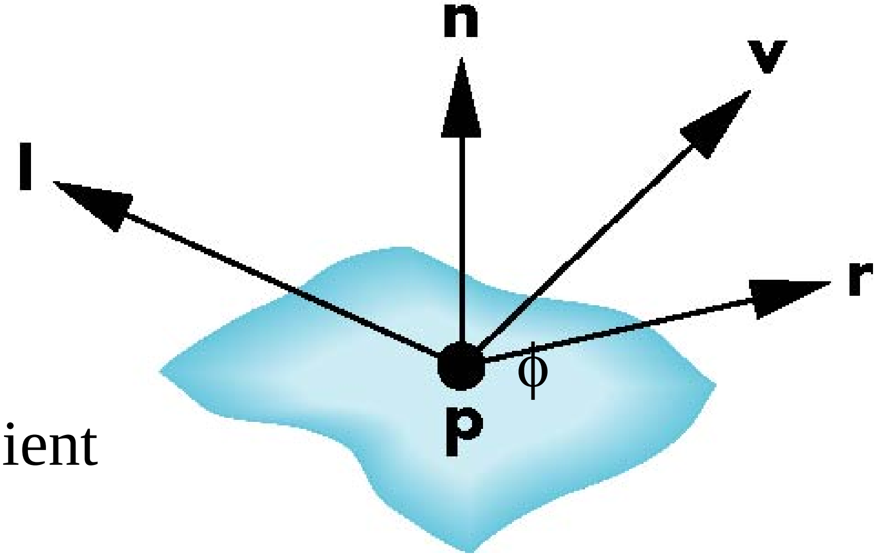
$$\mathbf{I}_r \sim k_s\, \mathbf{I} \cos^{\alpha}\phi$$

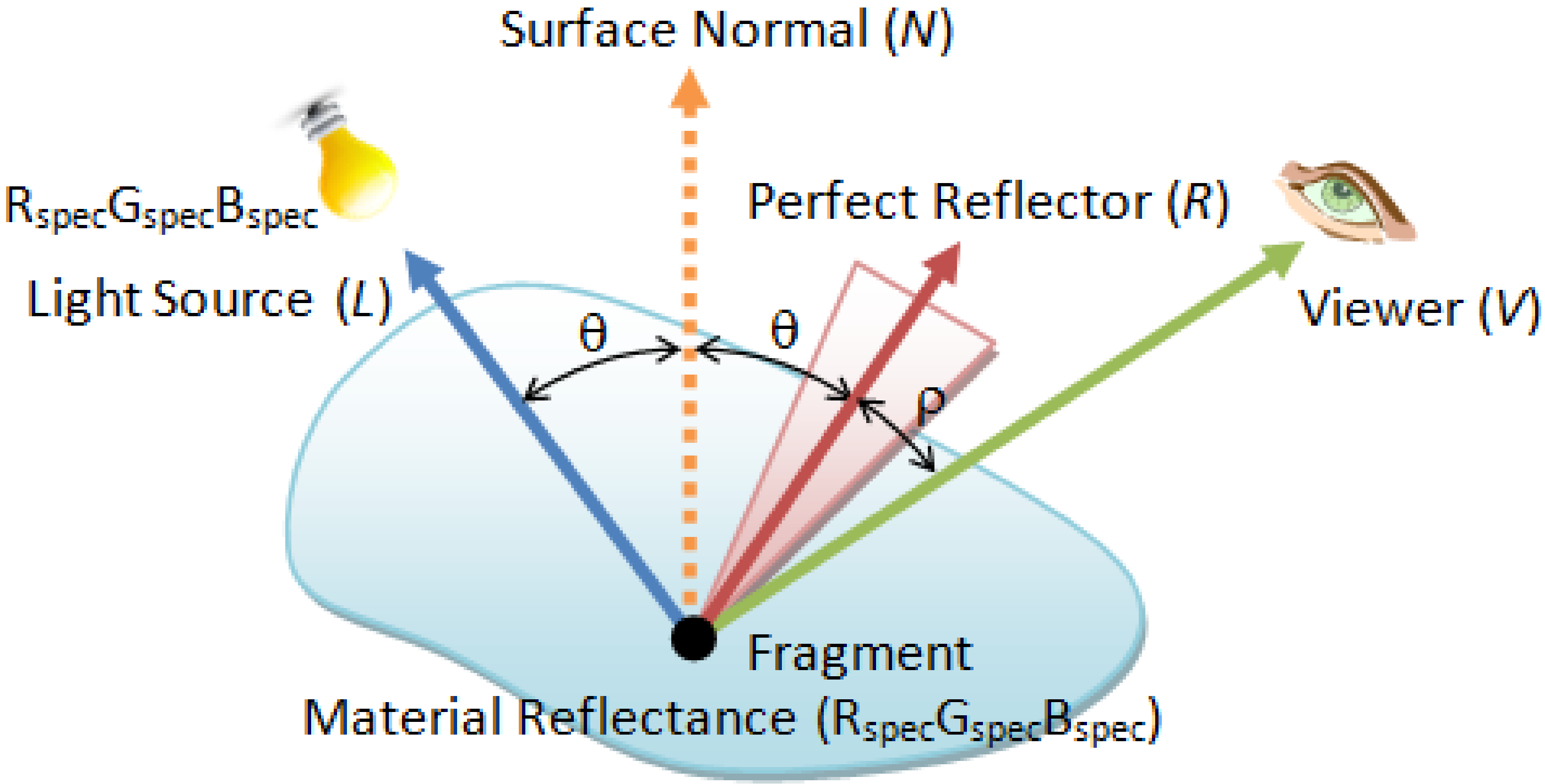
reflected intensity

absorption coefficient

incoming intensity

shininess coefficient

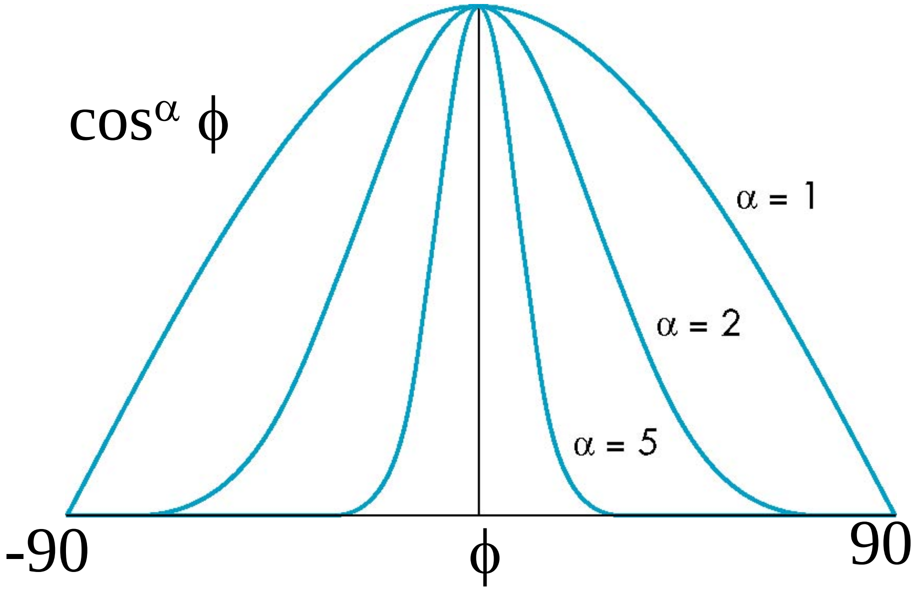




Specular Reflection

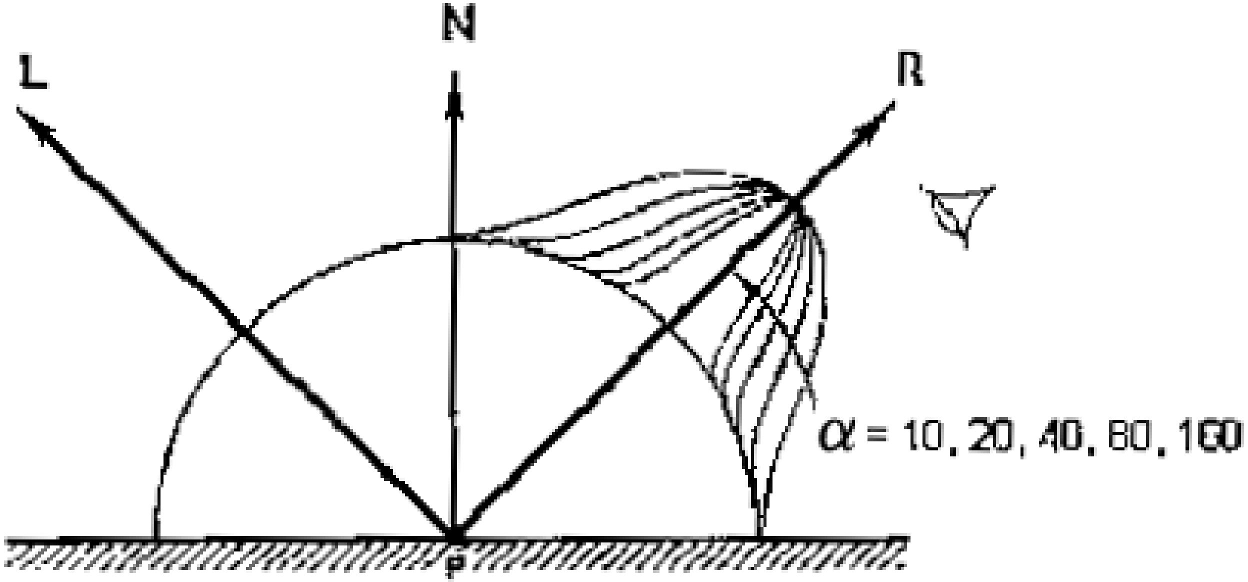
The Shininess Coefficient

- High values of α between 100 and 200 correspond to metals
- Low values between 5 and 10 give surface that look like plastic



The Shininess Coefficient

- High values of α between 100 and 200 correspond to metals
- Low values between 5 and 10 give surface that look like plastic



Ambient Light

- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment

- Amount and color depend on both the color of the light(s) and the material properties of the objects

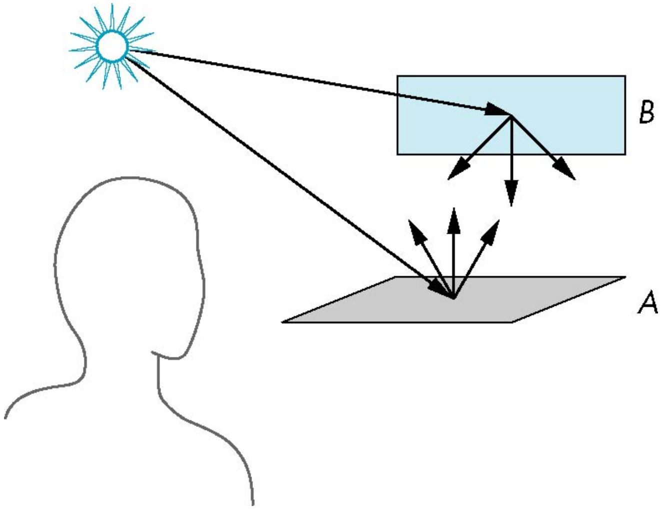
- Add k_a I_a to the equation

reflection coef.

intensity of ambient light

Distance Terms

- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them



- Instead of $\mathbf{d}^2$, we usually add a factor of the form $1/(a + b\mathbf{d} + c\mathbf{d}^2)$ to the diffuse and specular terms, where

  d: distance from the light source

  a,b,c: constants to soften the lighting

- The constant and linear terms soften the effect of the point source

Light Sources

- In the Phong Model, we add the results from each light source

- Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility
  - even though this form does not have a physical justification

- Separate red, green and blue components
- Hence, 9 coefficients for each point source
  - I_{dr}, I_{dg}, I_{db}, I_{sr}, I_{sg}, I_{sb}, I_{ar}, I_{ag}, I_{ab}

Material Properties

- Material properties match light source properties
  - 9 absorption coefficients

    k_{dr}, k_{dg}, k_{db}, k_{sr}, k_{sg}, k_{sb}, k_{ar}, k_{ag}, k_{ab}

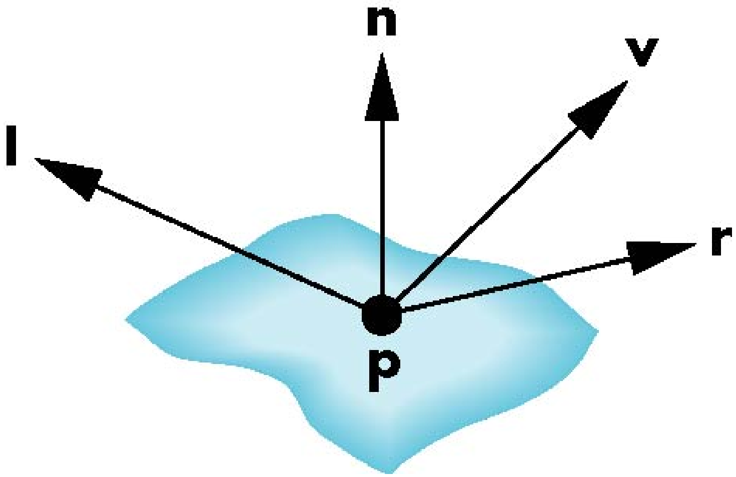
+ Shininess coefficient α

Adding up the Components

For each light source and each color component, the Phong model can be written (without the distance terms) as

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

For each color component we add contributions from all sources

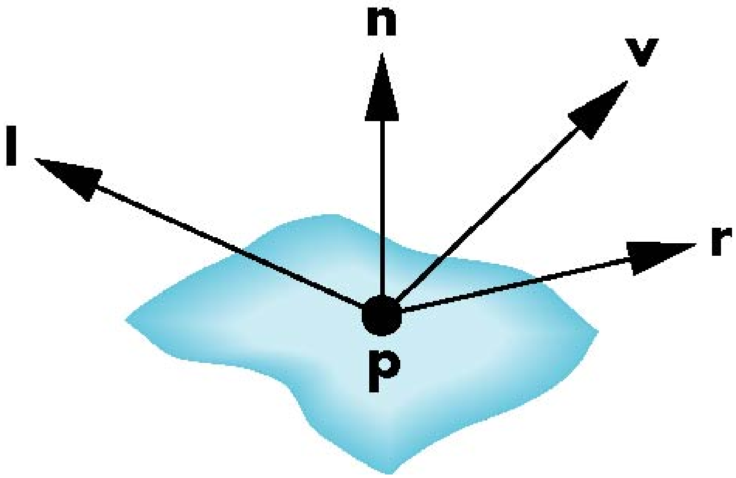


Adding up the Components

with distance terms added:

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

For each color component we add contributions from all sources

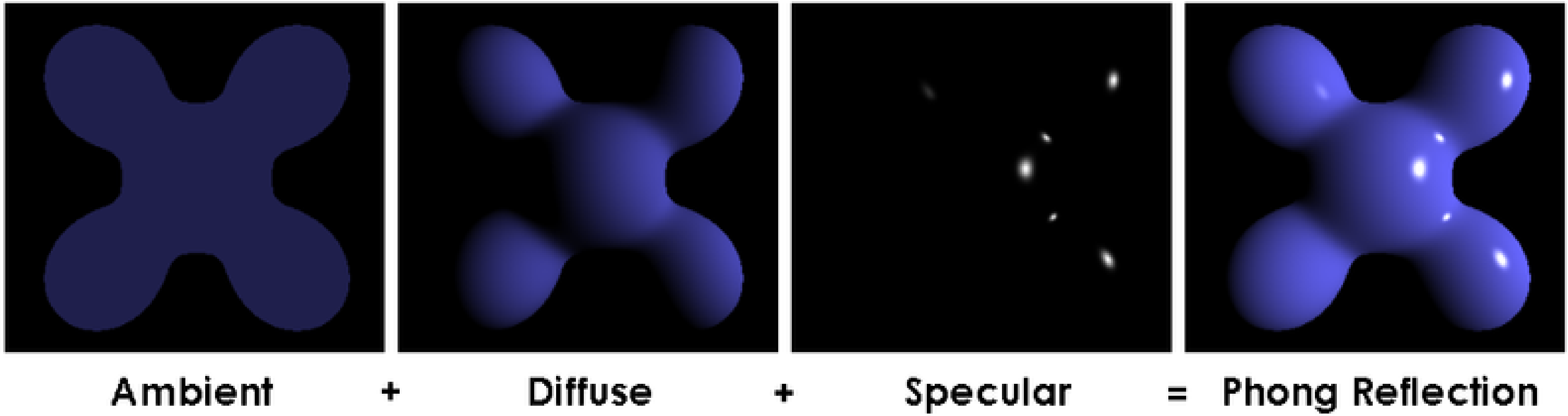


Material Properties

- Define the surface properties of a primitive

Property	Description
Diffuse	Base object color
Specular	Highlight color
Ambient	Low-light color
Emission (included if the object is emitting=producing=glowing light)	Glow color
Shininess	Surface smoothness

- you can have separate materials for front and back


Ambient
+
Diffuse
+
Specular
=
Phong Reflection

Modified Phong Lighting Model (**Blinn-Phong Lighting Model**)

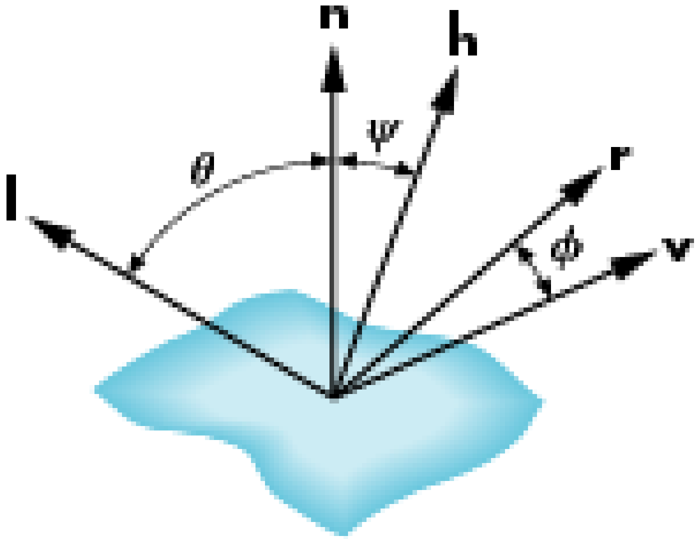
- The specular term in the Phong model is problematic: because it requires the calculation of a new reflection vector (**r**) and view vector (**v**) for each vertex

- Blinn suggested an approximation using the **halfway vector** (**h**) that is more efficient

The Halfway Vector

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

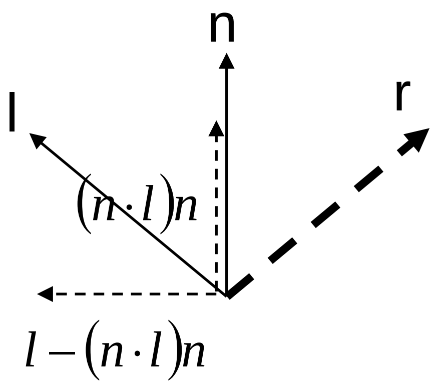
$$\mathbf{h} = (\,\mathbf{l} + \mathbf{v}\,)/\,|\,\mathbf{l} + \mathbf{v}\,|$$



$$2\psi = \phi$$

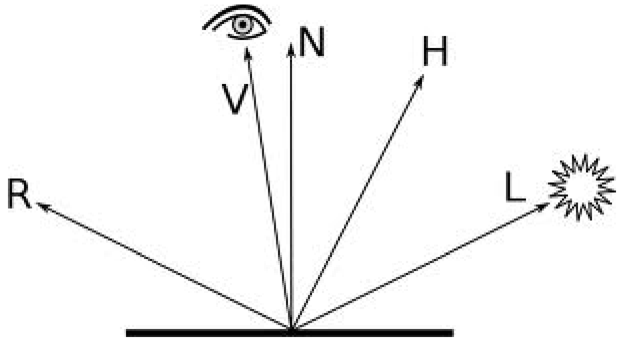
Blinn-Phong Lighting Model

$$I_s = k_s L_s (r \cdot v)^{\alpha}$$



$$r = (n \cdot l)n - [l - (n \cdot l)n]$$
$$= 2(n \cdot l)n - l$$

Time-Consuming!



An alternate formulation employs the **half vector** H

$$h = v + l, \hat{h} = \frac{h}{\|h\|}$$

$$I_s = k_s L_s (n \cdot h)^{\alpha}$$

Using the halfway vector

- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
  - This way we avoid calculation of $\mathbf{r}$
  - For flat surfaces where the surface normal is constant across the surface, this provides significant savings.
  - β is chosen to match shineness

- However, the savings using the halfway vector depends on the cases:
  - flat and curved surfaces,
  - near and far light sources,
  - and near and far viewers

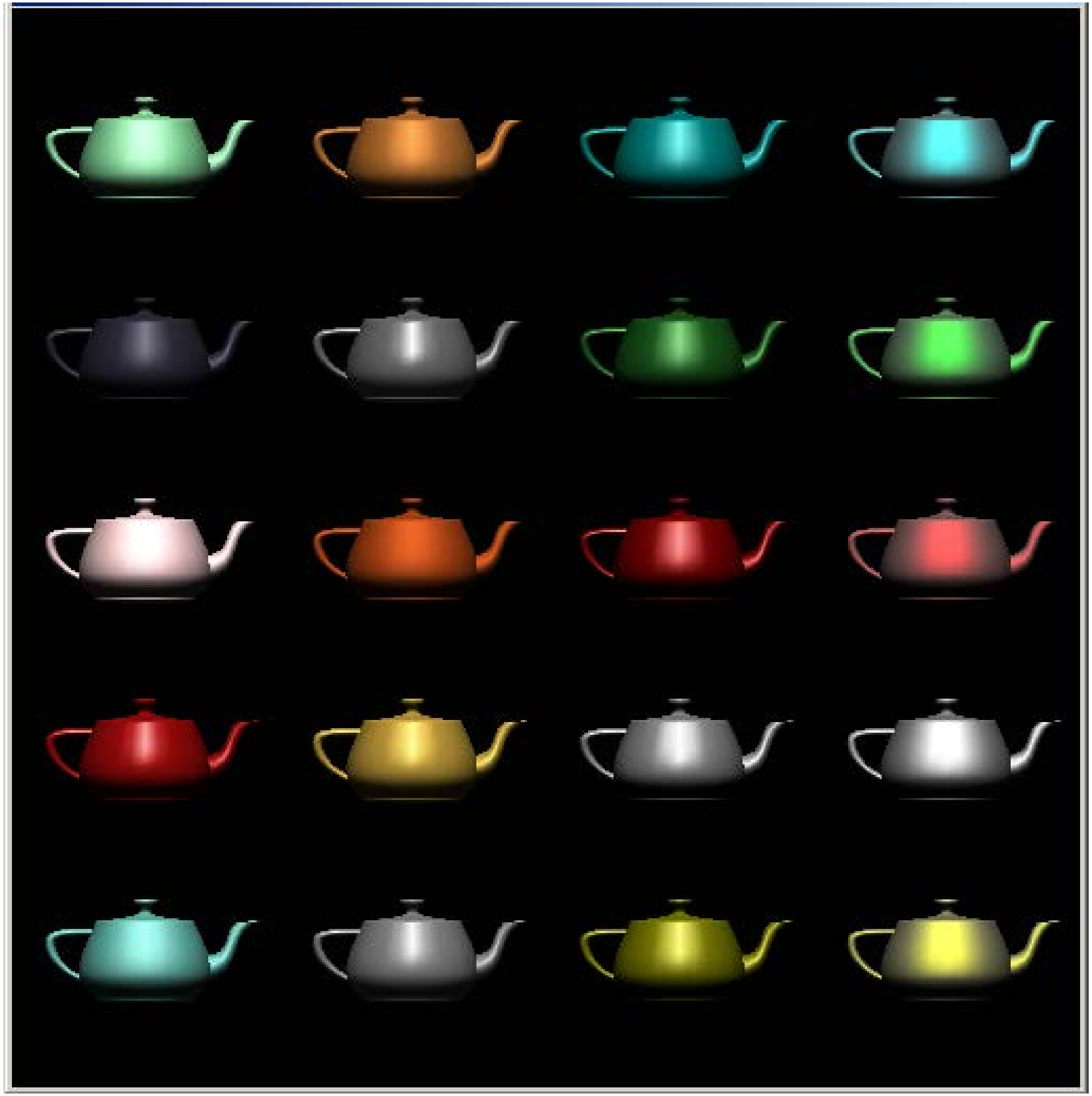
- If you assume very far light source, you can take the light vector constant across the surface, rather than computing it for each vertex separately.

- Note that halfway angle is half of angle between $\mathbf{r}$ and $\mathbf{v}$ if vectors are coplanar.

- Resulting model is known as the **Blinn-Phong lighting model**

Example

Only differences in these teapots are the parameters in the Blinn-Phong model



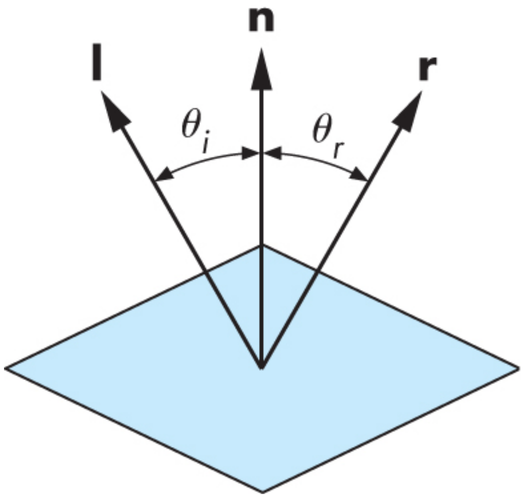
Computation of Vectors

- **l** and **v** are specified by the application
- Can compute **r** from **l** and **n**
- Problem is determining **n**
- For simple surfaces it can be calculated but how we determine **n** differs depending on underlying representation of surface.
- OpenGL leaves determination of normal to application (that means **you**!)
  - Exception for GLU quadrics and Bezier surfaces (Chapter 11)

Computing Reflection Direction

- angle of incidence = angle of reflection
- Normal, light direction and reflection direction are coplaner
- Want all three to be unit length

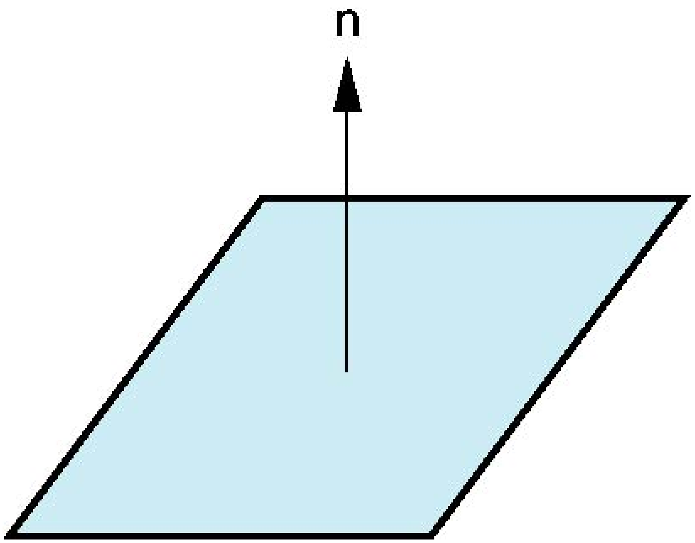
$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n})\,\mathbf{n} - \mathbf{l}$$



Plane Normals

- Equation of plane: $ax+by+cz+d = 0$
- From Chapter 3, we know that plane is determined by three points p_0, p_1, p_2 or normal $\mathbf{n}$ and p_0
- Normal can be obtained by

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



$$
\begin{aligned}
\vec{v} \times \vec{w} &= \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix} \\
&= \hat{i} \begin{vmatrix} v_2 & v_3 \\ w_2 & w_3 \end{vmatrix} - \hat{j} \begin{vmatrix} v_1 & v_3 \\ w_1 & w_3 \end{vmatrix} + \hat{k} \begin{vmatrix} v_1 & v_2 \\ w_1 & w_2 \end{vmatrix} \\
&= (v_2 w_3 - v_3 w_2)\hat{i} - (v_1 w_3 - v_3 w_1)\hat{j} + (v_1 w_2 - v_2 w_1)\hat{k}
\end{aligned}
$$

$$x_1 \times x_2 = \begin{vmatrix} i & j & k \\ 2 & -3 & 1 \\ -2 & 1 & 1 \end{vmatrix}$$

$$= \begin{vmatrix} -3 & 1 \\ 1 & 1 \end{vmatrix} i - \begin{vmatrix} 2 & 1 \\ -2 & 1 \end{vmatrix} j + \begin{vmatrix} 2 & -3 \\ -2 & 1 \end{vmatrix} k$$

$$= [(-3)(1)-(1)(1)]i - [(2)(1)-(-2)(1)]j + [(2)(1)-(-2)(-3)]k$$

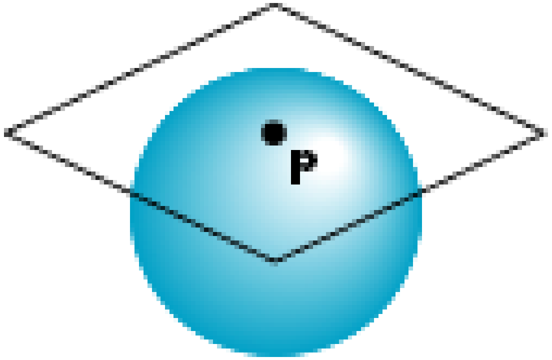
$$= -4i - 4j + 8k$$

Normal to Sphere

- Implicit function $f(x, y, z)=0$

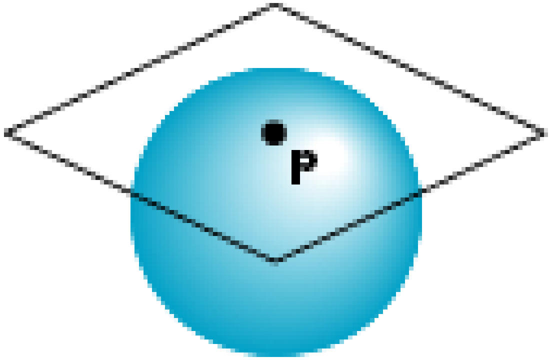
- Normal given by gradient

- Sphere $f(\mathbf{p}) = \mathbf{p} \cdot \mathbf{p} - 1$
- $n = [\partial f/\partial x, \partial f/\partial y, \partial f/\partial z]^T = \mathbf{p}$



Parametric Form for Sphere

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



- Tangent plane determined by vectors

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

where **p** = p(x,y,z)

- Normal given by cross product

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

General Case

- We can compute parametric normals for other simple cases
  - Quadrics
  - Parameteric polynomial surfaces
    - Bezier surface patches (Chapter 11)

SHADING IN OPENGL

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Objectives

- Introduce the OpenGL shading methods
  - per vertex vs per fragment shading
  - Where to carry out

- Discuss polygonal shading
  - Flat
  - Smooth
  - Gouraud

OpenGL shading

- **Need**
  - Normals
  - Material properties
  - Lights info

- State-based shading functions have been deprecated
- Compute in application or send attributes to shaders to compute there

Normalization

- Cosine terms in lighting calculations can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length but
  - Length can be affected by transformations
  - Note that scaling does not preserve length

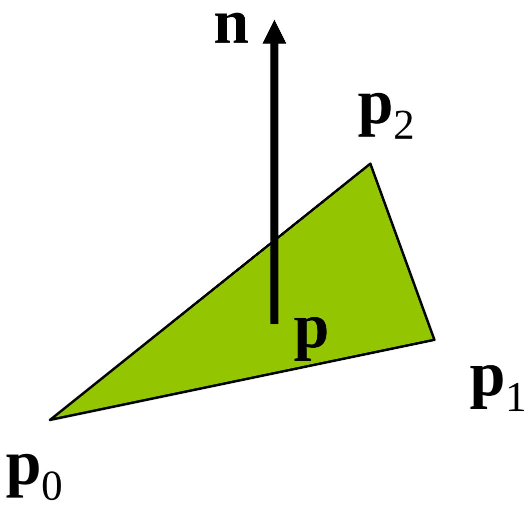
>> GLSL has a normalization function, use that when you need to normalize

Normal for Triangle

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

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

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



Note that right-hand rule determines outward face

Specifying a Point Light Source

- For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position

```
vec4 diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 light0_pos = vec4(1.0, 2.0, 3.0, 1.0);
```

Distance and Direction

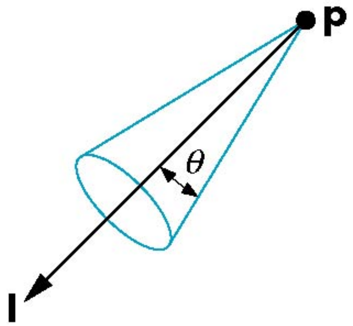
- The source colors are specified in RGBA

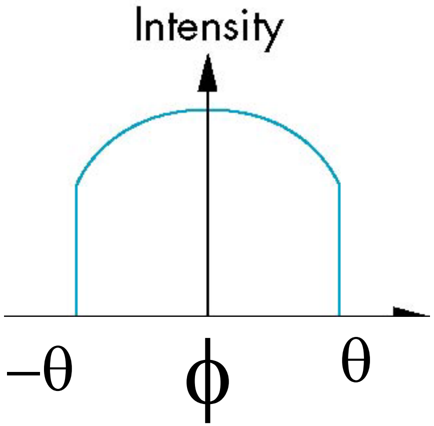
- The position is given in homogeneous coordinates
  - If w =1.0, we are specifying a source at finite location
  - If w =0.0, we are specifying a directional source with the given direction vector

- The coefficients in distance terms are usually quadratic (1/(a+(b*d)+(c*d*d))) where d is the distance from the point being rendered to the light source

Spotlights

- Derive from point source
  - Direction
  - Cutoff
  - Attenuation proportional to $\cos^{\alpha}\phi$





Global Ambient Light

- Ambient light depends on color of light sources
  - A red light in a white room will cause a red ambient term that disappears when the light is turned off

- A global ambient term that is often helpful for testing

Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix

- Depending on where we place the position (direction) setting function, we may
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently

Material Properties

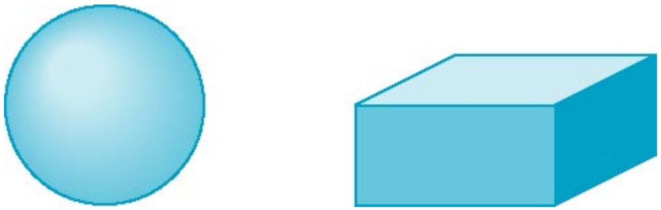
- Material properties should match the terms in the light model

- Reflectivities
- w component (alpha channel) gives opacity

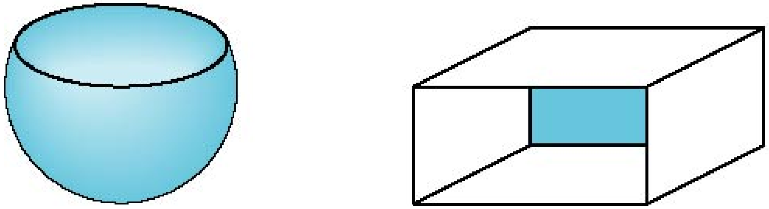
```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(1.0, 0.8, 0.0, 1.0);
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```

Front and Back Faces

- Every face has a front and back
- For many objects, we never see the back face so we don't care how or if it's rendered
- If it matters, we can handle in shader



back faces not visible



back faces visible

Emissive Term

- We can simulate a light source in OpenGL by giving the material an emissive component

- This component is unaffected by any sources or transformations

Transparency

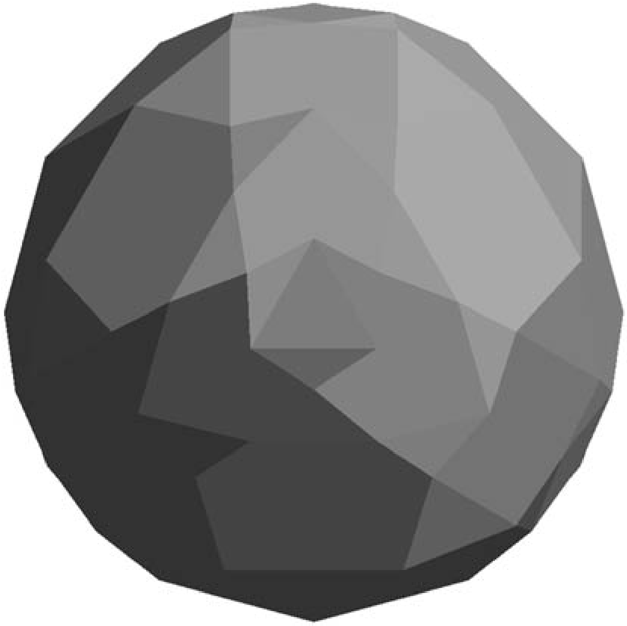
- Material properties are specified as RGBA values
- The A value can be used to make the surface translucent
- The default is that all surfaces are opaque regardless of A
- We can enable blending and use this feature

Polygonal Shading

- In per vertex shading, shading calculations are done for each vertex
    - Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
    - Alternately, we can send the parameters to the vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)
- We can also use uniform variables to shade with a single shade (flat shading)

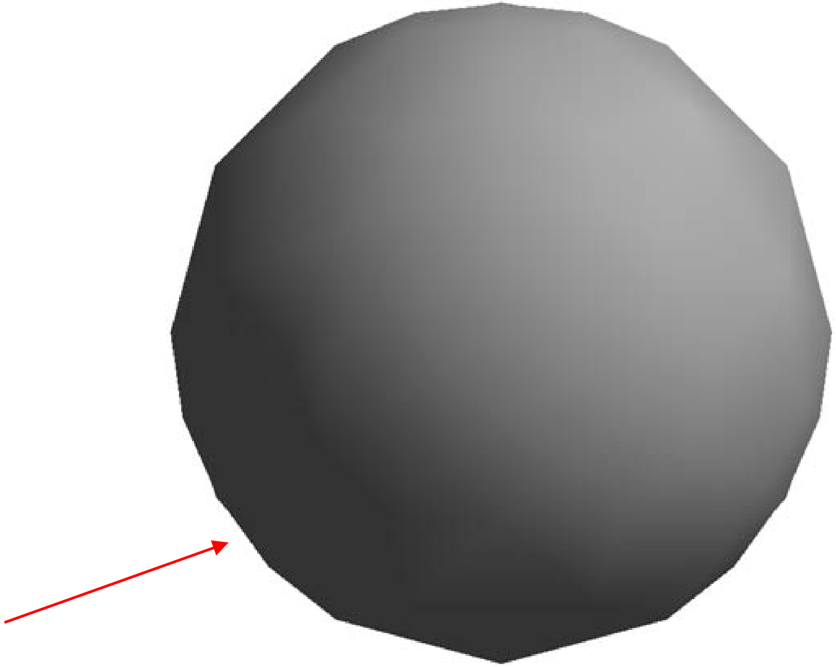
Flat Shading

- Triangles have a single normal
  - Shades at the vertices as computed by the Phong model can be almost the same
  - Identical for a distant viewer (default) or if there is no specular component
- Consider the model of a sphere
- Want different normals at each vertex even though this concept is not quite correct mathematically



Smooth Shading

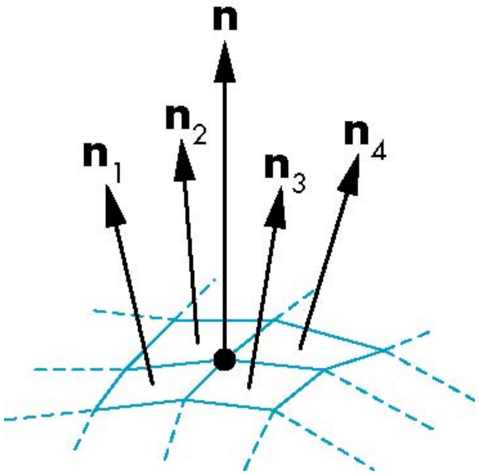
- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*



Mesh Shading

- The previous sphere example is not general because we knew the normal at each vertex analytically
- For more complex polygonal models, Gouraud proposed that we use the **average of the normals around a mesh vertex**

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



Gouraud and Phong Shading

- **Gouraud Shading**
  - Find average normal at each vertex (vertex normals)
  - Apply Blinn-Phong lighting model **at each vertex**
  - Send calculated per-vertex shade to fragment shader through rasterizer
    - vertex shades interpolated across each polygon

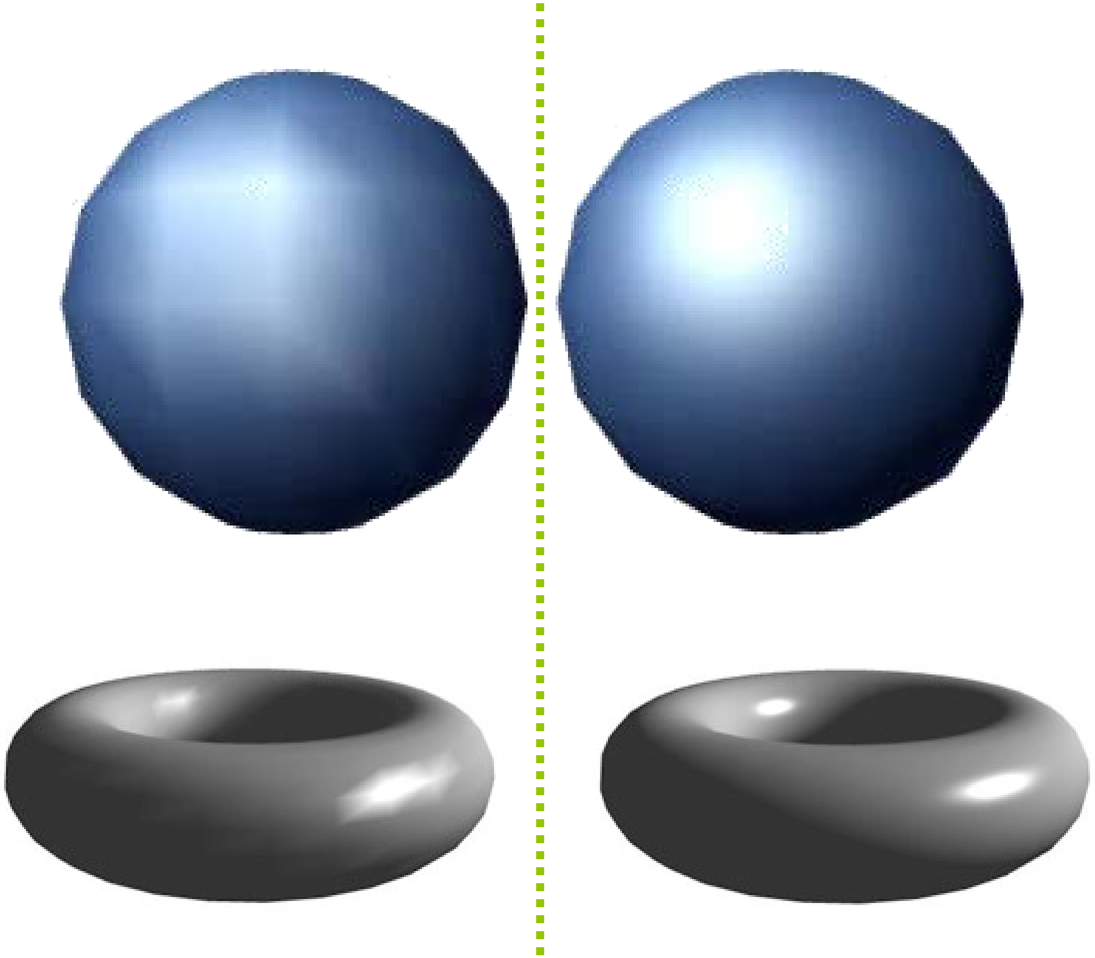
- **Phong Shading**
  - Find vertex normals
  - Send the normals to fragment shader through rasterizer
    - vertex normals interpolated across edges
    - edge normals interpolated across polygon
  - Apply Blinn-Phong lighting model **at each fragment** using the interpolated normal

Comparison

- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges

- Phong shading requires much more work than Gouraud shading
  - Until recently not available in real time systems
  - Now can be done using fragment shaders

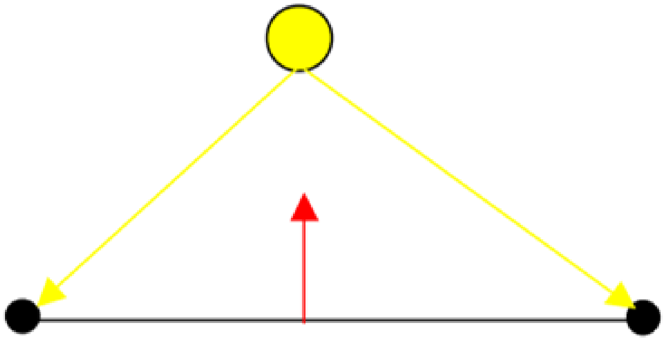
- Both need data structures to represent meshes so we can obtain vertex normals per-vertex Gouraud shading (left) vs per-fragment Phong shading (right)



- Notice how the per-vertex shading model doesn't result in a smooth highlight.
- **Vertex shaders operate on vertices in 3D space.**
- **Fragment shaders operate on fragments in 2D space.**

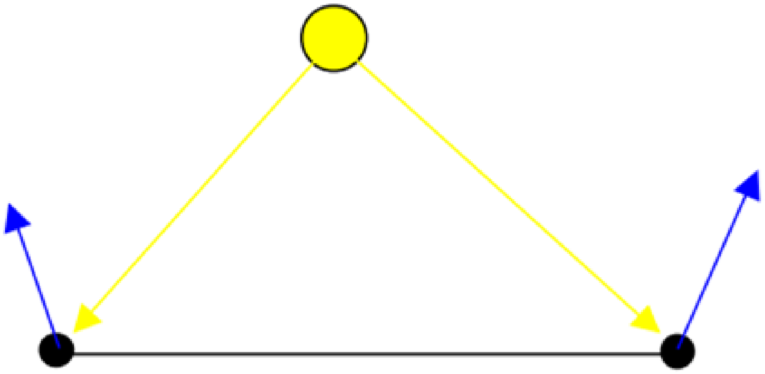
Vertex: ● Light: ○ Normal: →

Flat shading



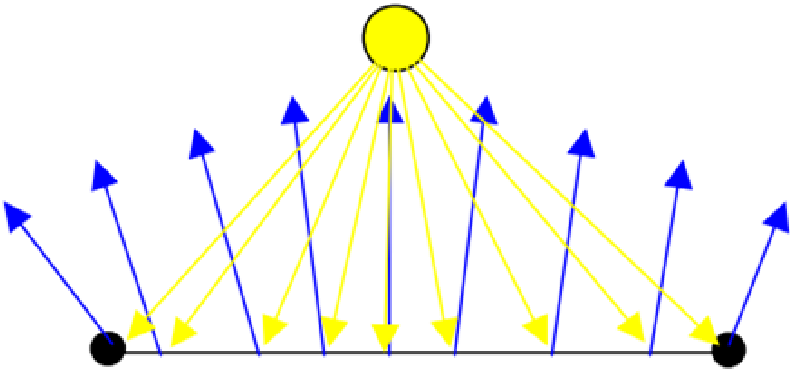
Only the first normal of the triangle is used to compute lighting in the entire triangle.

Goraud shading



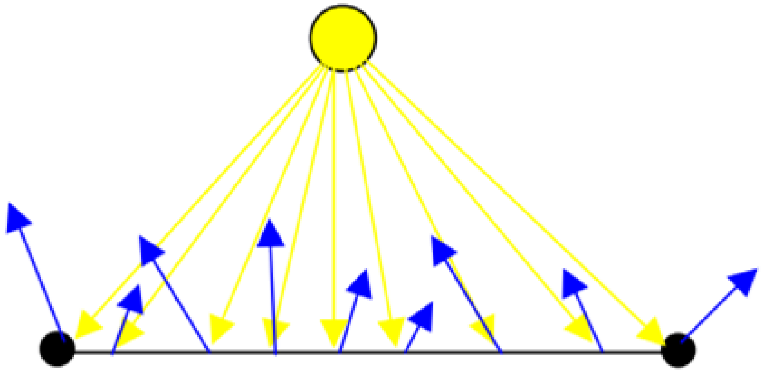
The light intensity is computed at each vertex and interpolated across the surface.

Phong shading



Normals are interpolated across the surface, and the light is computed at each fragment.

Bump mapping



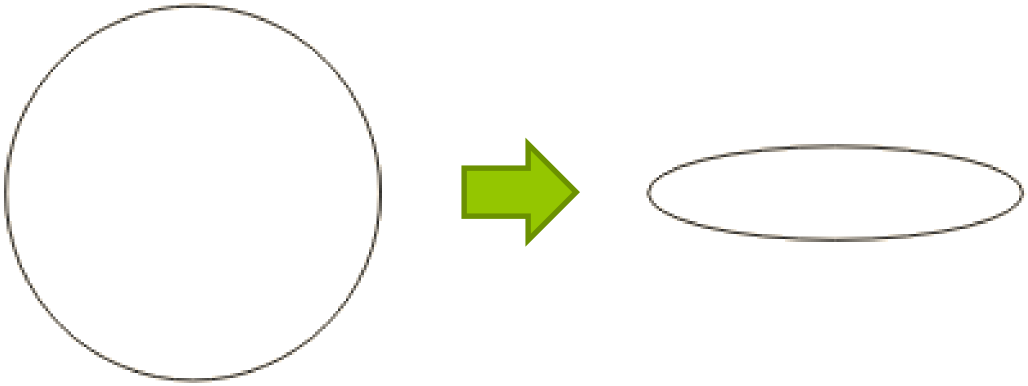
Normals are stored in a bumpmap texture, and used instead of Phong normals.

Vertex Lighting Shaders I

```
// vertex shader
in vec4 vPosition;
in vec4 vNormal;
out vec4 color;  //vertex shade


// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform float Shininess;
uniform vec4 LightPosition;
uniform mat4 ModelView, NormalMatrix;
uniform mat4 Projection;
```

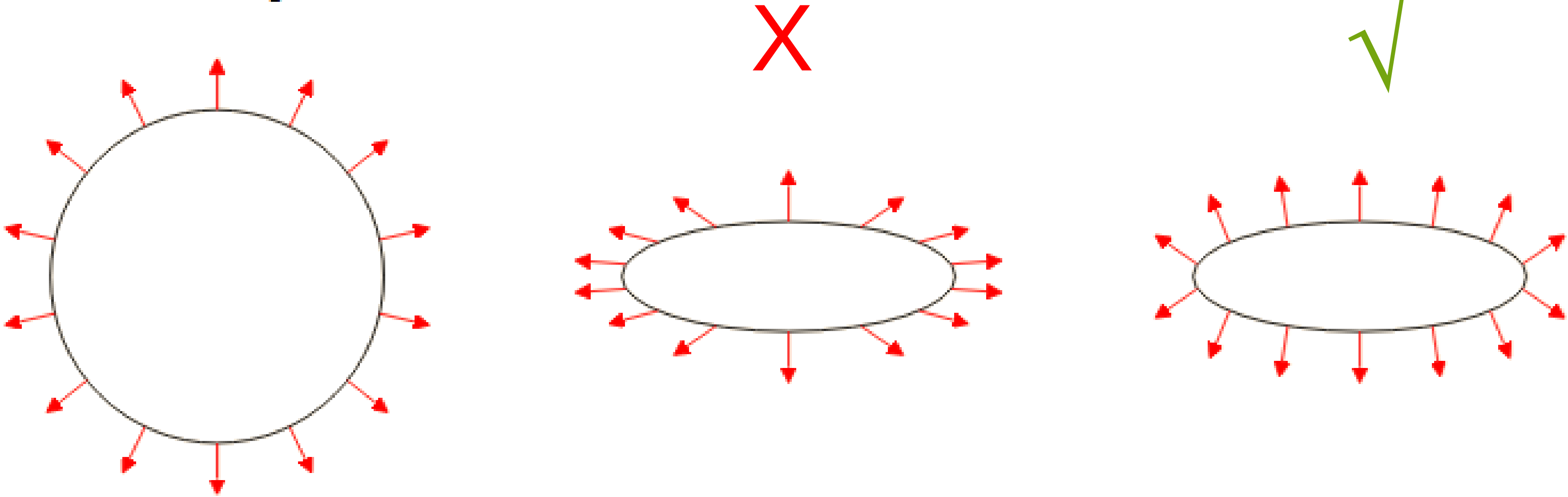
- Here we declare numerous variables that are needed in computing a color using a simple lighting model.
- All of the uniform values are passed in from the application and describe the material and light properties being rendered.

- NormalMatrix is the inverse transpose of ModelView Matrix



This is all well and good, but consider the normals in this transformation:

Circle Scaling with Normals



- The ellipse in the middle has the normals that you would expect if you transformed the normals from the circle by the same matrix the circle was transformed by.
- They may be unit length, but they no longer reflect the *shape* of the ellipse.
- The ellipse on the right has normals that reflect the actual shape.

- It turns out that, what you really want to do is transform the normals with the same rotations as the positions, but invert the scales.
- That is, a scale of 0.5 along the X axis will shrink positions in that axis by half. For the surface normals, you want to *double* the X value of the normals, then normalize the result.

- So, what we must do is compute something called the *inverse transpose* of the matrix in question.

$$N \bullet V = 0$$

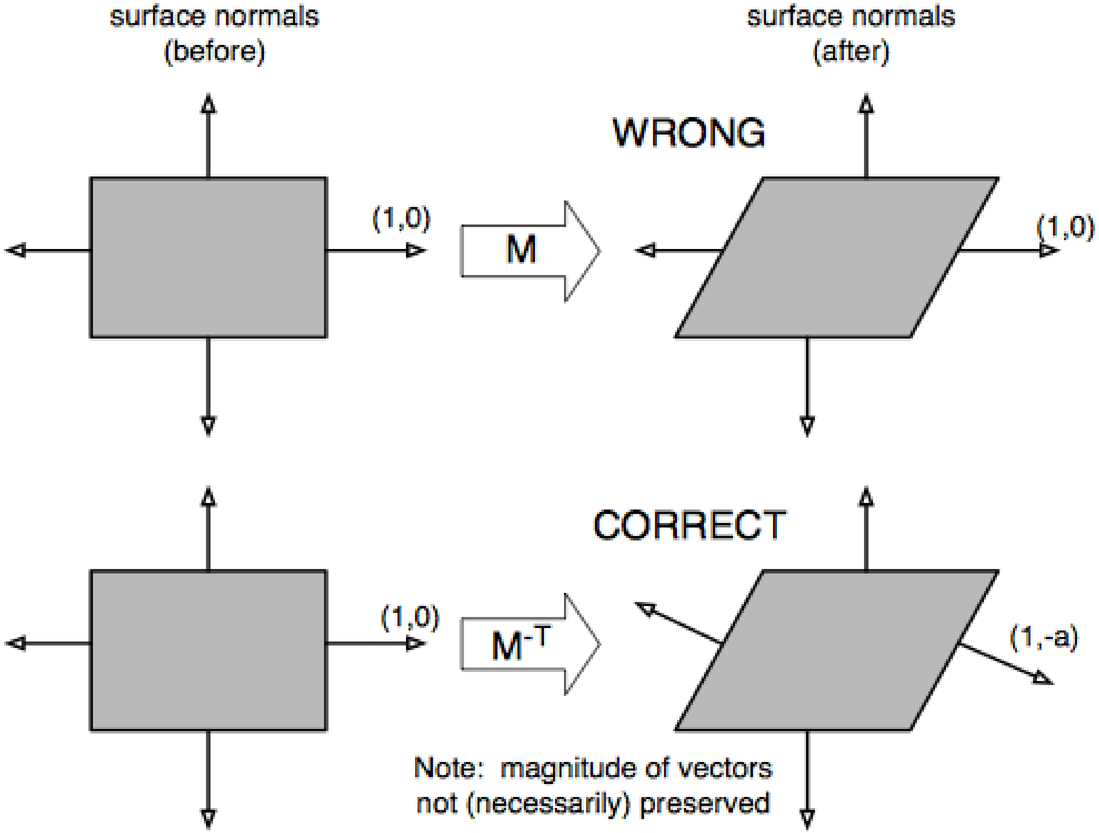
$$N^T V = 0$$

$$N^T M^{-1} M V = 0$$

$$\underbrace{M^{-T} N}_{N'^T} \underbrace{M V}_{V'} = 0$$

where N is the normal and V is a tangent vector. Since they are perpendicular their dot product is zero. M is an invertible transformation ($M^{-1}M = I$). N' and V' are the vectors transformed by M.

$$M = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \quad M^{-T} = \begin{bmatrix} 1 & 0 \\ -a & 1 \end{bmatrix}$$



Vertex Lighting Shaders II

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

    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 V = normalize( -pos );
    vec3 H = normalize( L + V );

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(NormalMatrix*vNormal).xyz;
```

- **pos** represents the vertex’s position in **eye coordinates** (so**: eye is at the origin after modelview transform**) (and remember **eye = camera = viewer**)

- **L** represents the vector from the vertex to the light

- **V** represents the “viewer” vector (vector to the camera), which is the vector from the vertex’s eye-space position to the origin

- **H** is the “half vector” which is the normalized half-way vector between the light and viewer vectors

- **N** is the transformed vertex normal

- Note that all of these quantities are vec3’s, since we’re dealing with vectors, as compared to homogenous coordinates.
- When we need to convert from a homogenous coordinate to a vector, we use a vector swizzle to extract the components we need.

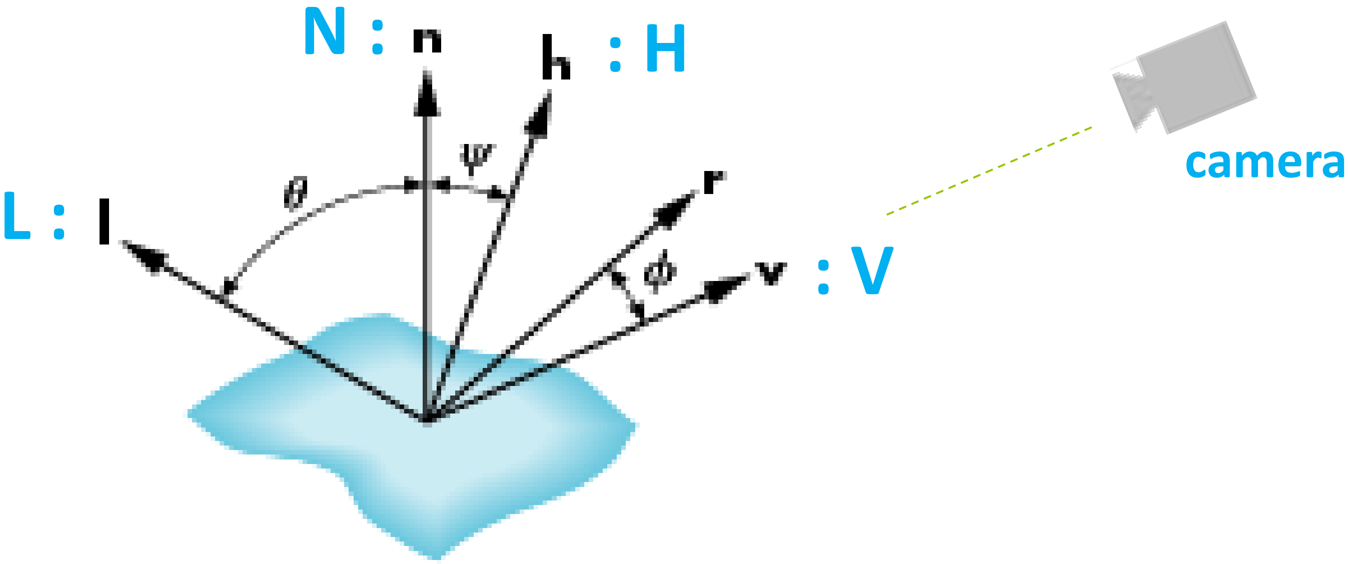
$$I = k_d I_d \ (\mathbf{l} \cdot \mathbf{n}) \ + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$$

- **Blinn-Phong Model:** Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- $\mathbf{h}$ is normalized vector halfway between $\mathbf{l}$ and $\mathbf{v}$:

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

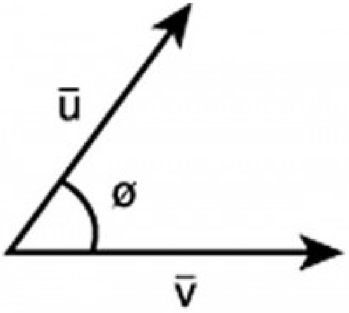
$$I = k_d I_d \ (\mathbf{l} \cdot \mathbf{n}) \ + k_s I_s (\mathbf{n} \cdot \mathbf{h})^{\beta} + k_a I_a$$

$$2\psi = \phi$$

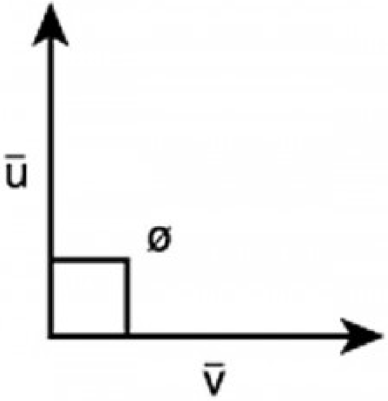


Vertex Lighting Shaders III

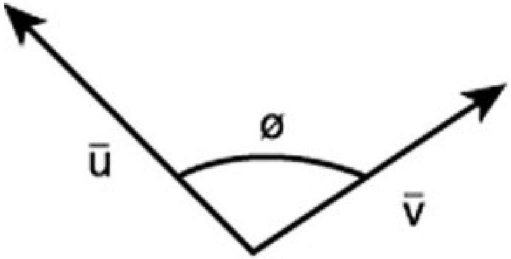
```
    // Compute terms in the illumination equation
    vec4 ambient = AmbientProduct;

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

    float Ks = pow( max(dot(N, H), 0.0), Shininess );
    vec4  specular = Ks * SpecularProduct;
    // if the light is behind the object, again we correct the
    // specular contribution
    if( dot(L, N) < 0.0 )
        specular = vec4(0.0, 0.0, 0.0, 1.0)

    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```



$\varnothing < 90$, Acute
$\bar{u} \bullet \bar{v} > 0$



$\varnothing = 90$, Right angle
$\bar{u} \bullet \bar{v} = 0$



$\varnothing > 90$, Obtuse
$\bar{u} \bullet \bar{v} < 0$

Starting with the most influential component of lighting, the diffuse color,

we use the dot product of the normal and light vector, clamping the value if the dot product is negative **which physically means that ray of light does not hit that vertex.**

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

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

    float Ks = pow( max(dot(N, H), 0.0), Shininess );
    vec4  specular = Ks * SpecularProduct;
    // if the light is behind the object, we correct the
    // specular contribution
    if( dot(L, N) < 0.0 )
        specular = vec4(0.0, 0.0, 0.0, 1.0)

    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```

$N \cdot H \leq 0$ means either the camera doesn't see that vertex or the light doesn't hit that vertex. Either way we don't want the negative contributions, so it is clamped to 0.

Vertex Lighting Shaders IV

```
// fragment shader

in vec4 color;

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

Per-Fragment Lighting

Now we do the same set of computations on the fragment-shader side

Fragment Lighting Shaders I

```
// vertex shader
in vec4 vPosition;
in vec4 vNormal;

// output values that will be interpolated per-fragment
out vec3 fN;
out vec3 fV;
out vec3 fL;

uniform mat4 ModelView, NormalMatrix;
uniform vec4 LightPosition; //in eye coordinates
uniform mat4 Projection;
```

Fragment Lighting Shaders II

```
void main()
{
   fN = (NormalMatrix*vNormal).xyz;
   fV = -(ModelView*vPosition).xyz;
   fL = LightPosition.xyz;

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

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

Fragment Lighting Shaders III

```
// fragment shader

// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fL;
in vec3 fV;

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

Fragment Lighting Shaders IV

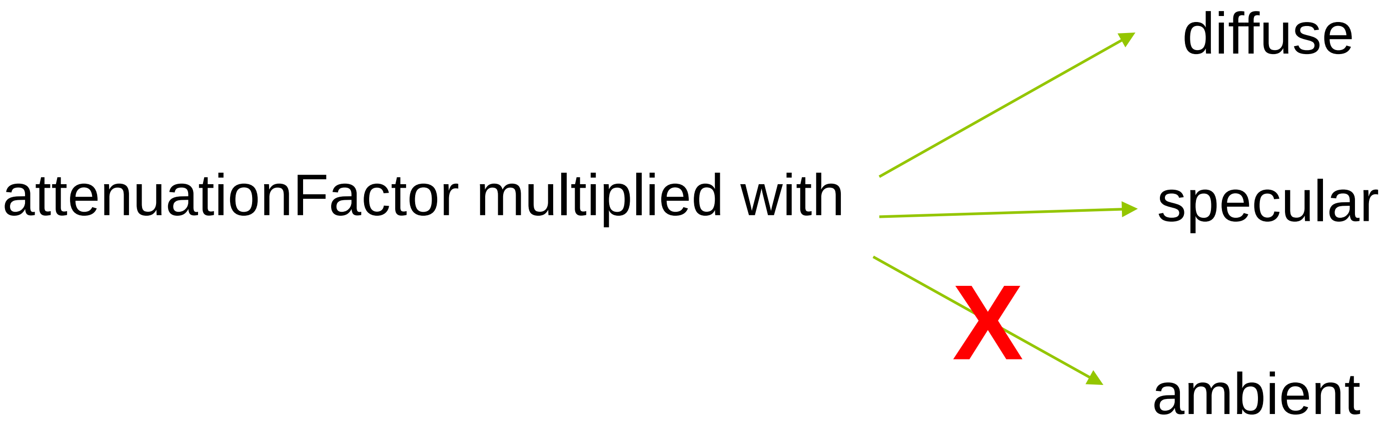
```
void main()
{
    // Normalize the input lighting vectors

   vec3 N = normalize(fN);
    vec3 V = normalize(fV);
    vec3 L = normalize(fL);

    vec3 H = normalize( L + V );
    vec4 ambient = AmbientProduct;
```

Fragment Lighting Shaders V

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

    float Ks = pow(max(dot(N, H), 0.0), Shininess);
    vec4 specular = Ks*SpecularProduct;

    // discard the specular highlight if the light's behind the vertex
    if( dot(L, N) < 0.0 )
        specular = vec4(0.0, 0.0, 0.0, 1.0);

    gl_FragColor = ambient + diffuse + specular;
    gl_FragColor.a = 1.0;
}
```

Attenuation?

```
float distToLight = length(lightPosition - vertexPosition);

float attenuationFactor =
1.0 / ( a0 + a1*distToLight + a2*distToLight*distToLight );
```



attenuationFactor multiplied with

diffuse

specular

X

ambient