

6. Lighting and Shading

Lecture Overview

- Lighting and Shading (ANG Ch. 6)

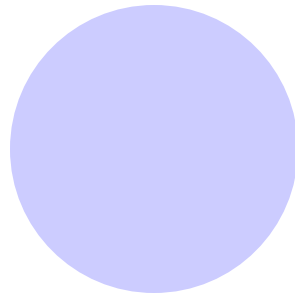
Shading I

Objectives

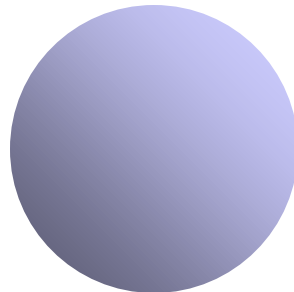
- Learn to shade objects so their images appear three-dimensional
- Introduce the types of **light-material interactions**
- Build a simple reflection model---the **Phong model**--- that can be used with real time graphics hardware

Why we need shading

- Suppose we build a model of a sphere using many polygons and color it with `glColor`. We get something like

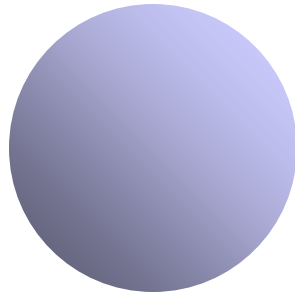


- But we want



Shading

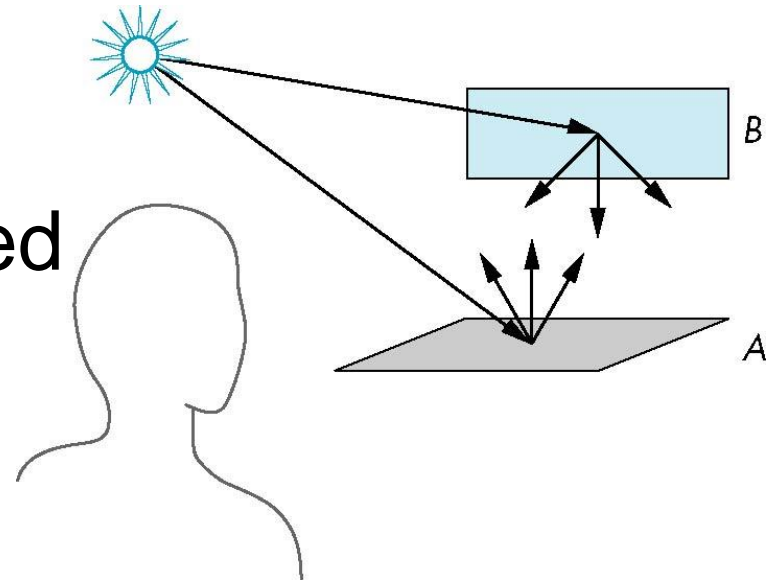
- Why does the image of a real sphere look like



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

Scattering

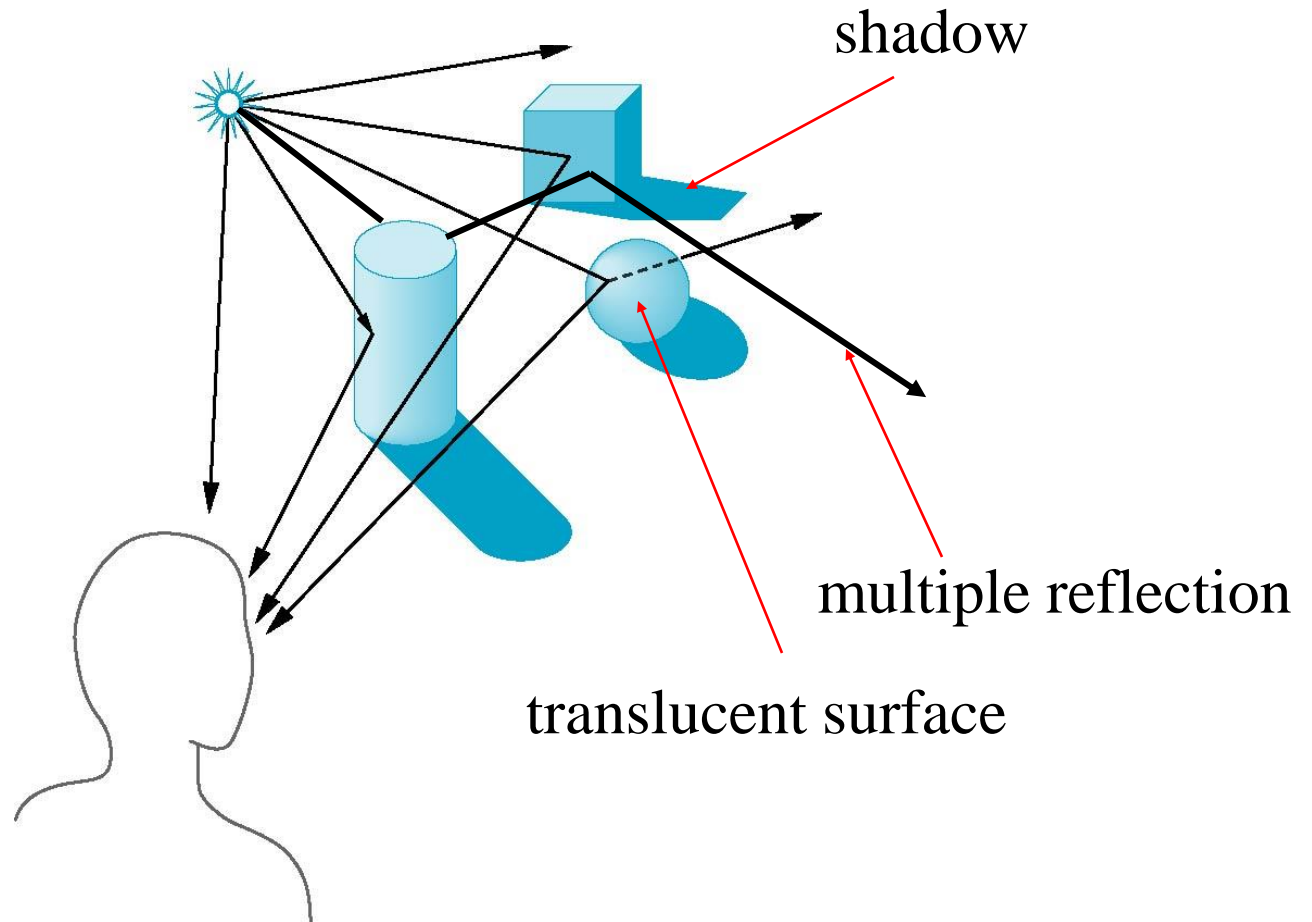
- Light strikes A
 - Some scattered
 - Some absorbed
- Some of scattered light strikes B
 - Some scattered
 - Some absorbed
- Some of this scattered light strikes A and so on



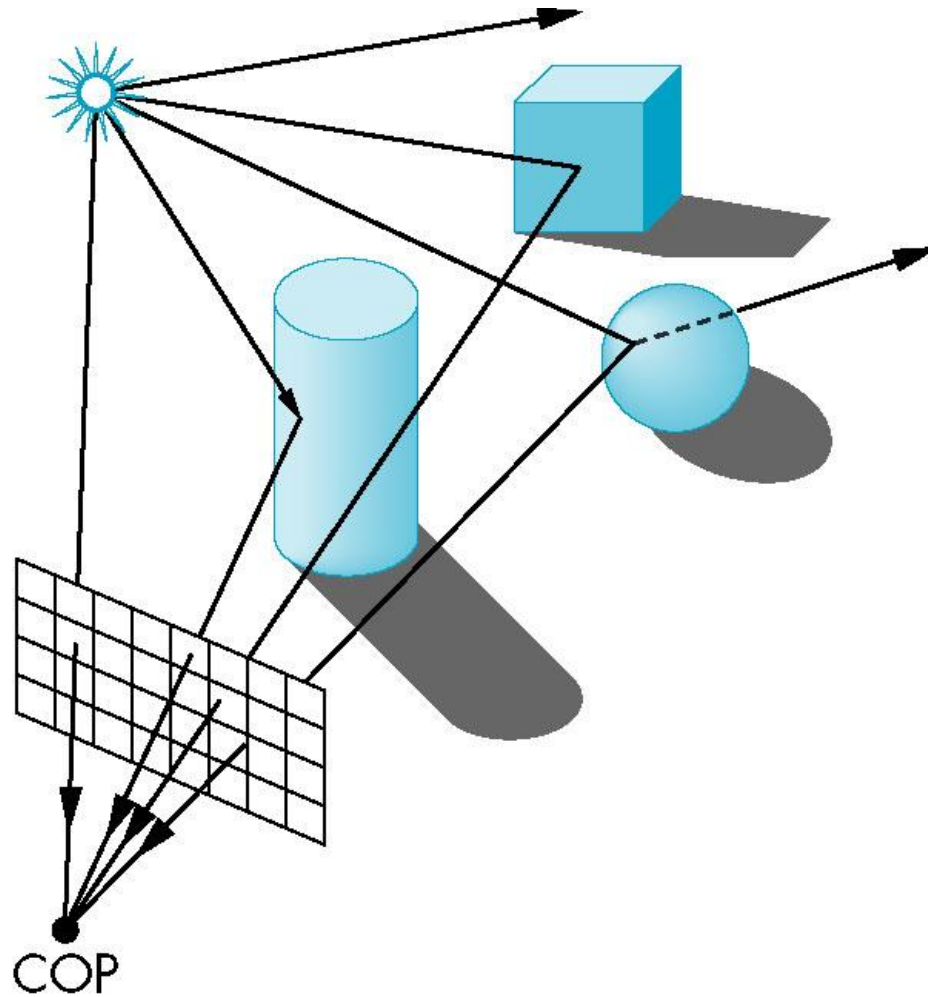
Rendering Equation

- The infinite scattering and absorption of light can be described by the *rendering equation*
 - Cannot be solved in general
 - 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



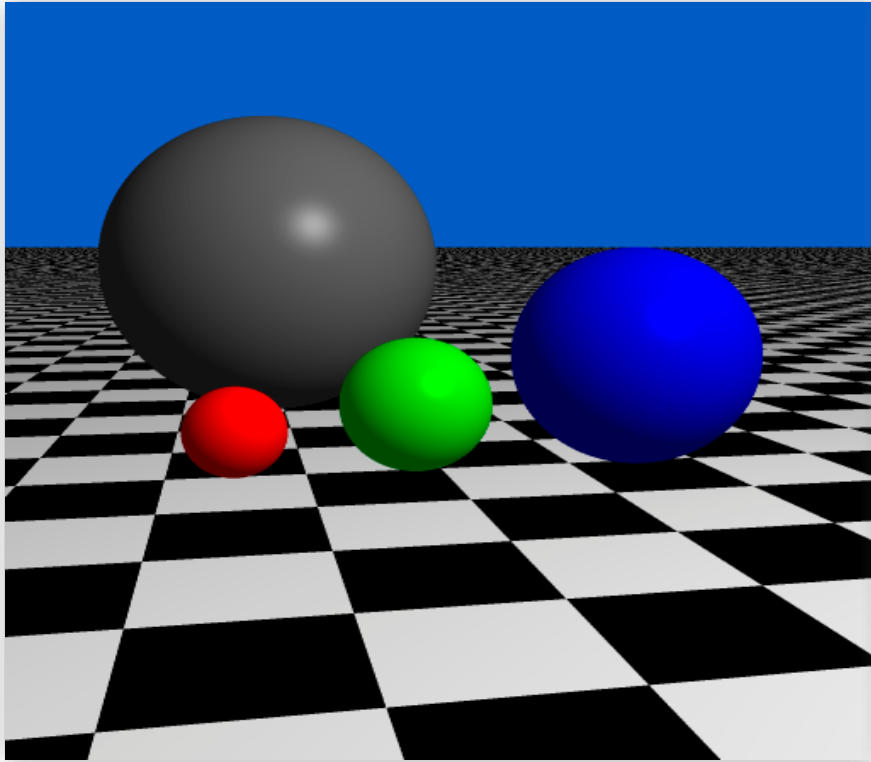
Light, Surfaces, and Computer Imaging



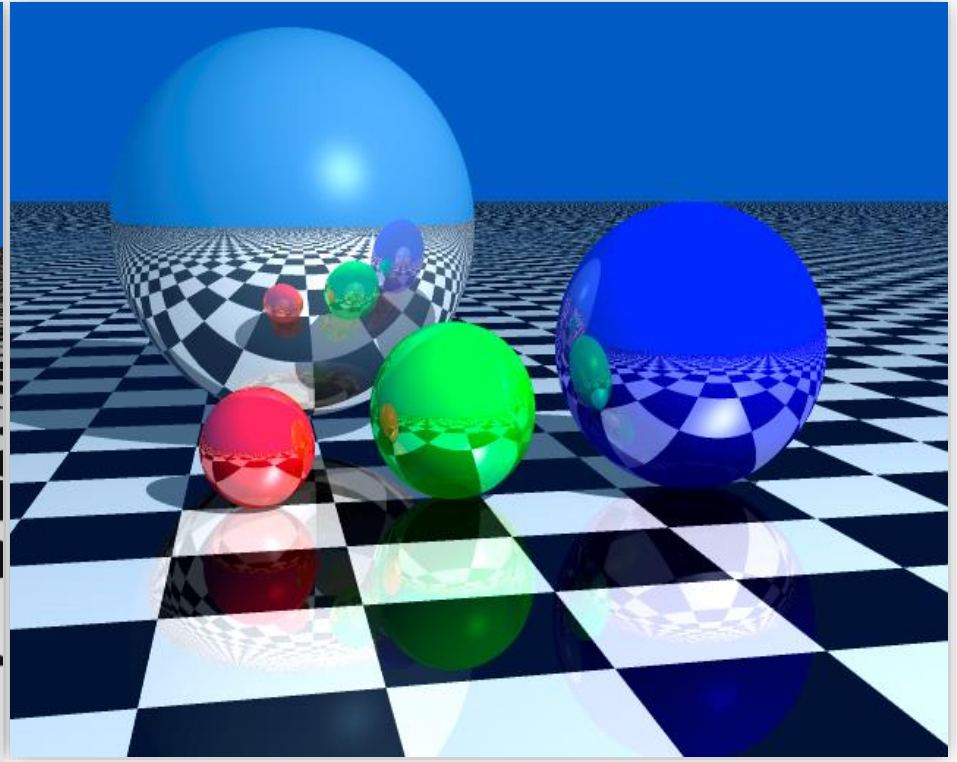
Local vs Global Rendering

- 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 real time graphics, we are happy if things “look right”
 - Exist many techniques for approximating global effects

Local Effect



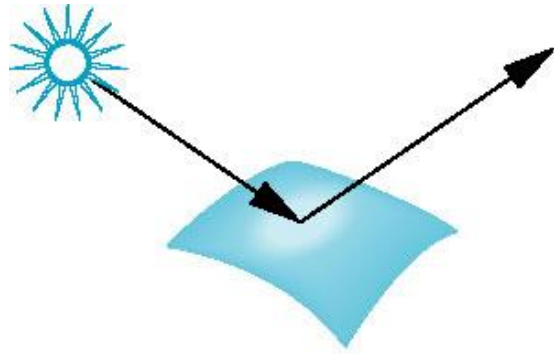
Global Effect



Light-Material Interaction

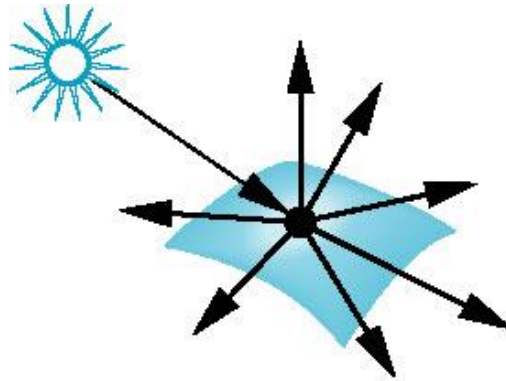
- Light that strikes an object is **partially absorbed** and **partially scattered** (reflected)
- The amount reflected determines the **color** and **brightness** of the object
 - A surface appears red under white light because the red component of the light is reflected and the rest is absorbed
- The **reflected light** is scattered in a manner that depends on the **smoothness** and **orientation** of the surface

Light-Material Interactions



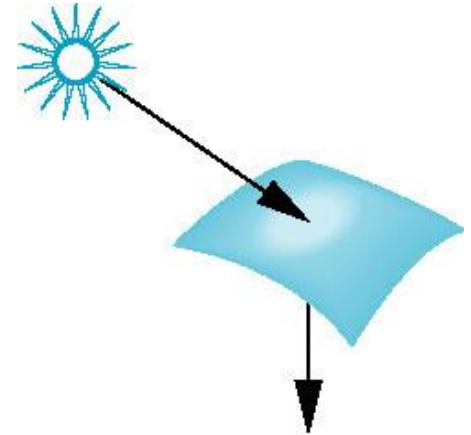
(a)

Specular surface



(b)

Diffuse surface

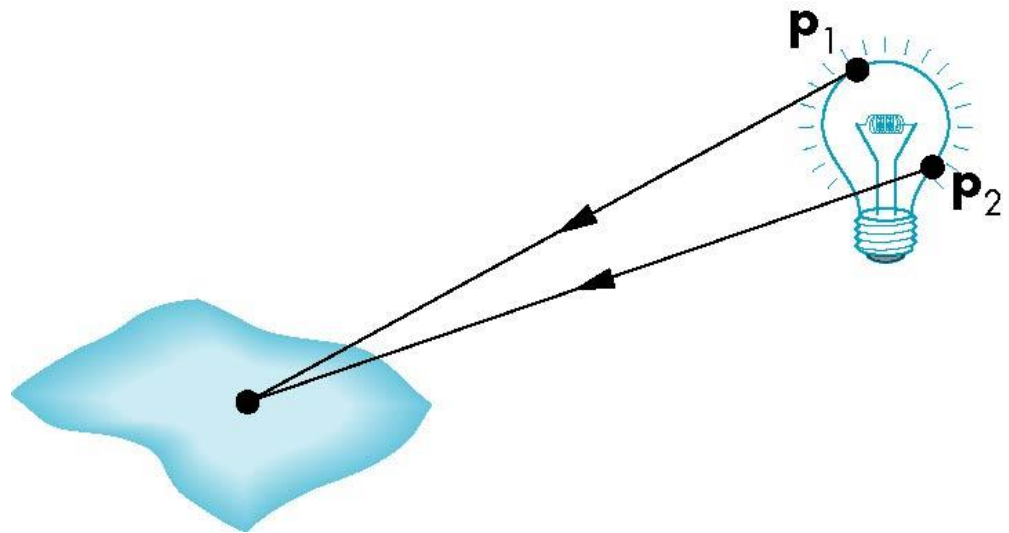
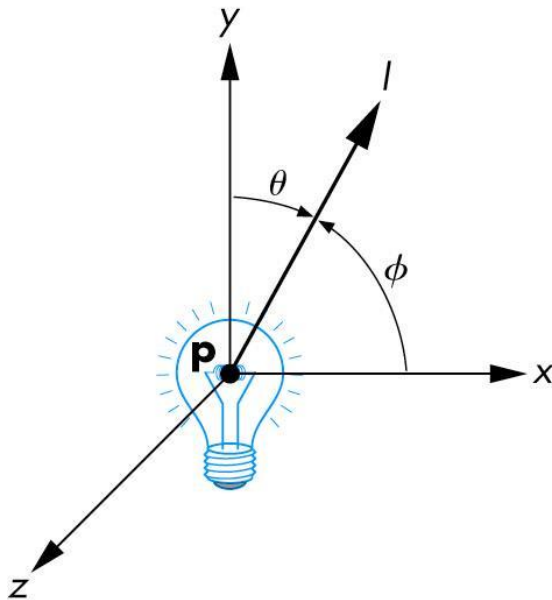


(c)

Translucent surface

Light Sources

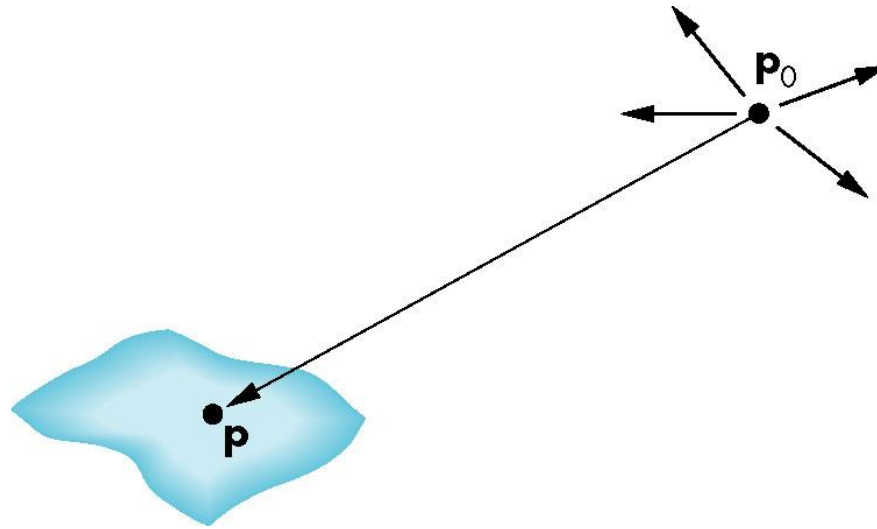
General light sources are difficult to work with because we must integrate light coming from all points on the source



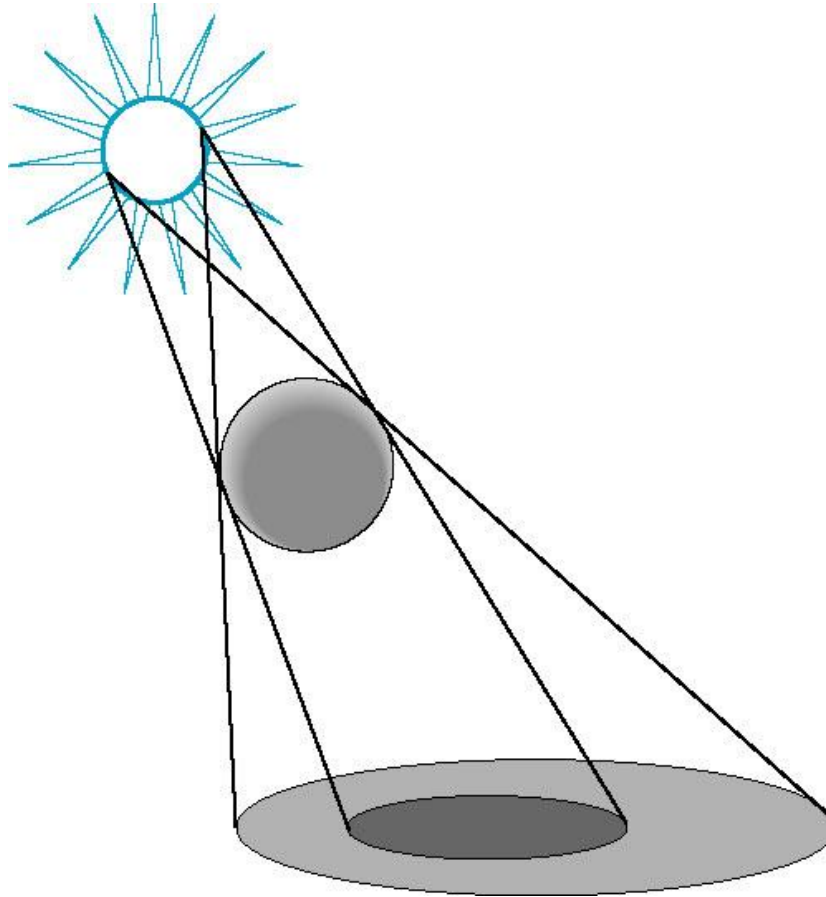
Simple Light Sources

- Point source
 - Model with position and color
 - Distant source = infinite distance away (parallel)
- Spotlight
 - Restrict light from ideal point source
- Ambient light
 - Same amount of light everywhere in scene
 - Can model contribution of many sources and reflecting surfaces

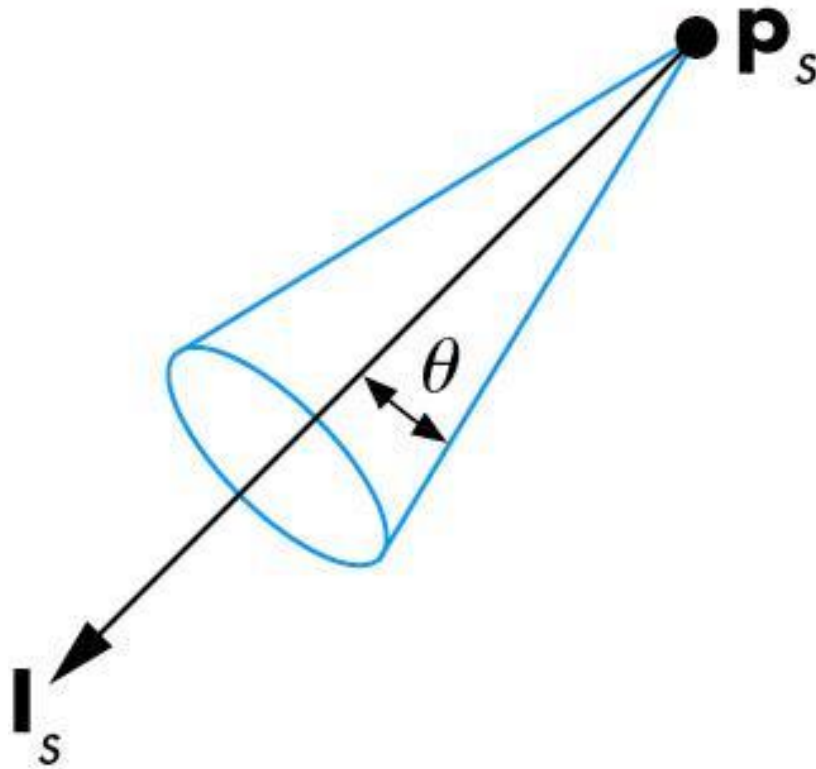
Point source illuminating a surface



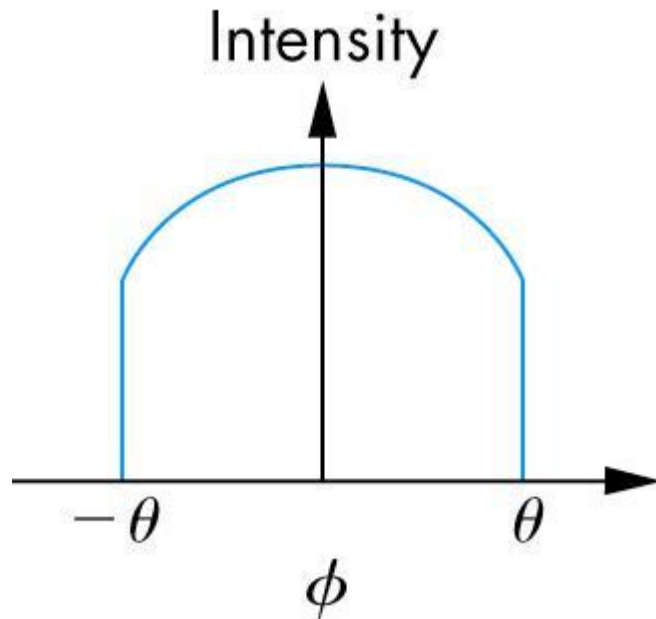
Shadow created by finite-size light source



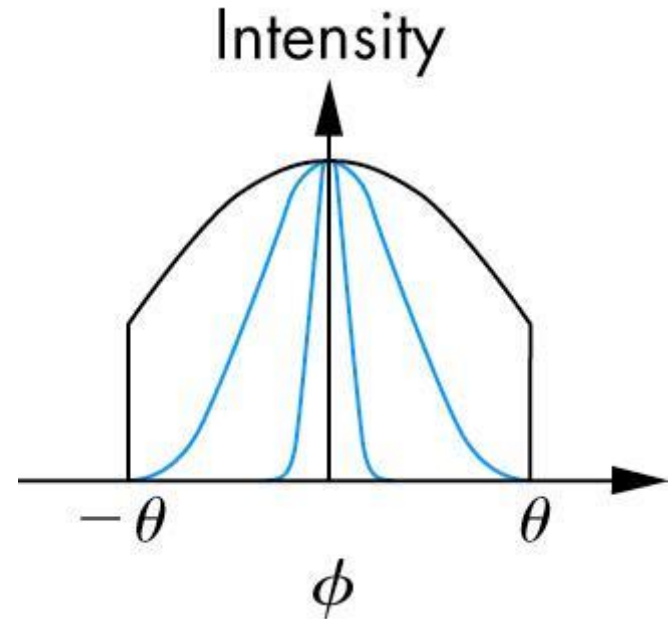
Spotlight



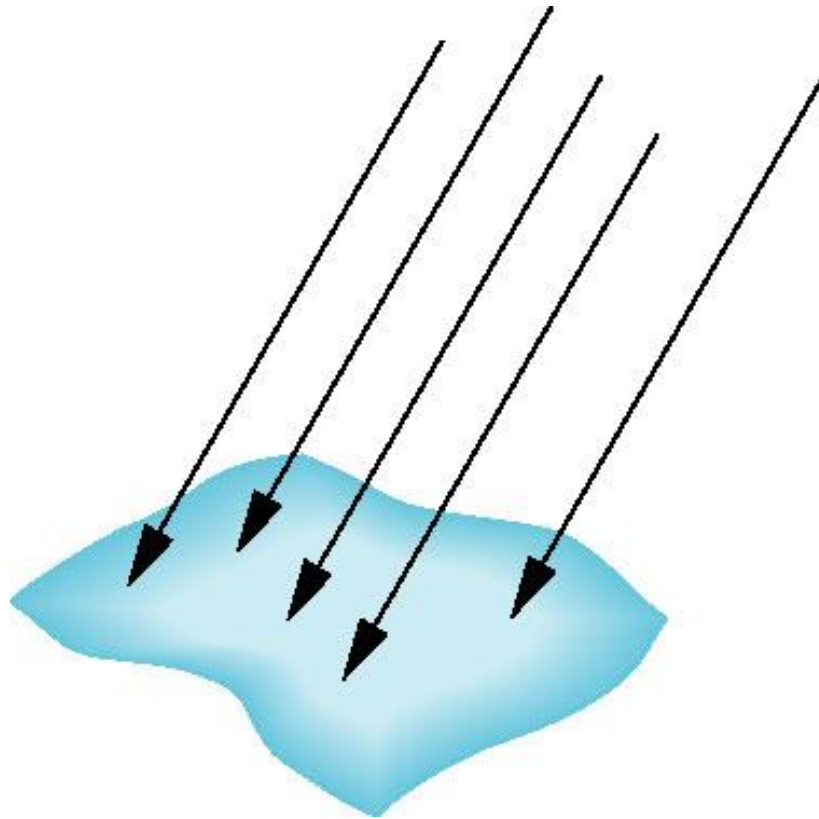
Attenuation of a spotlight



Spotlight exponent

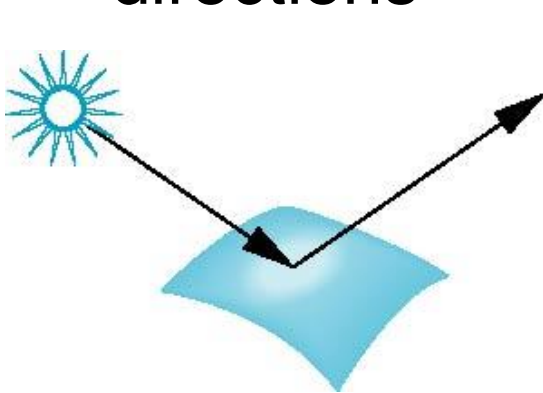


Parallel Light Source

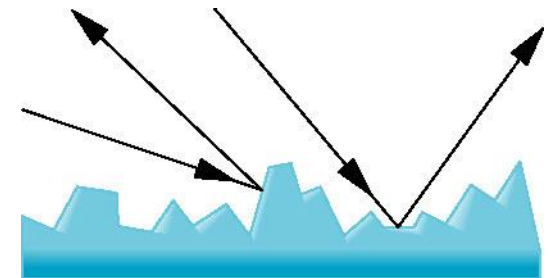
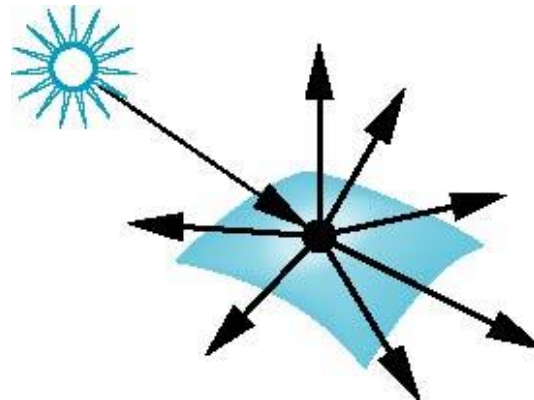


Surface Types

- 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



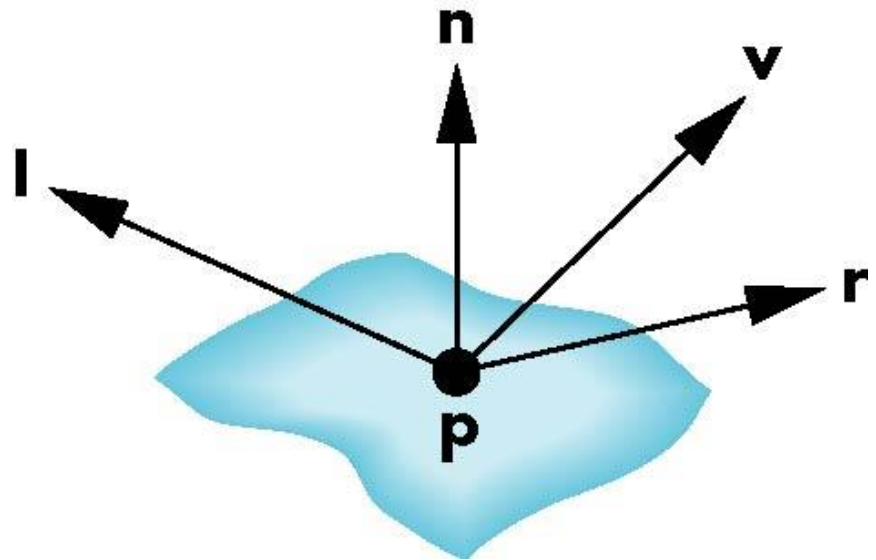
smooth surface



rough surface

Phong Model

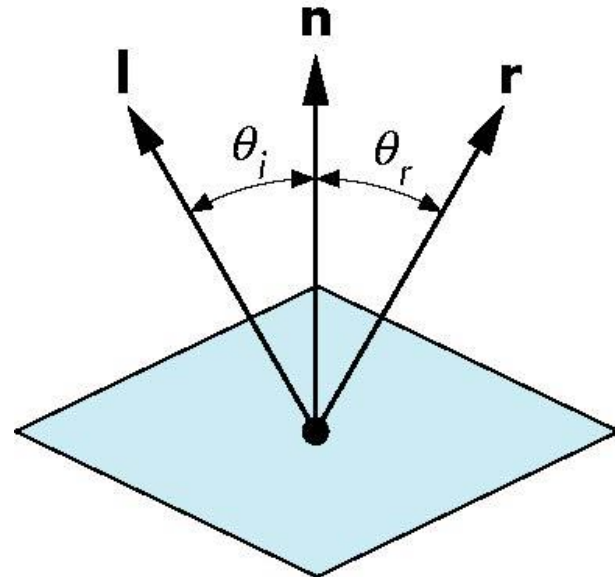
- A **simple model** that can be computed **rapidly**
- Has three components
 - Diffuse
 - Specular
 - Ambient
- Uses four vectors
 - To source
 - To viewer
 - Normal
 - Perfect reflector



Ideal Reflector

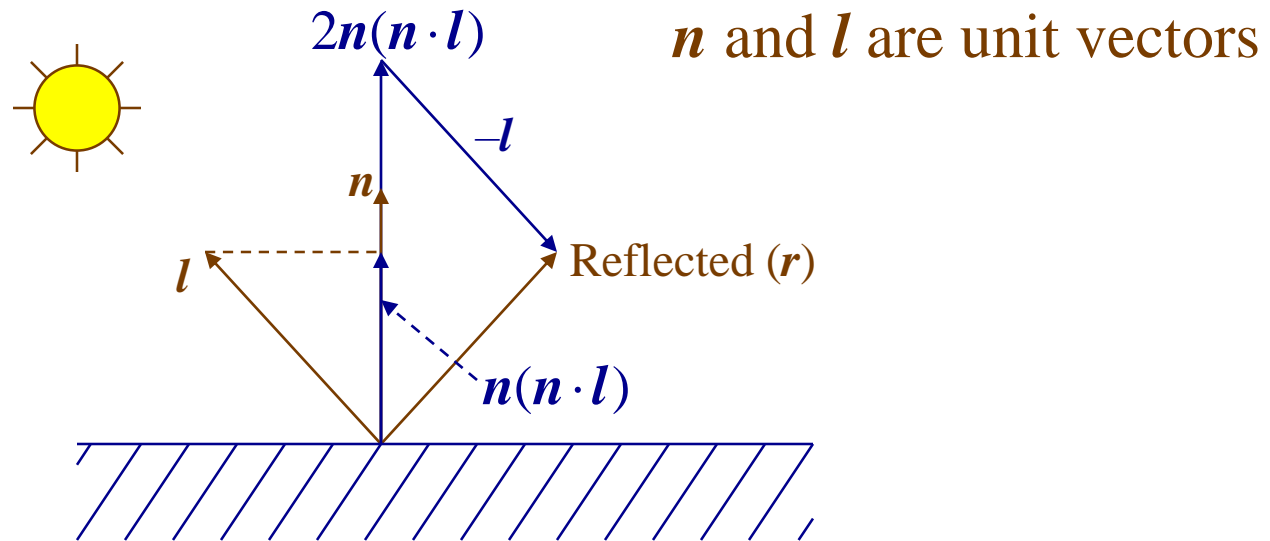
- Normal is determined by local orientation
- Angle of incidence = angle of reflection
- The three vectors must be coplanar

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



Phong Illumination: Computing r

- How can we compute the reflection vector r ?

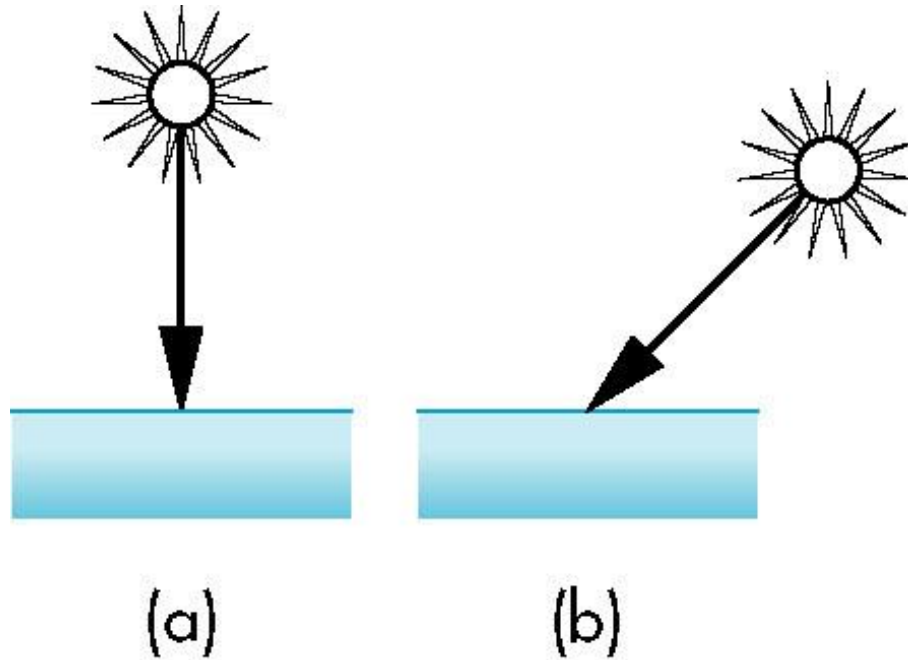


$$r = 2n(n \cdot l) - l$$

Lambertian Surface

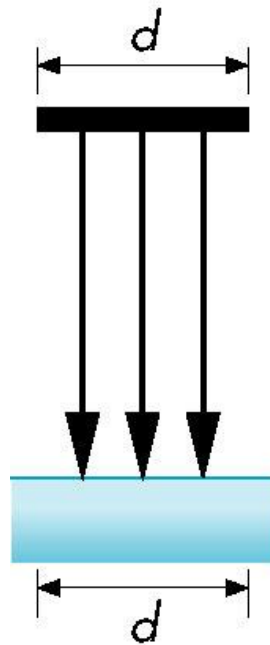
- **Perfectly diffuse** reflector
- Light scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
 - reflected light $\sim \cos \theta_i$
 - $\cos \theta_i = \mathbf{l} \cdot \mathbf{n}$ if vectors normalized
 - There are also **three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

Diffuse Reflection

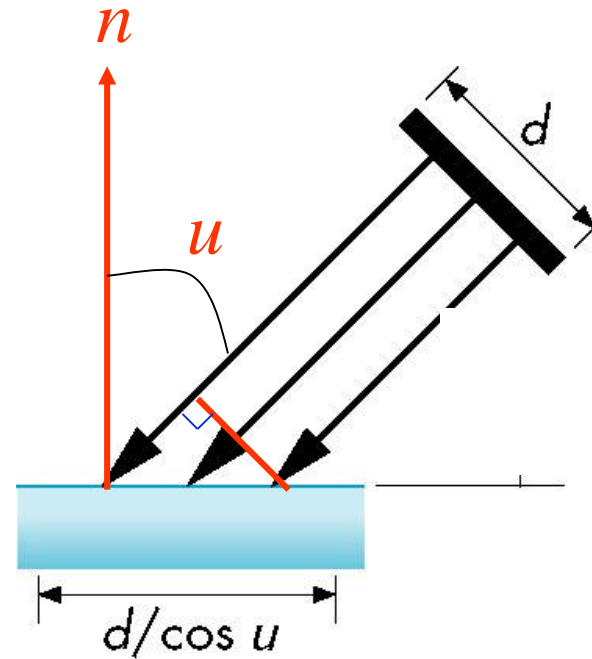


Illumination of a diffuse surface.
(a) at noon. (b) In the afternoon.

Diffuse Reflection



(a)



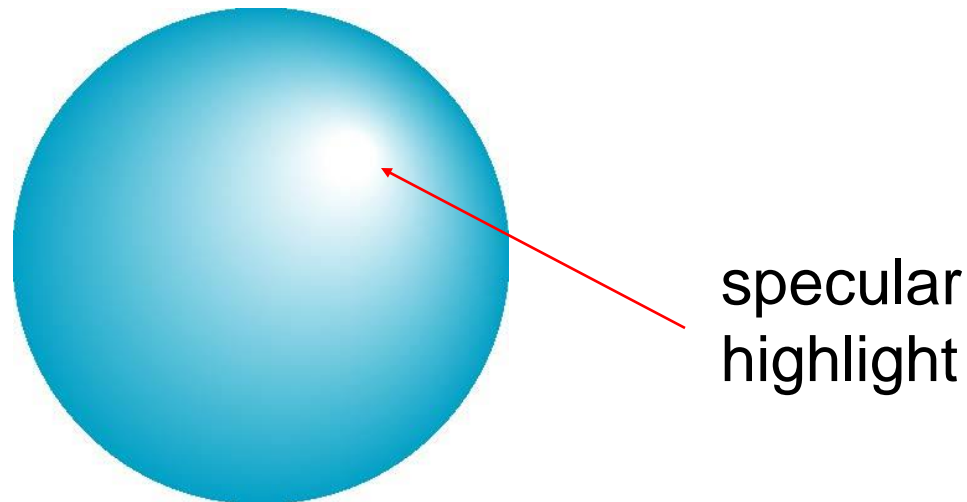
(b)

Vertical contributions by Lambert's Law.

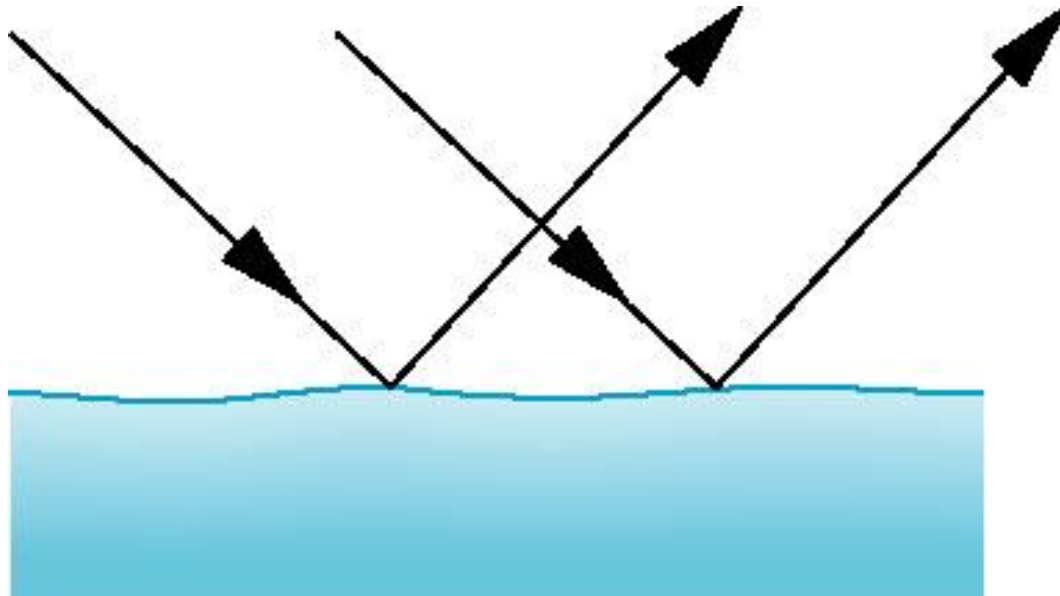
(a) At noon. (b) In the afternoon.

Specular Surfaces

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

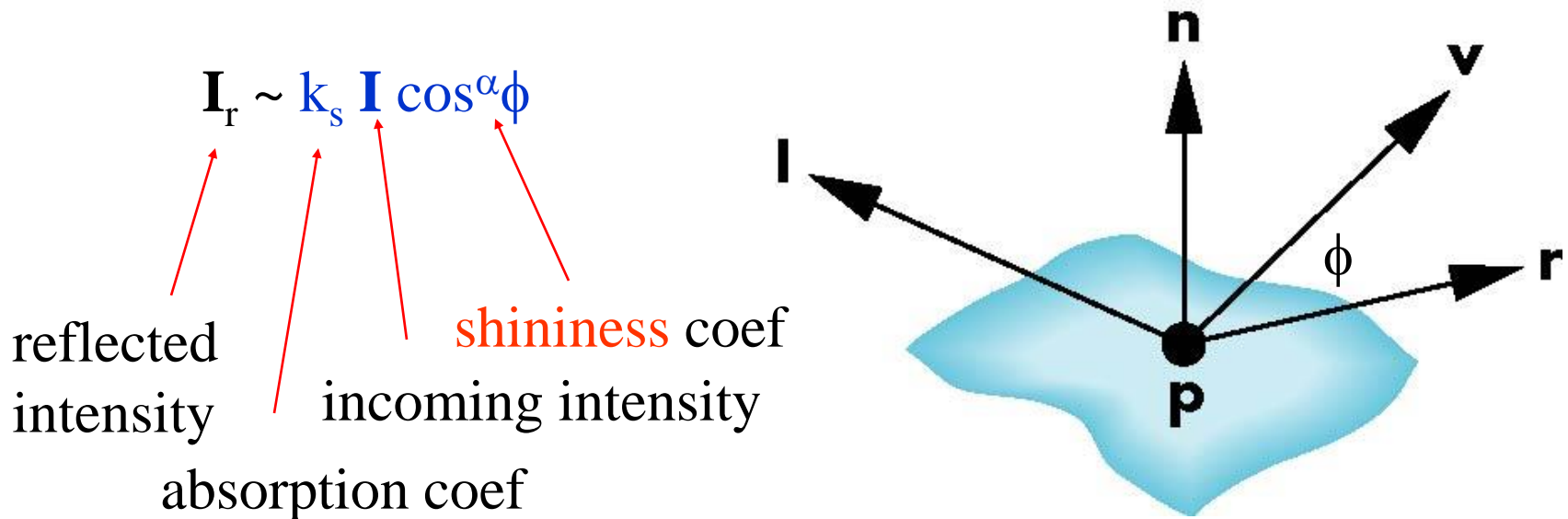


Specular Surfaces



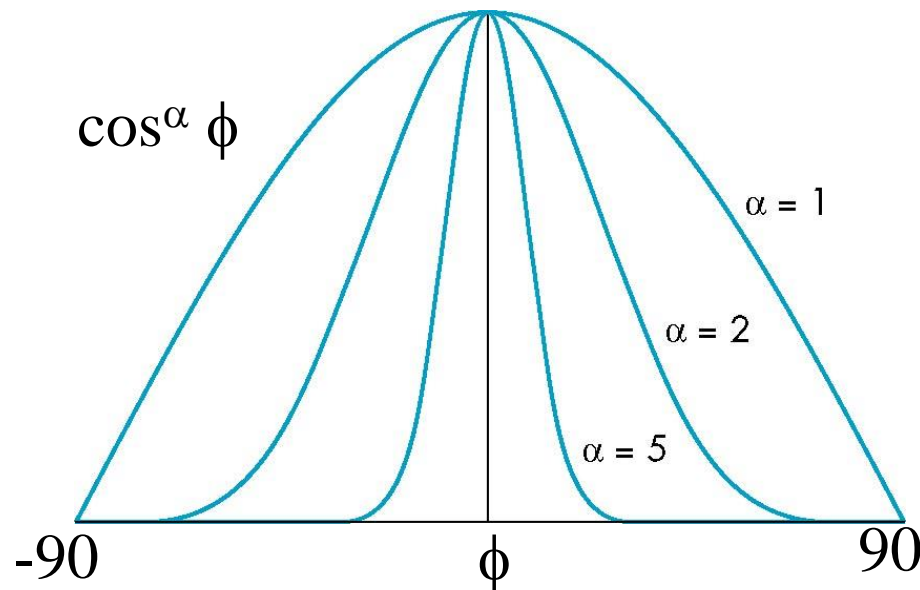
Modeling Specular Reflections

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



The Shininess Coefficient

- Values of α between 100 and 200 correspond to **metals**
- Values between 5 and 10 give surface that look like **plastic**



Shading II

Objectives

- Continue discussion of shading
- Introduce modified Phong model
- Consider computation of required vectors

Ambient Light

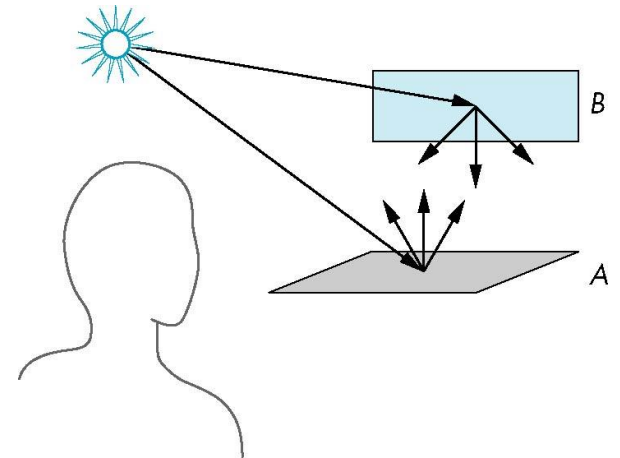
- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment
- Amount and color depend on both the color of the light(s) and the material properties of the object
- Add $k_a I_a$ to diffuse and specular terms

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
- We can add a factor of the form $1/(ad + bd + cd^2)$ to the **diffuse** and **specular** terms
- 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

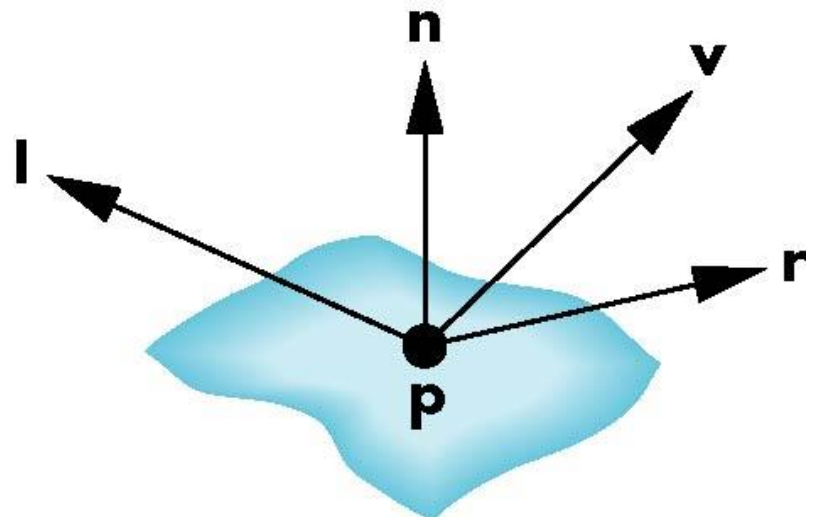
- Material properties match light source properties
 - Nine absorption coefficients
 - k_{dr} , k_{dg} , k_{db} , k_{sr} , k_{sg} , k_{sb} , k_{ar} , k_{ag} , k_{ab}
 - Shininess coefficient α

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**

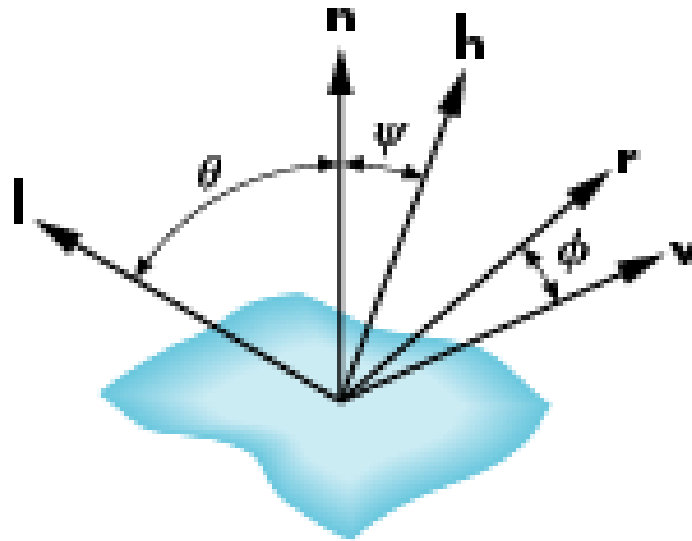


Modified Phong Model

- The *specular term* in the Phong model is *problematic* because it requires the calculation of a **new reflection vector** and **view vector** for each vertex
- Blinn suggested an *approximation* using the **halfway vector** that is more efficient

The Halfway Vector

- ***h*** is normalized vector halfway between ***l*** and ***v***
and ***v*** $\mathbf{h} = (\mathbf{l} + \mathbf{v}) / |\mathbf{l} + \mathbf{v}|$



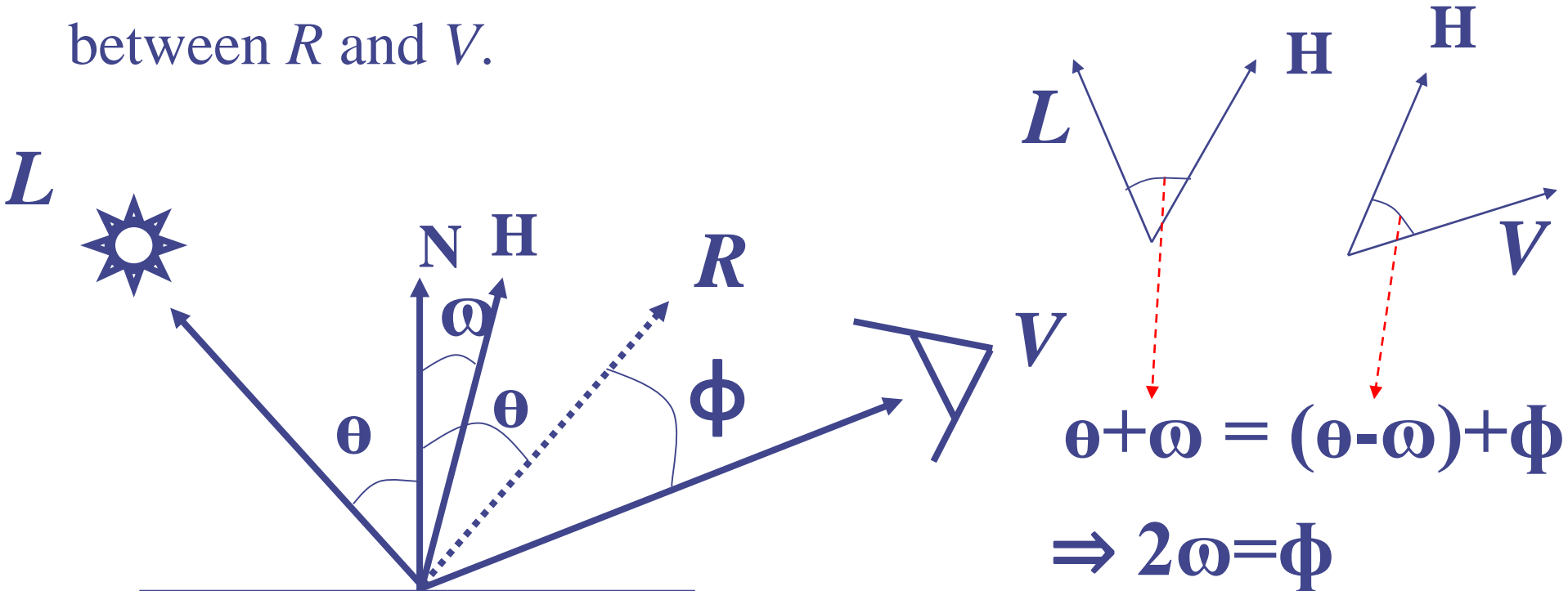
Using the halfway vector

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

Using the halfway vector

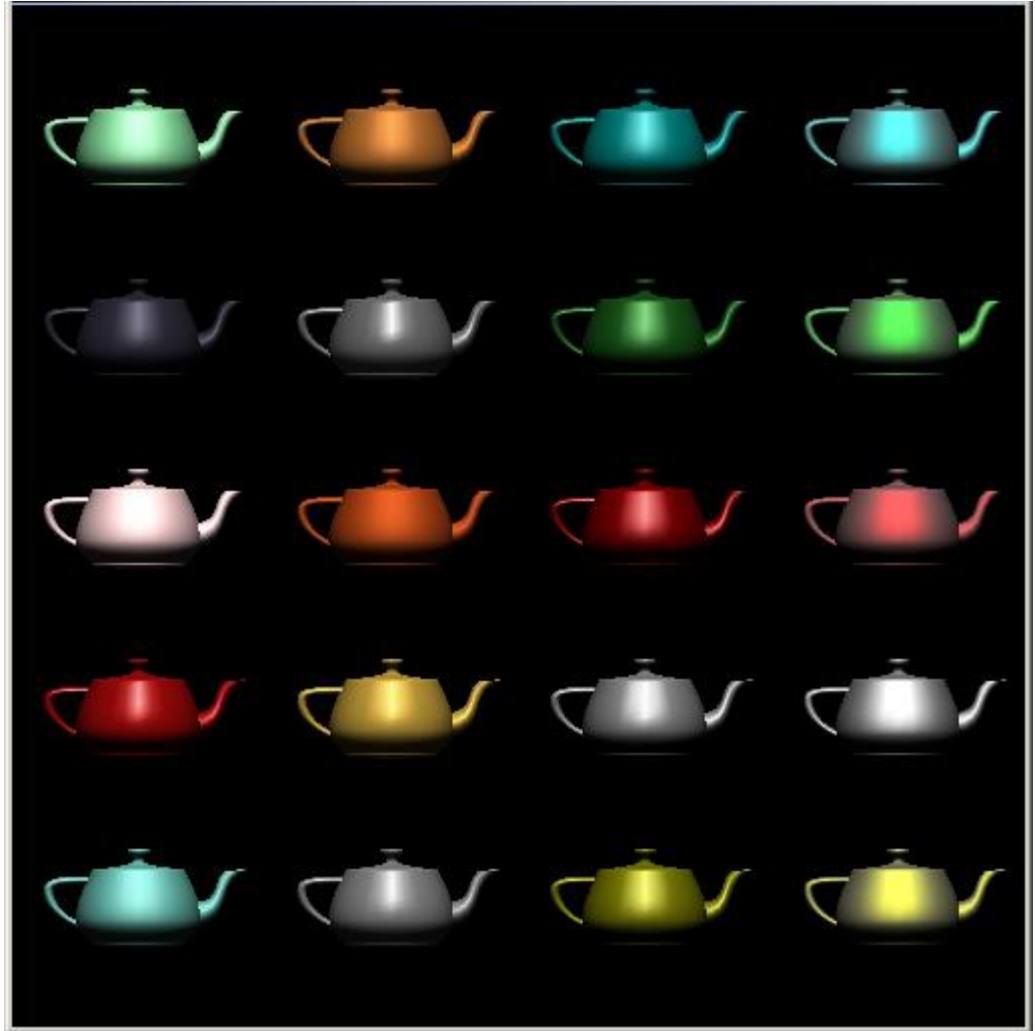
$(R \cdot V)^p$ can be replaced by $(N \cdot H)^p$
where $H = (L + V) / 2$.

The angle between N and H is **half** the size of the angle between R and V .



Example

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



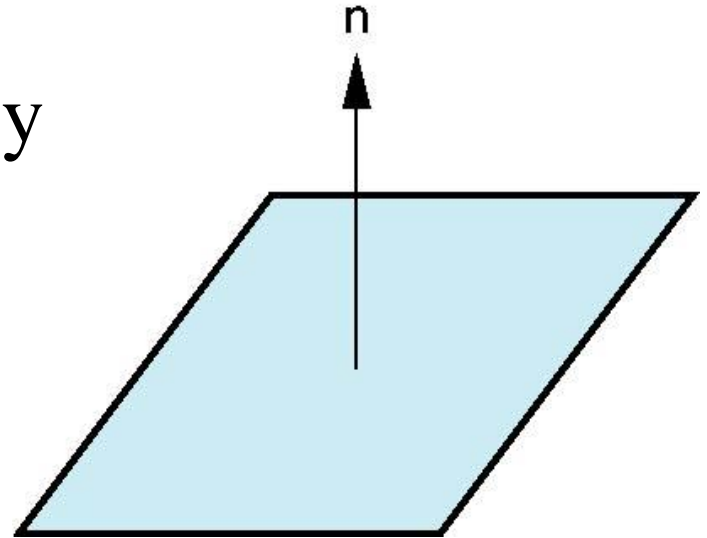
Computation of Vectors

- \mathbf{l} and \mathbf{v} are specified by the application
- Can compute \mathbf{r} from \mathbf{l} and \mathbf{n}
- Problem is determining \mathbf{n}
- For simple surfaces \mathbf{n} can be determined but how we determine \mathbf{n} differs depending on underlying representation of surface
- OpenGL leaves determination of normal to application
 - Exception for GLU quadrics and Bezier surfaces (Chapter 11)

Plane Normals

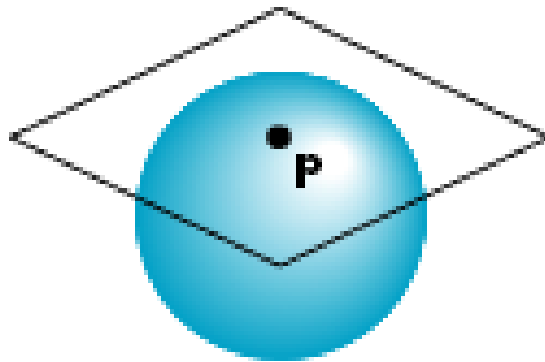
- Equation of plane: $ax+by+cz+d=0$
- From Chapter 4 we know that plane is determined by three points p_0, p_1, p_2 or normal \mathbf{n} and p_0
- Normal can be obtained by

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



Normal to Sphere

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



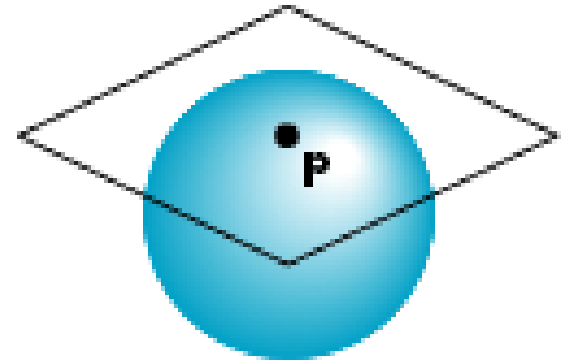
Parametric Form

- For sphere

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

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

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



- Tangent plane determined by vectors

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

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

- Normal given by cross product

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

General Case

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

Shading in OpenGL

Objectives

- Introduce the OpenGL shading functions
- Discuss polygonal shading
 - Flat
 - Smooth
 - Gouraud

Steps in OpenGL shading

1. Enable shading and select model
2. Specify normals
3. Specify material properties
4. Specify lights

Normals

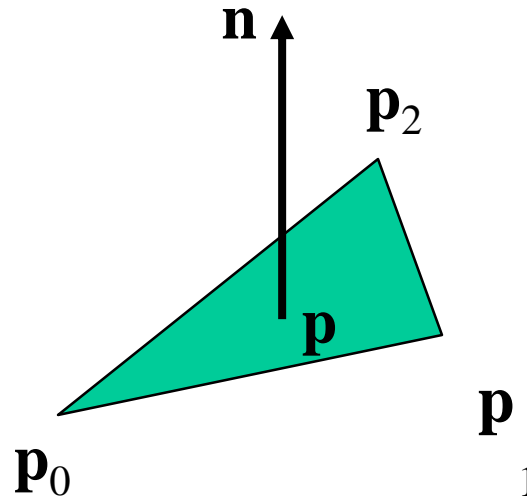
- In OpenGL the normal vector is part of the state
- Set by **glNormal* ()**
 - **glNormal3f (x, y, z) ;**
 - **glNormal3fv (p) ;**
- Usually we want to set the normal to have unit length so cosine calculations are correct
 - Length can be affected by transformations
 - Note that scaling does not preserved length
 - **glEnable (GL_NORMALIZE)** allows for autonormalization at a performance penalty

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

Enabling Shading

- Shading calculations are enabled by
 - **glEnable(GL_LIGHTING)**
 - Once lighting is enabled, glColor() ignored
- Must enable each light source individually
 - **glEnable(GL_LIGHTi) i=0,1.....**
- Can choose light model parameters
 - **glLightModeli(parameter, GL_TRUE)**
 - **GL_LIGHT_MODEL_LOCAL_VIEWER** do not use simplifying distant viewer assumption in calculation
 - **GL_LIGHT_MODEL_TWO_SIDED** shades both sides of polygons independently

Defining a Point Light Source

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

```
GL float diffuse0[]={1.0, 0.0, 0.0, 1.0};
GL float ambient0[]={1.0, 0.0, 0.0, 1.0};
GL float specular0[]={1.0, 0.0, 0.0, 1.0};
GLfloat light0_pos[]={1.0, 2.0, 3.0, 1.0};

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, light0_pos);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
glLightfv(GL_LIGHT0, GL_SPECULAR, specular0);
```


Distance and Direction

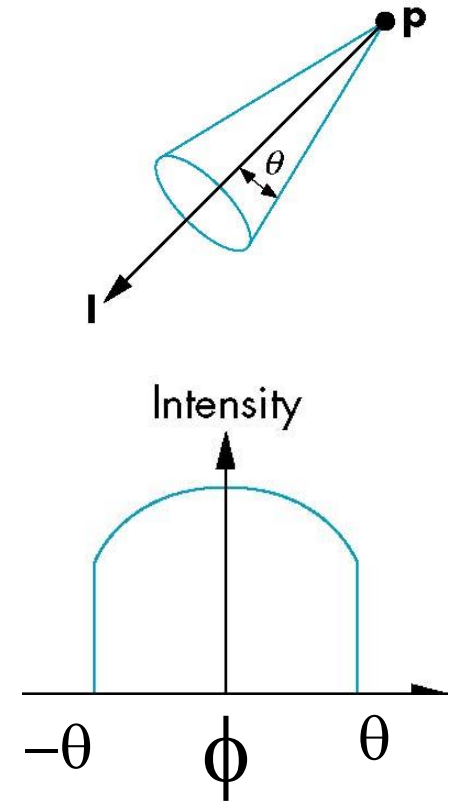
- The source colors are specified in RGBA
- The **position** is given in **homogeneous coordinates**
 - If **w = 1.0**, we are specifying a finite location
 - If **w = 0.0**, we are specifying a parallel source with the given direction vector
- The coefficients in the distance terms are by default **a=1.0** (constant terms), **b=c=0.0** (linear and quadratic terms). Change by

a= 0.80;

glLightf(GL_LIGHT0, GLCONSTANT_ATTENUATION, a);

Spotlights

- Use **glLightfv** to set
 - Direction **GL_SPOT_DIRECTION**
 - Cutoff **GL_SPOT_CUTOFF**
 - Attenuation **GL_SPOT_EXPONENT**
 - 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
- OpenGL also allows a global ambient term that is often helpful for testing
 - `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`

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 can
 - 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

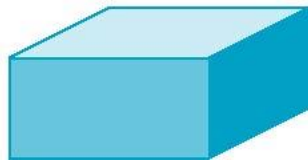
Material Properties

- Material properties are also part of the OpenGL state and match the terms in the modified Phong model
- Set by `glMaterialv()`

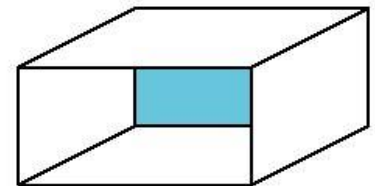
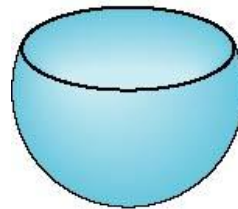
```
GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
GLfloat diffuse[] = {1.0, 0.8, 0.0, 1.0};
GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat shine = 100.0
glMaterialf(GL_FRONT, GL_AMBIENT, ambient);
glMaterialf(GL_FRONT, GL_DIFFUSE, diffuse);
glMaterialf(GL_FRONT, GL_SPECULAR, specular);
glMaterialf(GL_FRONT, GL_SHININESS, shine);
```

Front and Back Faces

- The **default** is shade only **front faces** which works correctly for convex objects
- If we set two sided lighting, OpenGL will shade both sides of a surface
- Each side can have its own properties which are set by using **GL_FRONT**, **GL_BACK**, or **GL_FRONT_AND_BACK** in **glMaterialf**



back faces not visible



back faces visible

Emissive Term

- We can simulate a **light source** in OpenGL by giving a material an **emissive component**
- This component is unaffected by any sources or transformations

```
GLfloat emission[] = {0.0, 0.3, 0.3, 1.0};  
glMaterialf(GL_FRONT, GL_EMISSION, emission);
```

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
- Later we will enable blending and use this feature

Efficiency

- Because material properties are part of the state, if we **change materials** for many surfaces, we can **affect performance**
- We can make the code cleaner by defining a material structure and setting all materials during initialization

```
typedef struct materialStruct {  
    GLfloat ambient[4];  
    GLfloat diffuse[4];  
    GLfloat specular[4];  
    GLfloat shininess;  
} MaterialStruct;
```

- We can then select a material by a pointer

Polygonal Shading

- Shading calculations are done for each vertex
 - Vertex colors become vertex shades
- By default, vertex shades are interpolated across the polygon
 - `glShadeModel (GL_SMOOTH) ;`
- If we use `glShadeModel (GL_FLAT) ;` the color at the first vertex will determine the shade of the whole polygon

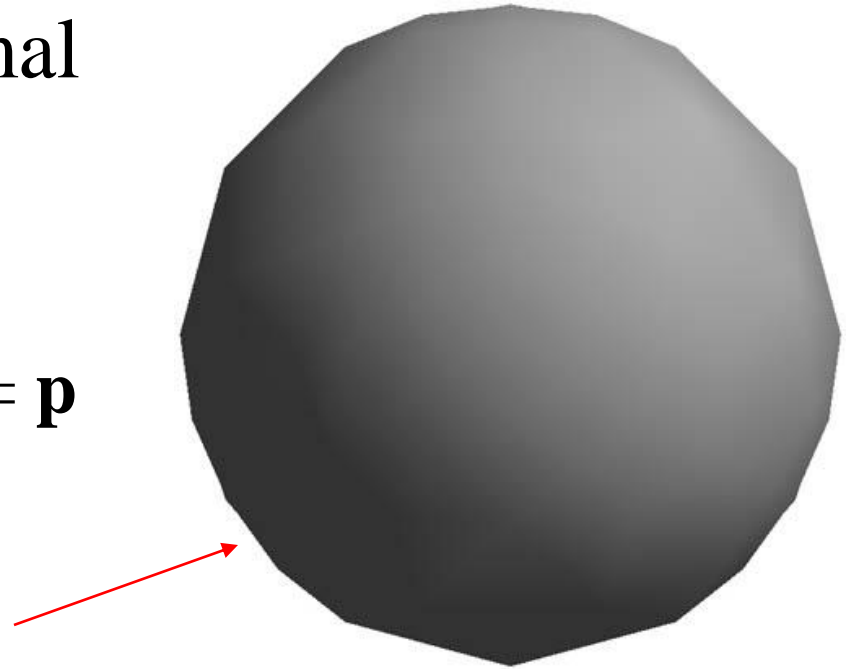
Polygon Normals

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



Smooth Shading

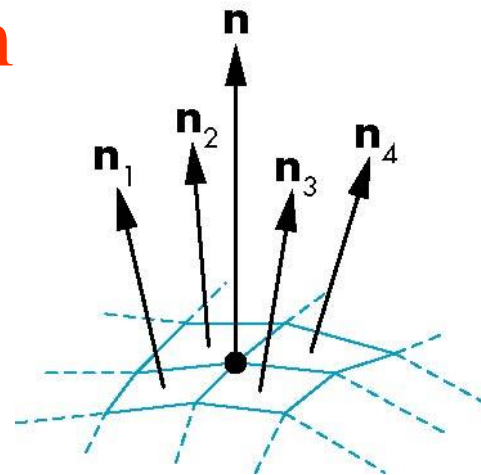
- We can set a new normal at each vertex
- Easy for sphere model
 - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*



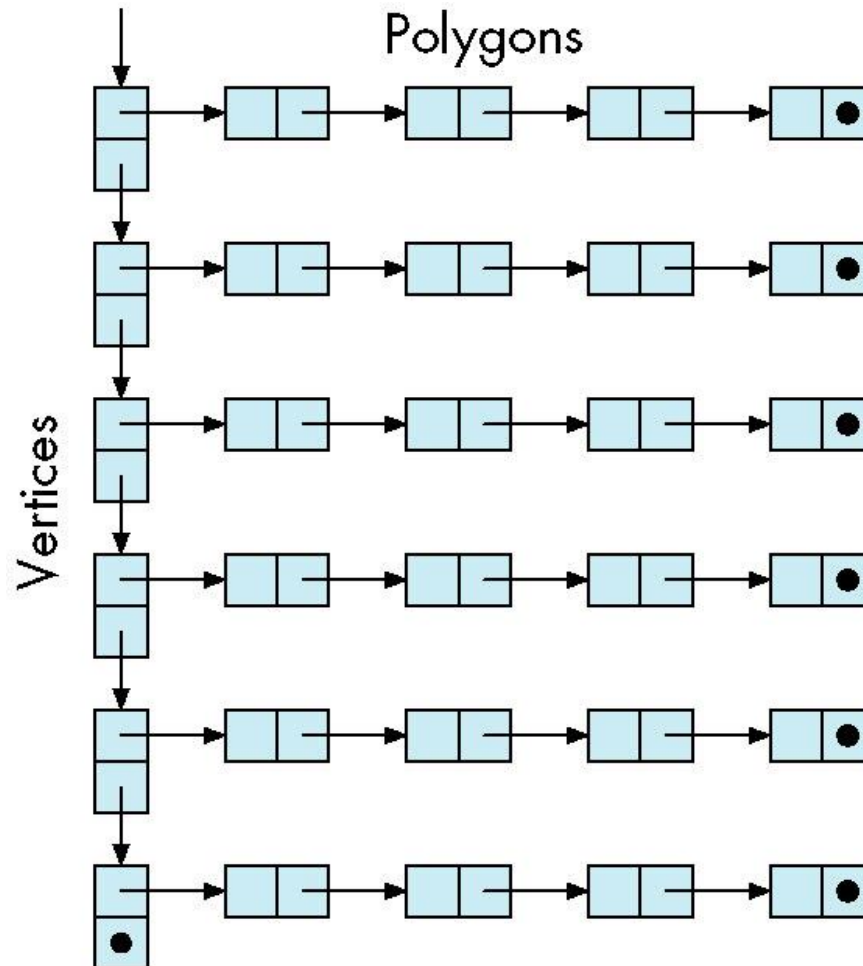
Mesh Shading

- The previous example is **not general** because we knew **the normal at each vertex** analytically
- For polygonal models, **Gouraud** proposed we use the **average of the norm** mesh vertex

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



Mesh Data Structure



Gouraud and Phong Shading

- Gouraud Shading

- Find **average normal** at each vertex (vertex normals)
- Apply **modified Phong model** at each vertex
- Interpolate vertex shades across each polygon

- Phong shading

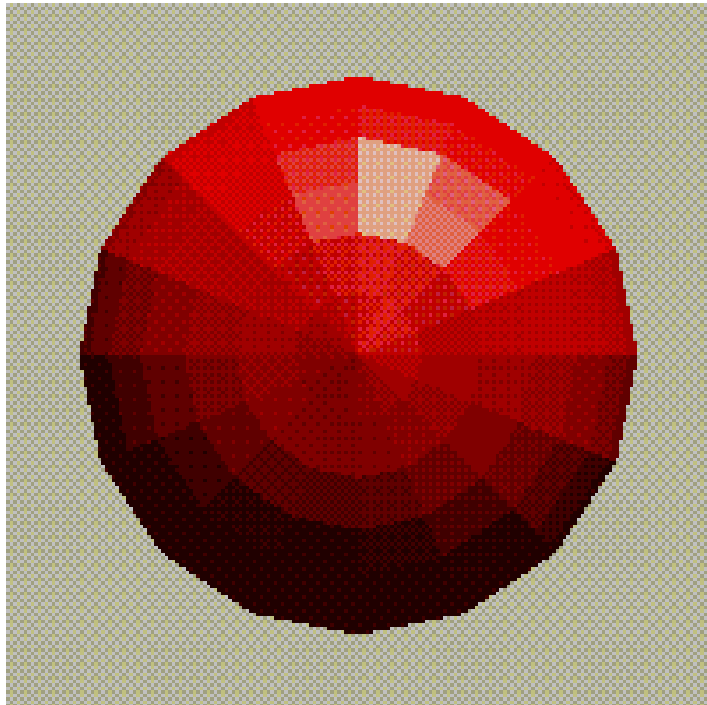
- Find vertex normals
- Interpolate vertex normals across edges
- Interpolate edge normals across polygon
- Apply **modified Phong model** at each fragment

Types of Shading

- There are several well-known / commonly-used shading methods
 - Flat shading
 - Gouraud shading
 - Phong shading

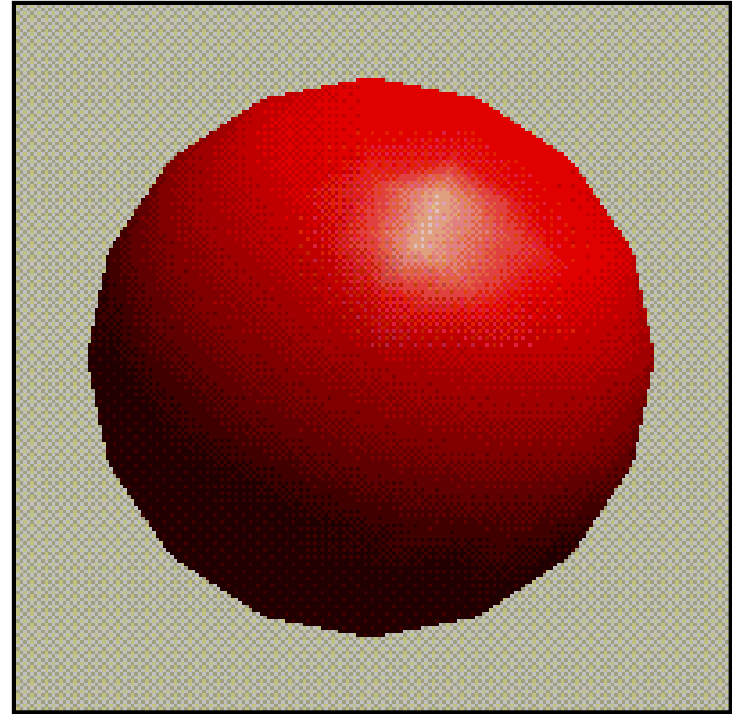
Shading Schemes

Flat Shading: same shade to entire polygon



Shading Schemes

Gouraud Shading:
smoothly blended
intensity across
each polygon



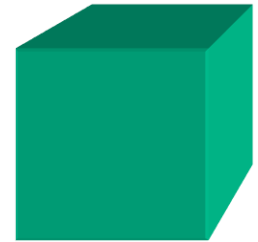
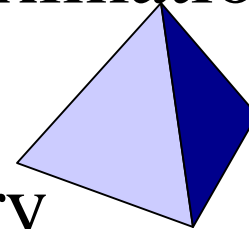
Shading Schemes

Phong Shading:
interpolated
normals to
compute intensity
at each point



Flat Shading

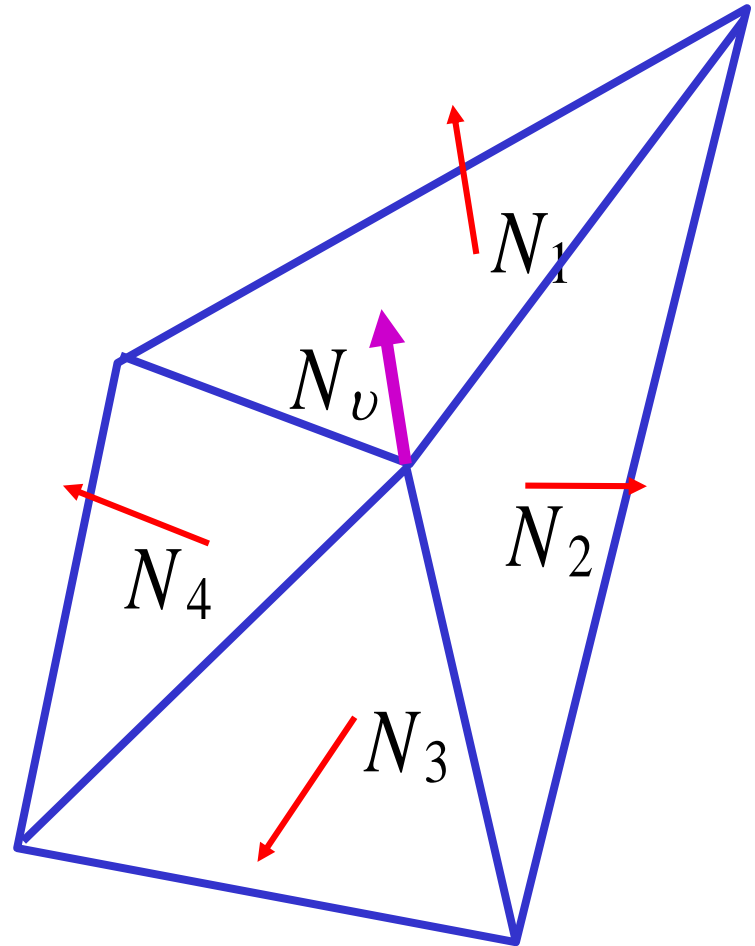
- Compute 1 normal for the polygon
- Assumes **light** (and viewer if using Phong illumination model) **are at infinity**
- Assumes polygon exactly represents the actual surface, not an approximation:
 - i.e. cube vs. cylinder
- No interpolation is necessary
- Faceting occurs when used on approximate surfaces



Flat (Cosine) Shading

- Compute constant shading function, over each polygon, based on simple cosine term
- Same normal and light vector across whole polygon
- Constant shading for polygon

$$\sim N \cdot L$$



Flat (Cosine) Shading

$$\begin{aligned} I &= I_p k_d \cos(\theta) \\ &= I_p k_d N \cdot L, \quad \text{for unit } N, L \end{aligned}$$

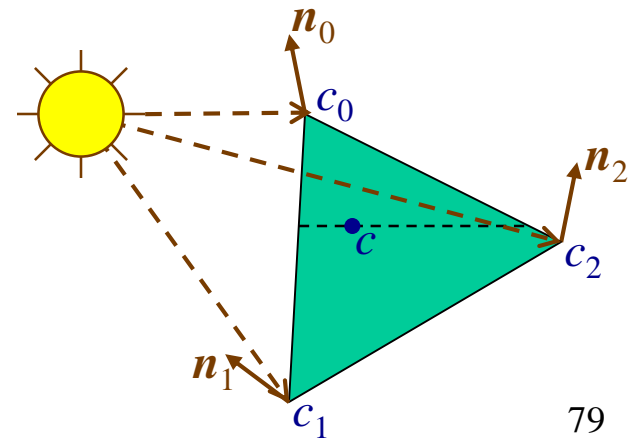
Where,

I_p = intensity of point light source

k_d = diffuse reflection coefficient

Gouraud Shading

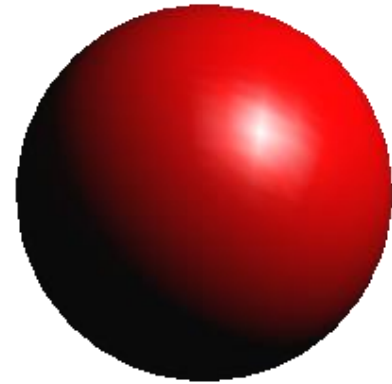
- Compute the normal, \mathbf{n}_i , and color at each vertex:
 - If \mathbf{n}_i not already provided \rightarrow average of the normal of the faces that share the vertex
 - Compute color at each vertex using illumination model
- Interpolate colors across the projected polygon during scan conversion
- Assumes the polygons approximate the surface
- Problems?



Flat vs. Gouraud Shading



`glShadeModel(GL_FLAT)`



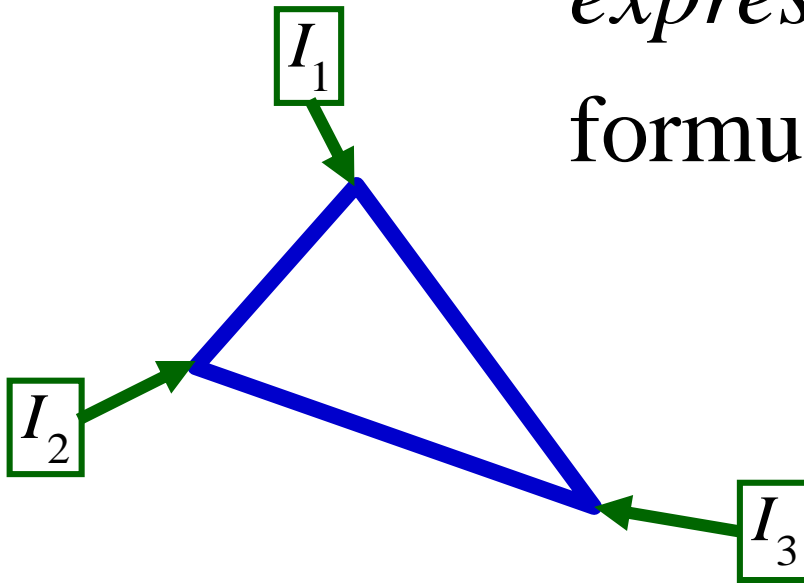
`glShadeModel(GL_SMOOTH)`

Flat - Determine that each face has a single normal, and color the entire face a single value, based on that normal.

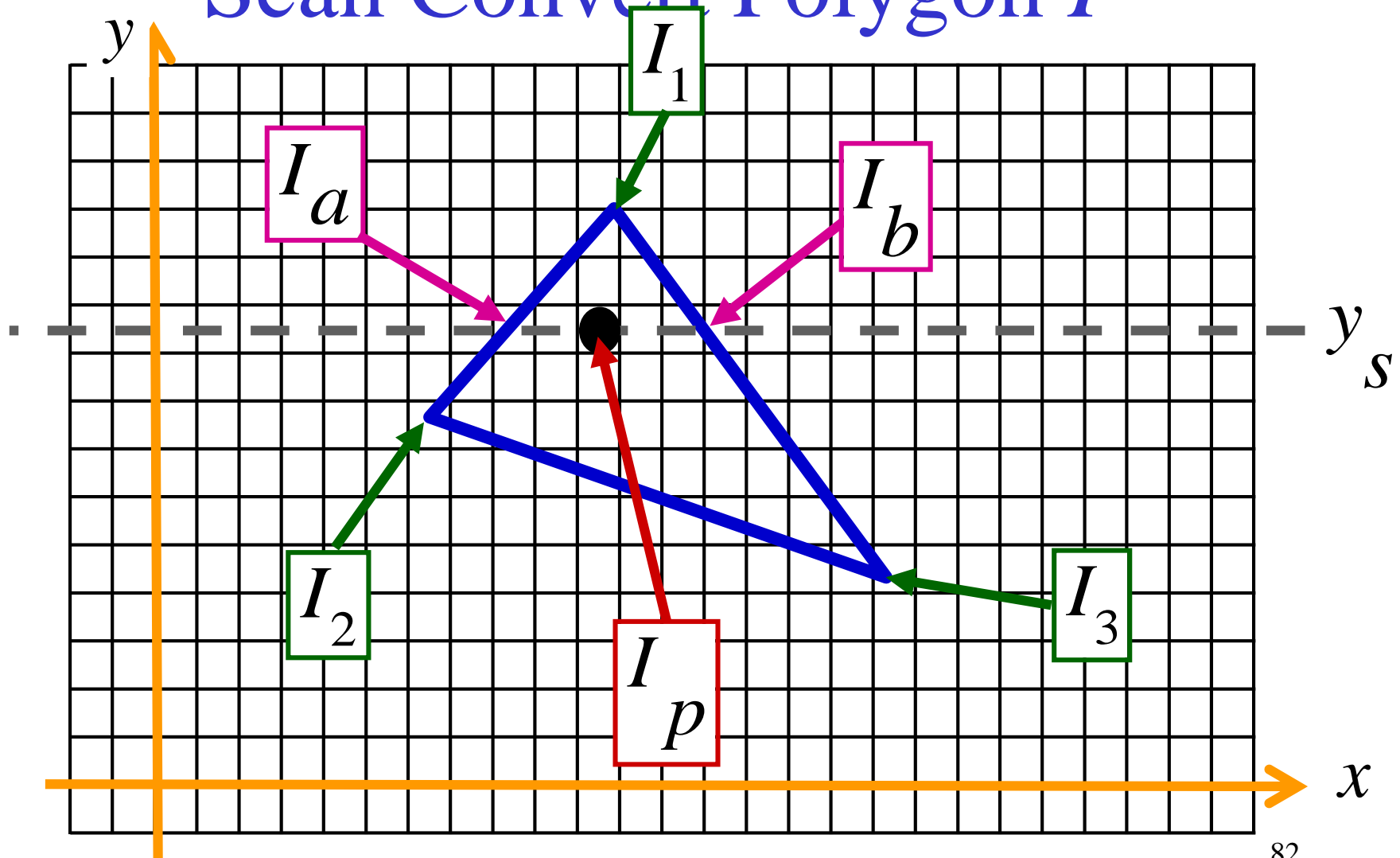
Gouraud – Determine the color at each vertex, using the normal at that vertex, and interpolate linearly for the pixels between the vertex locations.

Gouraud Shading: Intensity Interpolation

I_1, I_2, I_3 : Compute by direction evaluation of *illumination expression*, whichever formula is being used

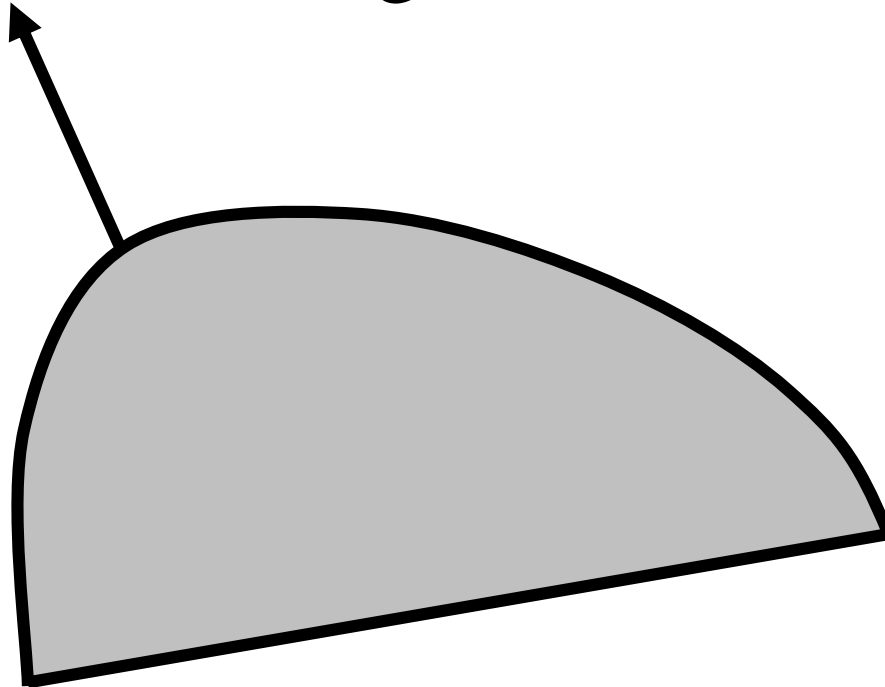


Scan Convert Polygon P

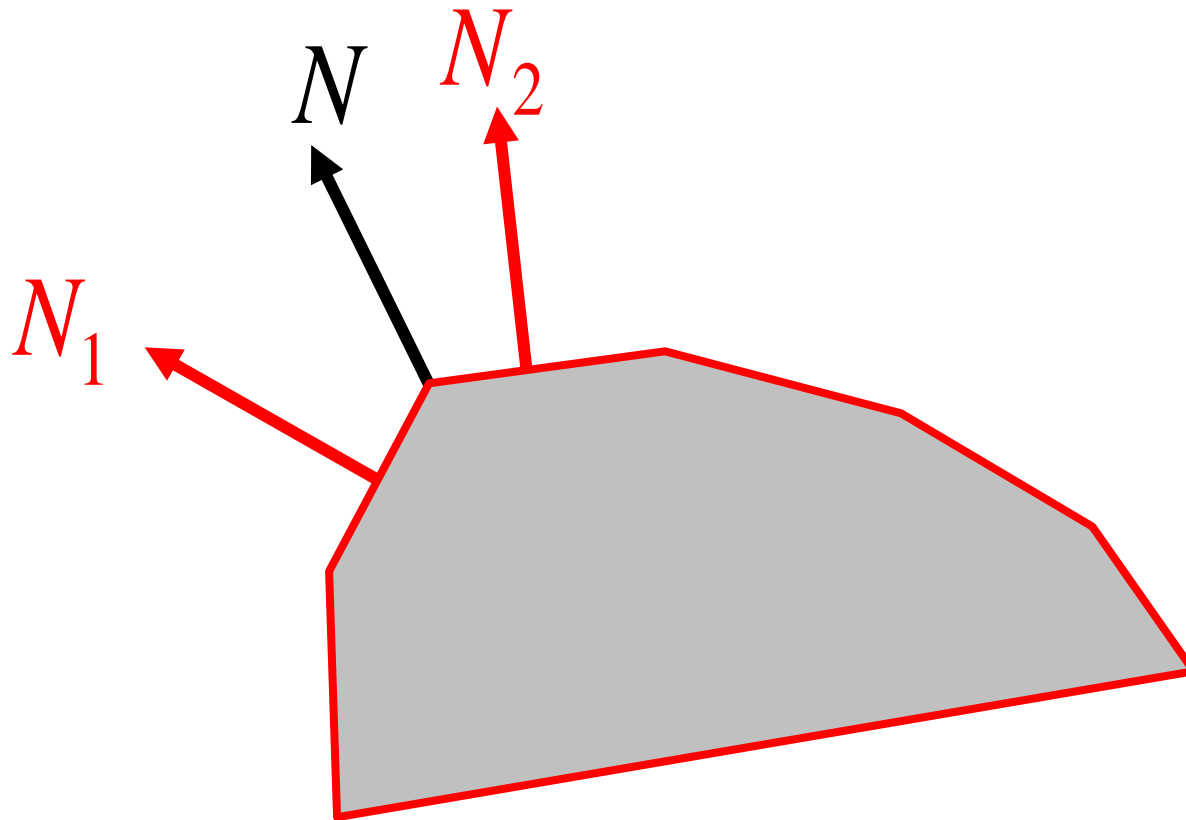


Using Average Normals

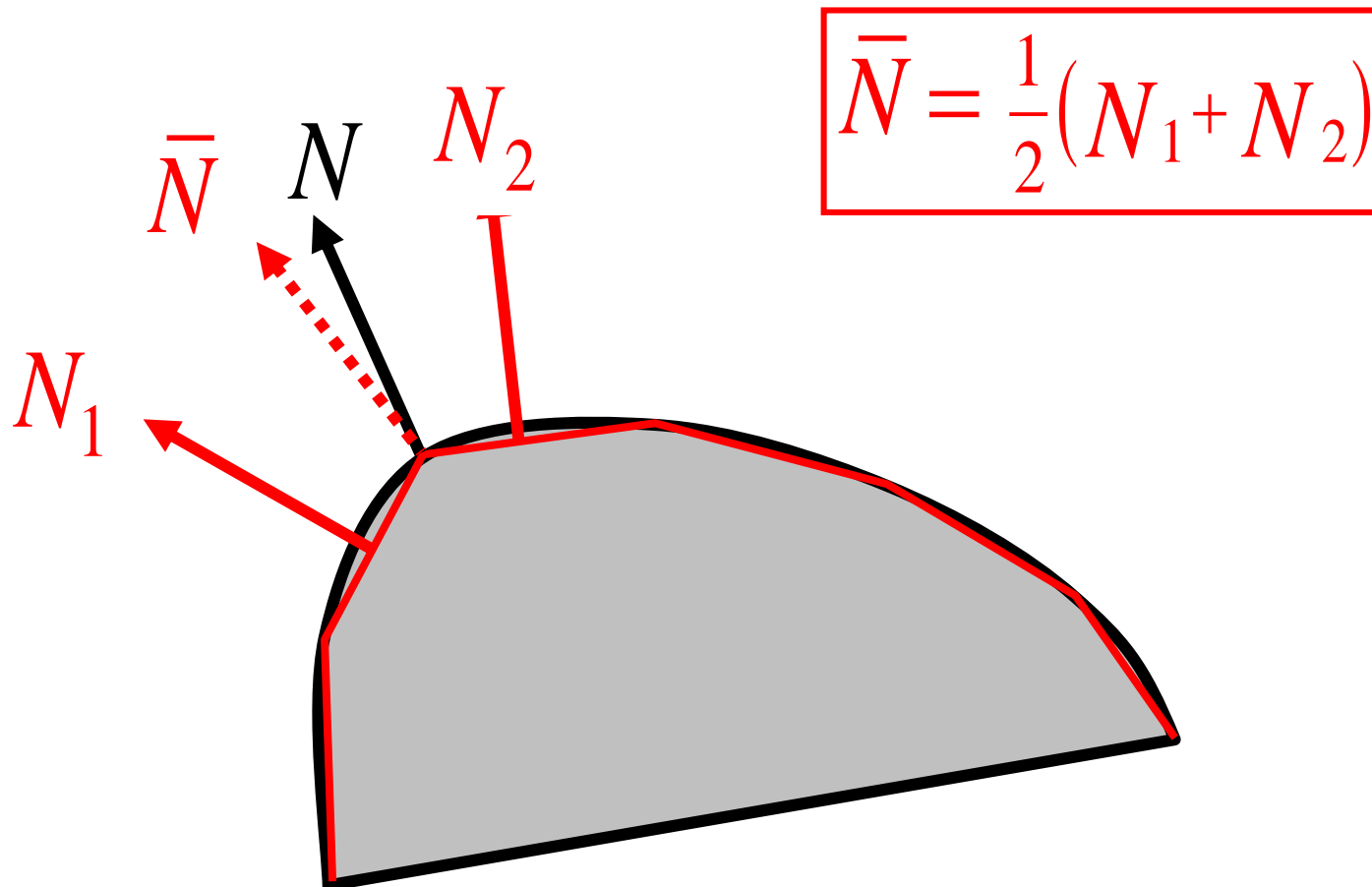
N = true (geometric) normal



Using Average Normals



Using Average Normals

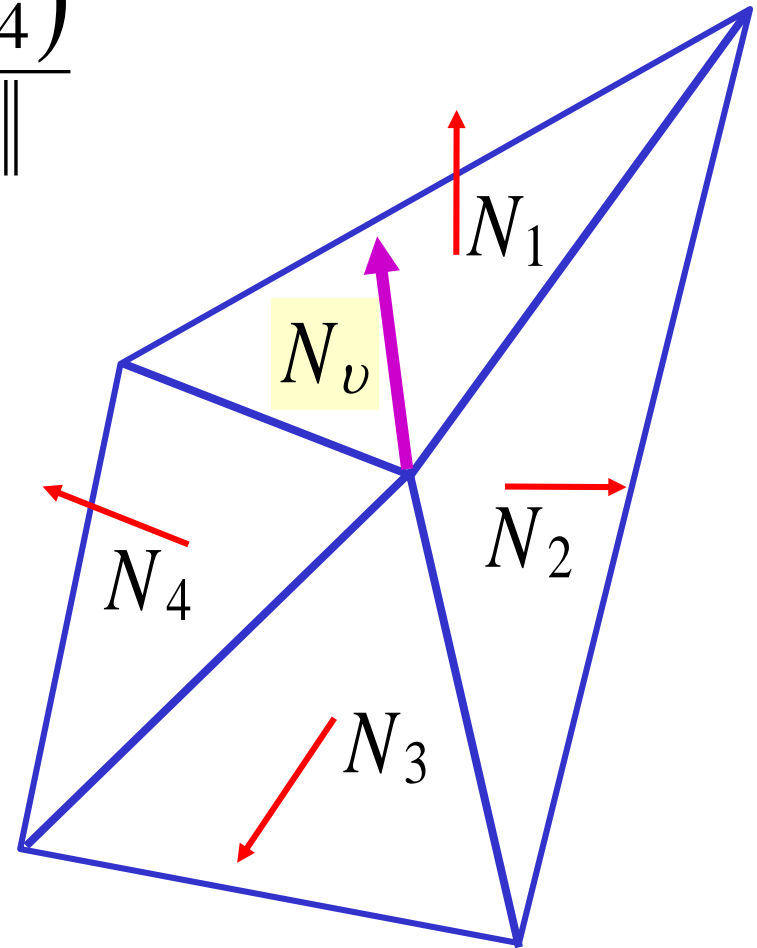


What should corner normals be?

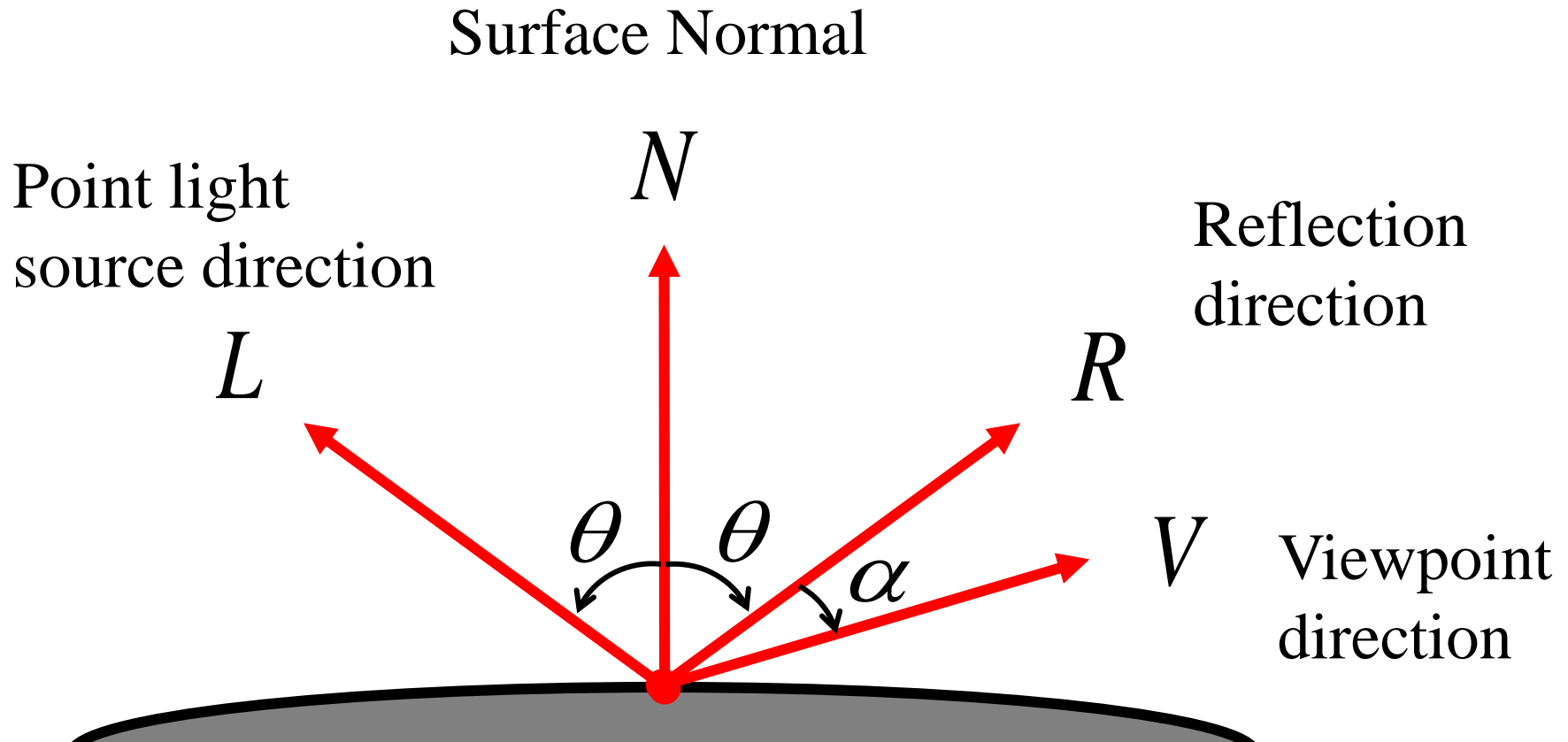
$$N_v = \frac{(N_1 + N_2 + N_3 + N_4)}{\|N_1 + N_2 + N_3 + N_4\|}$$

More generally,

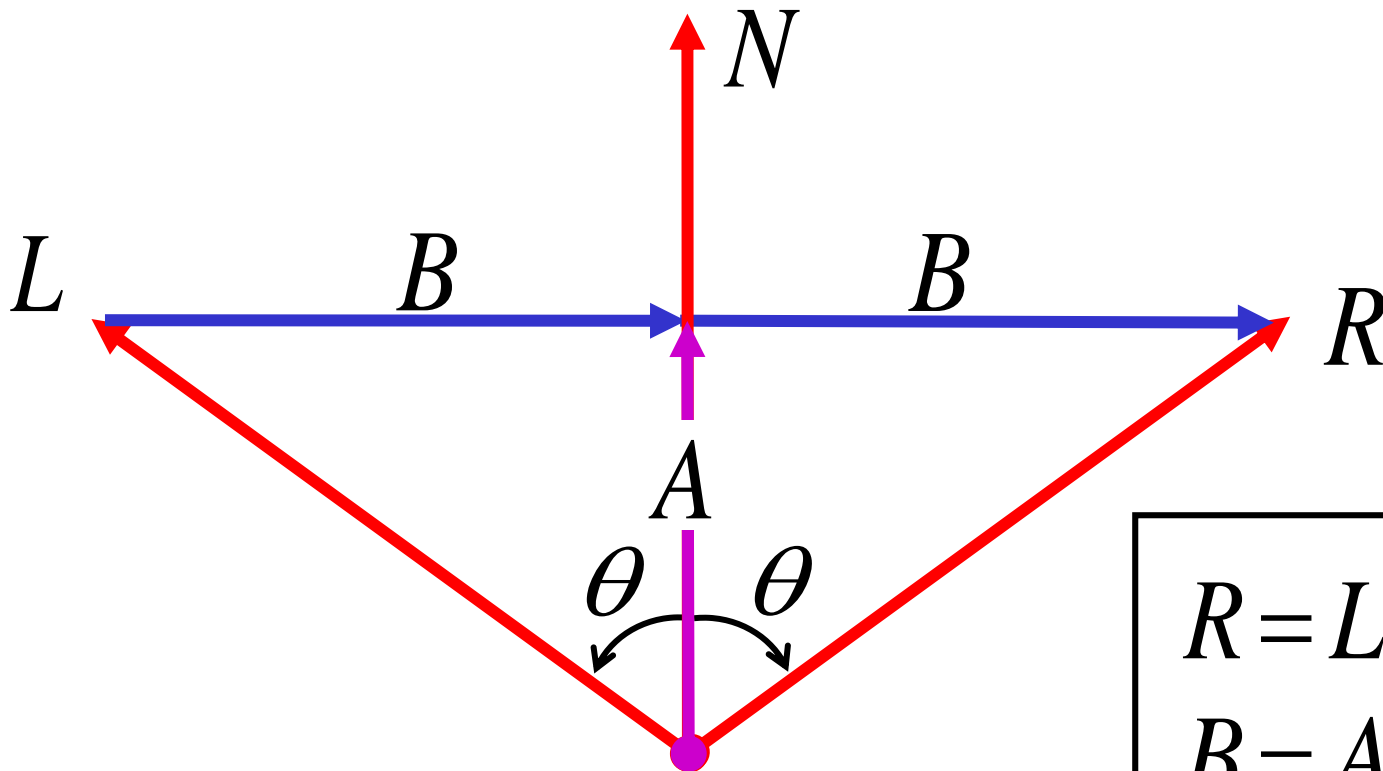
$$N_v = \frac{\sum_{i=1}^n N_i}{\left| \sum_{i=1}^n N_i \right|}$$



Relevant Light (unit) Vectors

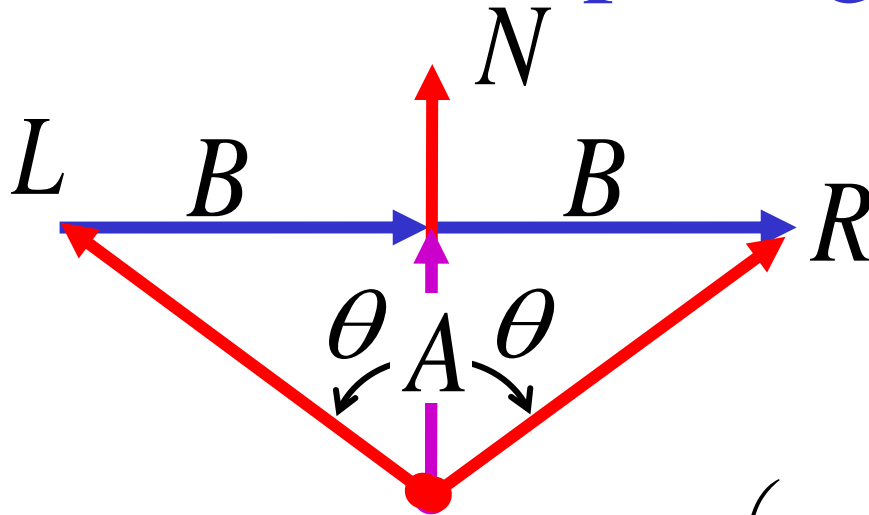


Computing R Vector



$$R = L + 2B$$
$$B = A - L$$

Computing R Vector



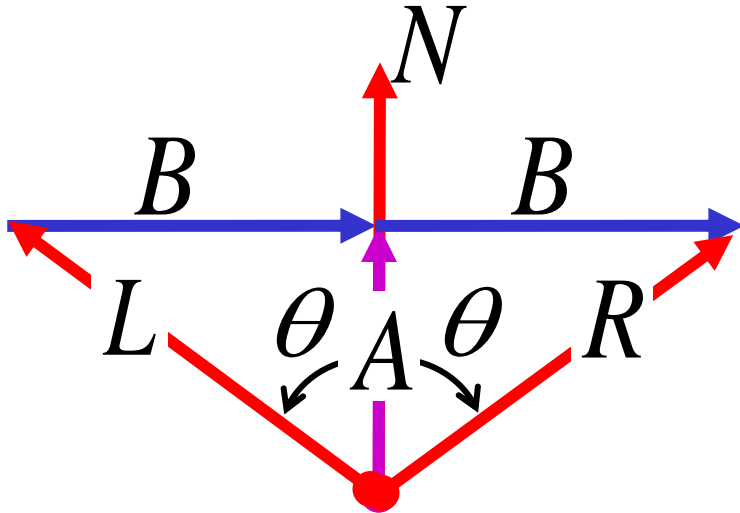
$$A = (L \cdot N)N$$

$$A = N \cos \theta$$

$$R = L + 2 \left(\underbrace{N \cos \theta - L}_B \right)$$

$$R = 2N \cos \theta - L$$

Computing R Vector

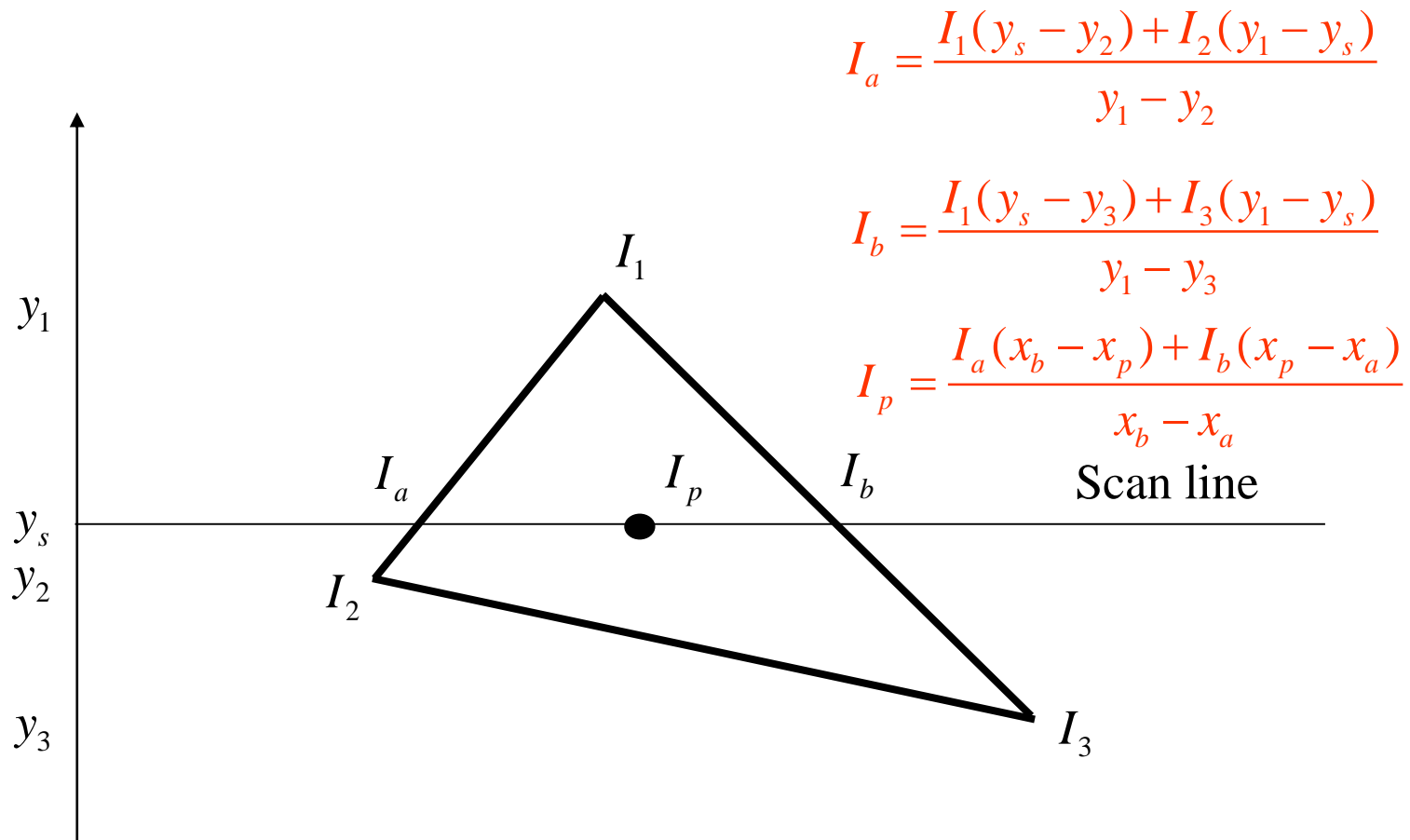


$$R = 2N \cos \theta - L$$

$$R = 2N (N \cdot L) - L$$

$$R \cdot V = (2N (N \cdot L) - L) \cdot V$$

Gouraud Shading – Details



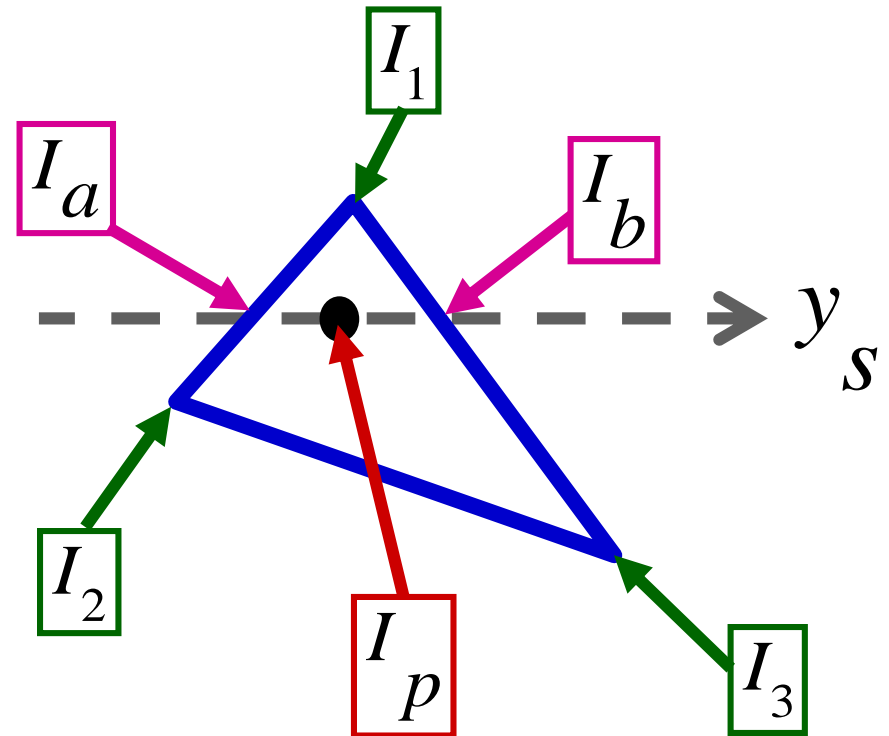
Actual implementation efficient: difference equations while scan converting

Intensity Interpolation (Gouraud)

$$I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2}$$

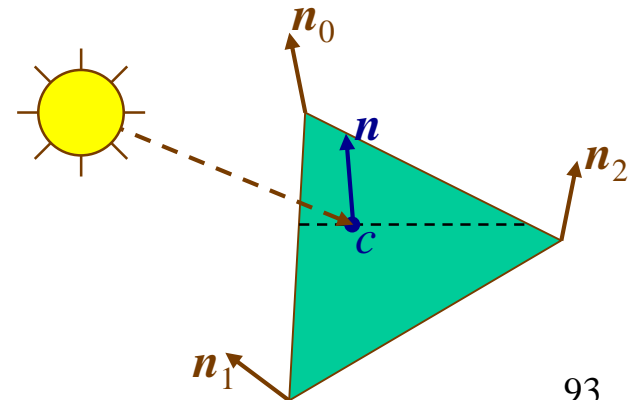
$$I_b = I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3}$$

$$I_p = I_a \frac{x_b - x_p}{x_b - x_a} + I_b \frac{x_p - x_a}{x_b - x_a}$$



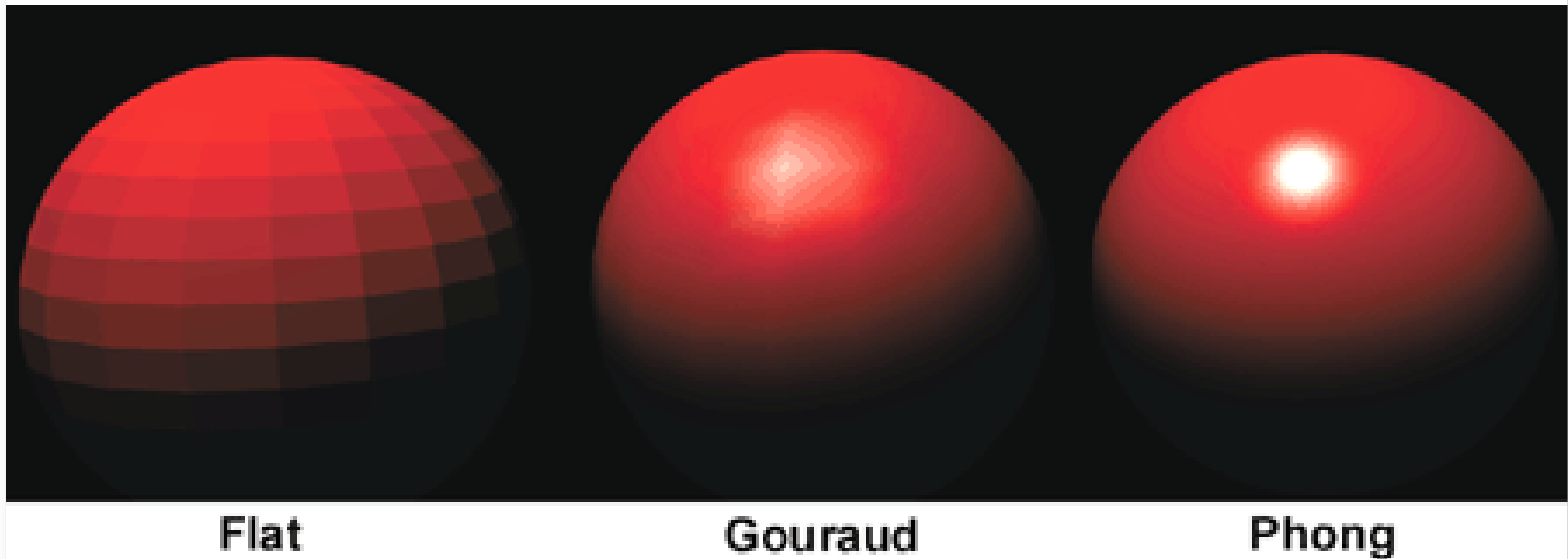
Phong Shading

- Compute normal, \mathbf{n}_i , at each vertex (if not already given)
- Interpolate normals during scan conversion
- Compute color with the interpolated normals
 - **Expensive**: compute illumination for every visible point on a surface
 - Captures **highlights** in the middle of a polygon
 - Looks **smoother** across edges



Shading Comparison

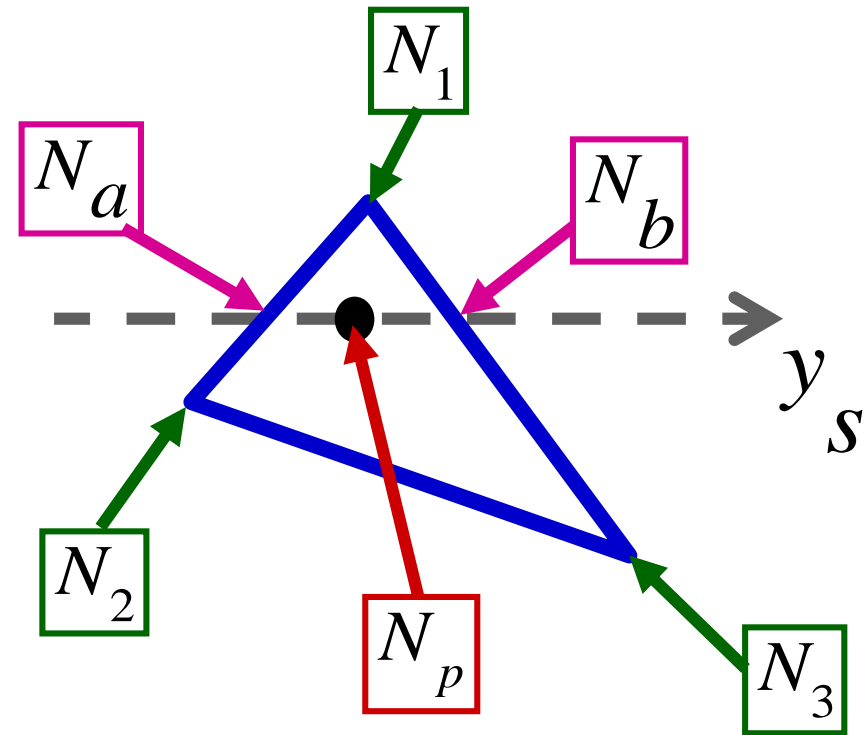
From Computer Desktop Encyclopedia
Reproduced with permission.
© 2001 Intergraph Computer Systems



Normal Interpolation (Phong)

$$N_a = N_1 \frac{y_s - y_2}{y_1 - y_2} + N_2 \frac{y_1 - y_s}{y_1 - y_2}$$

$$N_b = N_1 \frac{y_s - y_3}{y_1 - y_3} + N_3 \frac{y_1 - y_s}{y_1 - y_3}$$



Normal Interpolation (Phong)

$$\tilde{N}_p = \frac{N_a}{\|N_a\|} \begin{bmatrix} x_b - x_p \\ x_b - x_a \end{bmatrix} + \frac{N_b}{\|N_b\|} \begin{bmatrix} x_p - x_a \\ x_b - x_a \end{bmatrix}$$


$$N_p = \frac{\tilde{N}_p}{\|\tilde{N}_p\|}$$

Normalizing makes
this a unit vector

Illumination Formula (1/2)

$$I_{\lambda} = I_{a\lambda} k_a O_{d\lambda}$$

$$+ f_{att} I_{p\lambda} \left[k_d O_{d\lambda} (N \cdot L) + k_s O_{s\lambda} (R \cdot V)^n \right]$$


$$\cos^n(\alpha)$$

Illumination Formula (2/2)

Where,

a denotes *ambient* term

d denotes *diffuse* term

s denotes *specular* term

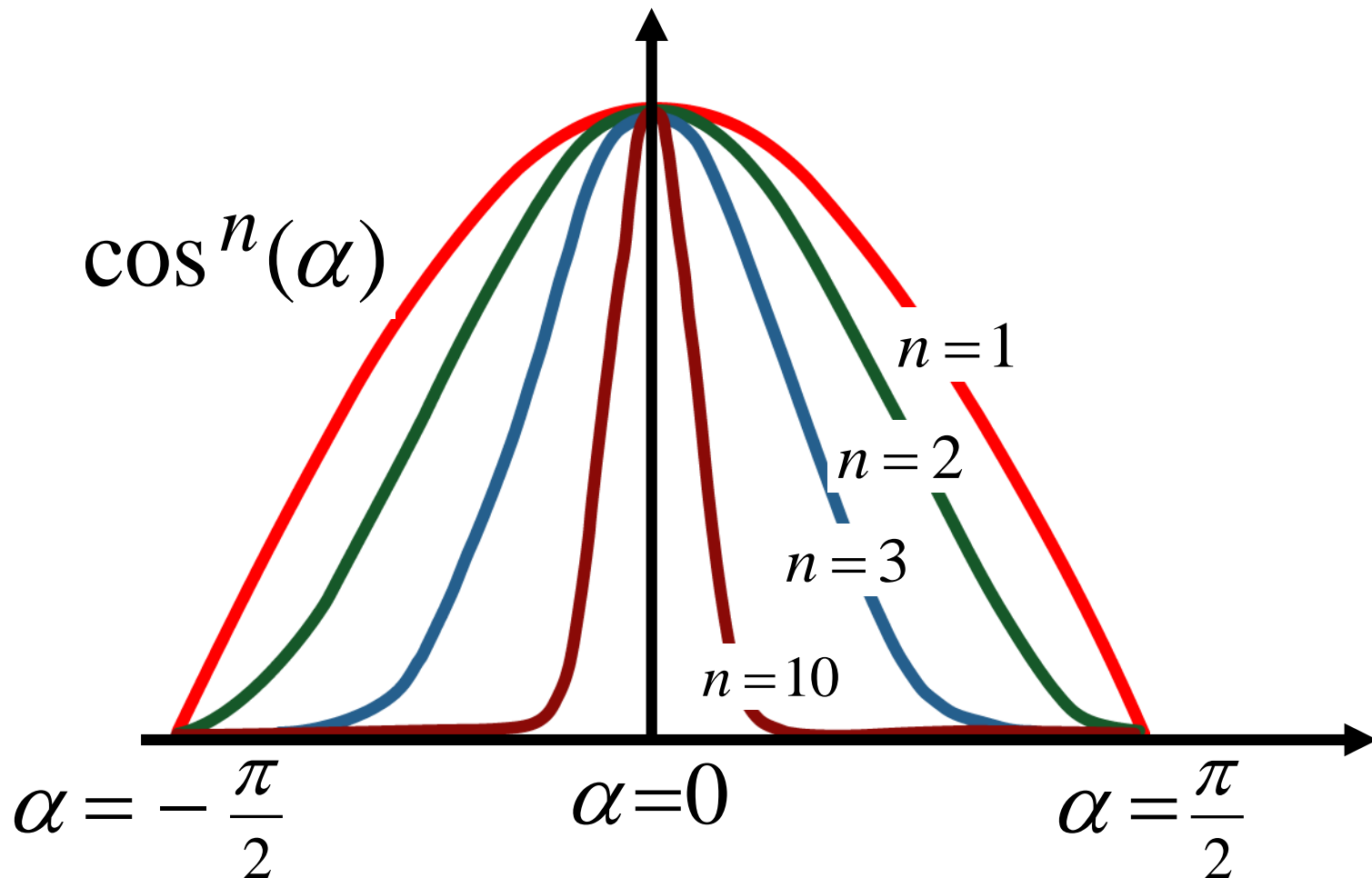
O denotes *object*

k denotes *constant*

I denotes *intensity*

Effect of Exponent Parameter

As n increases, highlight is more concentrated, surface appears glossier

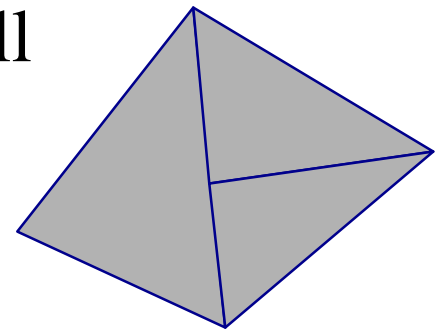


Problems with Interpolated Shading

- Silhouettes are still polygonal
- Interpolation in screen, not object space: perspective distortion
- **Not rotation or orientation-independent**
- How to compute vertex normals for sharply curving surfaces?
- But at end of day, polygons is mostly preferred to explicitly representing curved objects like spline patches for rendering

Problems with Interpolated Shading

- Silhouettes
- Perspective distortion causes problems: Imagine a polygon with 1 vertex at a very different depth than others
 - Interpolation considers equal steps in y , but foreshortening produces unequal steps in depth
 - Problem reduced by using many small polygons
- Mach banding
- **Orientation** dependence for non-triangles
- Shared vertices on an edge



Realism in Shading

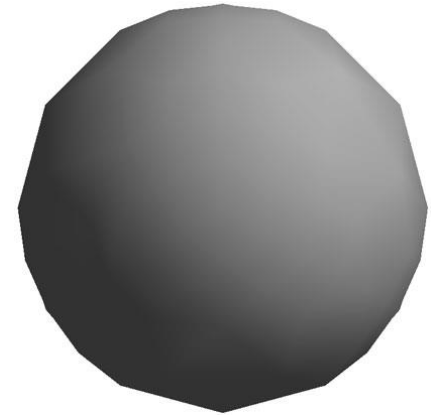
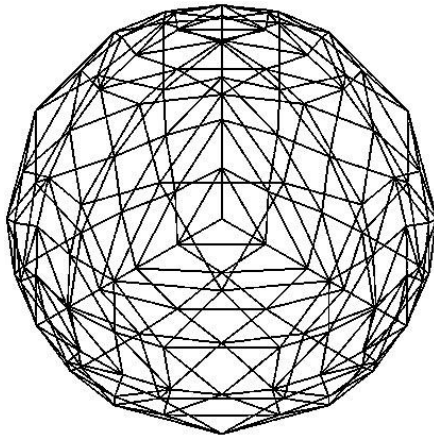
- We've discussed flat, Gouraud, and Phong shading
 - These all attempt to imitate the appearance of objects in the real world
- This may not be desirable for all applications
 - Consider biological drawings

Comparison

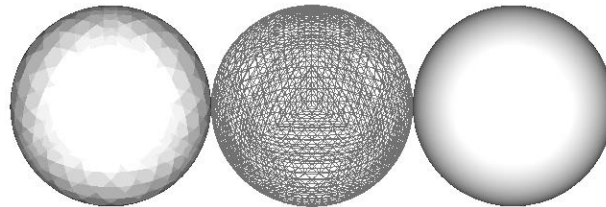
- If the polygon mesh approximates *surfaces with a high curvatures*, Phong shading may look smooth while Gouraud shading may show edges
- Phong shading requires much more work than Gouraud shading
 - Until recently not available in real time systems
 - Now can be done using fragment shaders (see Chapter 9)
- Both need data structures to represent meshes so we can obtain vertex normals

Sample Programs

- **Sphere Program**
 - A.12 sphere.c



A.12 sphere.c



/* Recursive subdivision of cube (Chapter 6). Three
display

A.12 sphere.c (2/12)

modes: *wire frame*, *constant*, and *interpolative shading* */

/*Program also illustrates defining materials and light
sources

in myiit() */

/* mode 0 = wire frame, mode 1 = constant shading,
mode 3 = interpolative shading */

#include <stdlib.h>

#ifdef __APPLE__

#include <GLUT/glut.h>

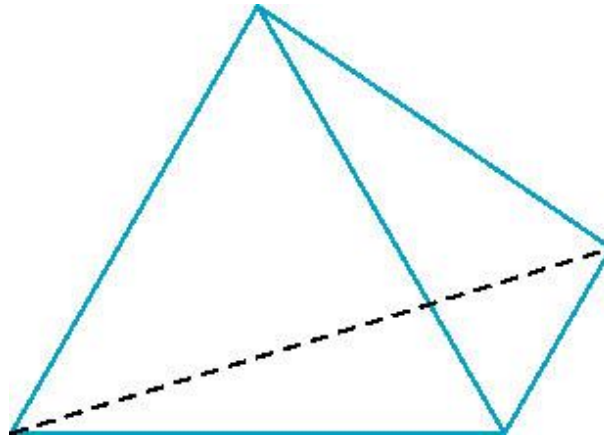
#else

#include <GL/glut.h>

#endif

A.12 sphere.c (3/12)

```
typedef float point[4];  
/* initial tetrahedron */  
point v[] = { {0.0, 0.0, 1.0}, {0.0, 0.942809, -0.333333},  
              {-0.816497, -0.471405, -0.333333}, {0.816497, 0.471405, -0.333333} };  
int n;  
int mode;
```

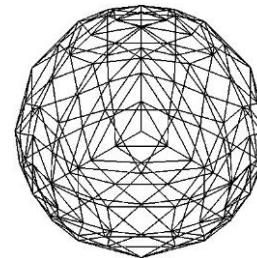


```
void triangle( point a, point b, point c)
```

A.12 sphere.c (4/12)

```
/* display one triangle using a line loop for wire frame, a single  
normal for constant shading, or three normals for interpolative  
shading */
```

```
{  
    if ( mode==0 ) glBegin(GL_LINE_LOOP);  
    else glBegin(GL_POLYGON);  
        if( mode==1 ) glNormal3fv(a);  
        if( mode==2 ) glNormal3fv(a);  
        glVertex3fv(a);  
        if( mode==2 ) glNormal3fv(b);  
        glVertex3fv(b);  
        if( mode==2 ) glNormal3fv(c);  
        glVertex3fv(c);  
    glEnd();  
}
```



Mode=0



Mode=1



Mode=2

```
void normal(point p)
```

```
{
```

```
/* normalize a vector */
```

```
double sqrt();
```

```
float d =0.0;
```

```
int i;
```

```
for(i=0; i<3; i++) d+=p[i]*p[i];
```

```
d=sqrt(d);
```

```
if( d > 0.0) for(i=0; i<3; i++) p[i]/=d;
```

```
}
```

A.12 sphere.c (5/12)

```
void divide_triangle(point a, point b, point c, int m)
```

```
{    /* triangle subdivision using vertex numbers
```

```
    righthand rule applied to create outward pointing faces */
```

```
    point v1, v2, v3;
```

```
    int j;
```

```
    if ( m>0 )
```

```
    { for(j=0; j<3; j++) v1[j]=a[j]+b[j];
```

```
      normal(v1);
```

```
      for(j=0; j<3; j++) v2[j]=a[j]+c[j];
```

```
      normal(v2);
```

```
      for(j=0; j<3; j++) v3[j]=b[j]+c[j];
```

```
      normal(v3);
```

```
      divide_triangle(a, v1, v2, m-1);
```

```
      divide_triangle(c, v2, v3, m-1);
```

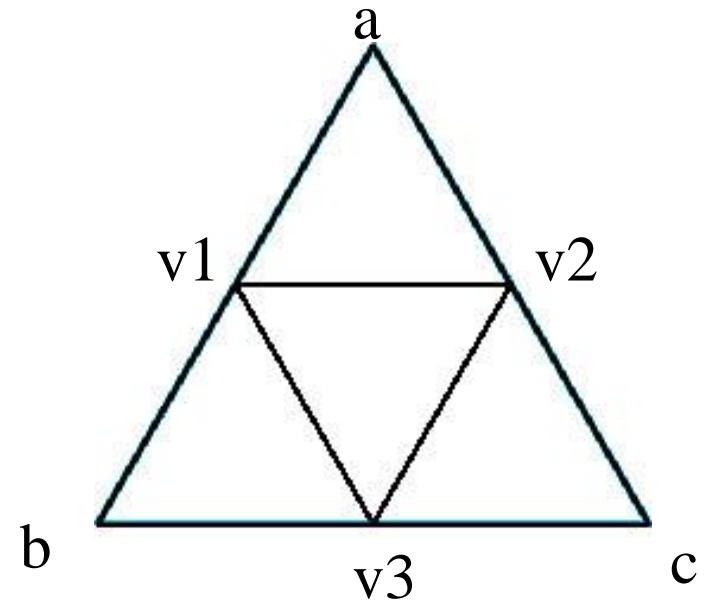
```
      divide_triangle(b, v3, v1, m-1);
```

```
      divide_triangle(v1, v3, v2, m-1); }
```

```
    else (triangle(a,b,c)); /* draw triangle at end of recursion */
```

```
}
```

A.12 sphere.c (6/12)



```
void tetrahedron( int m)
```

```
{
```

```
/* Apply triangle subdivision to faces of tetrahedron */
```

```
    divide_triangle(v[0], v[1], v[2], m);
```

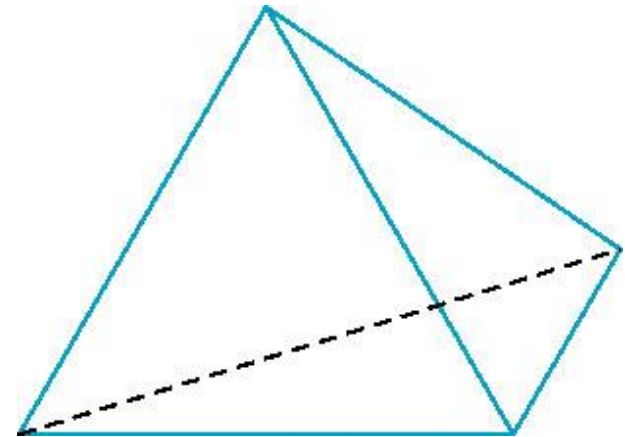
```
    divide_triangle(v[3], v[2], v[1], m);
```

```
    divide_triangle(v[0], v[3], v[1], m);
```

```
    divide_triangle(v[0], v[2], v[3], m);
```

```
}
```

A.12 sphere.c (7/12)



```
void display(void)
```

```
{
```

```
    /* Displays all three modes, side by side */
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glLoadIdentity();
```

```
    mode=0;
```

```
    tetrahedron(n);
```

```
    mode=1;
```

```
    glTranslatef(-2.0, 0.0,0.0);
```

```
    tetrahedron(n);
```

```
    mode=2;
```

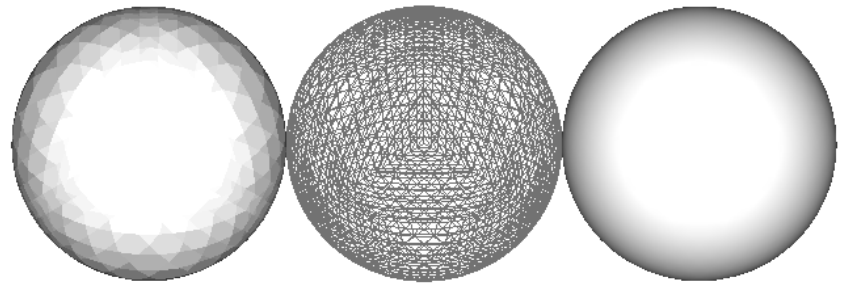
```
    glTranslatef( 4.0, 0.0,0.0);
```

```
    tetrahedron(n);
```

```
    glFlush();
```

```
}
```

A.12 sphere.c (8/12)



Mode=1

Mode=0

Mode=2


```
void myReshape(int w, int h)
```

A.12 sphere.c (9/12)

```
{  
    glViewport(0, 0, w, h);  
    glMatrixMode(GL_PROJECTION);  
    glLoadIdentity();  
    if (w <= h)  
        glOrtho(-4.0, 4.0, -4.0 * (GLfloat) h / (GLfloat) w,  
                4.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);  
    else  
        glOrtho(-4.0 * (GLfloat) w / (GLfloat) h,  
                4.0 * (GLfloat) w / (GLfloat) h, -10.0, 10.0);  
    glMatrixMode(GL_MODELVIEW);  
    display();  
}
```

void myinit()

A.12 sphere.c (10/12)

```
{
    GLfloat mat_specular[]={ 1.0, 1.0, 1.0, 1.0};
    GLfloat mat_diffuse[]={ 1.0, 1.0, 1.0, 1.0};
    GLfloat mat_ambient[]={ 1.0, 1.0, 1.0, 1.0};
    GLfloat mat_shininess={ 100.0};
    GLfloat light_ambient[]={ 0.0, 0.0, 0.0, 1.0};
    GLfloat light_diffuse[]={ 1.0, 1.0, 1.0, 1.0};
    GLfloat light_specular[]={ 1.0, 1.0, 1.0, 1.0};

    /* set up ambient, diffuse, and specular components for light 0
    */
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

A.12 sphere.c (11/12)

/* define material proerties for front face of all polygons */

```
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);  
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);  
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);  
glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);
```

```
glShadeModel(GL_SMOOTH); /*enable smooth shading */  
glEnable(GL_LIGHTING); /* enable lighting */  
glEnable(GL_LIGHT0); /* enable light 0 */  
glEnable(GL_DEPTH_TEST); /* enable z buffer */
```

```
glClearColor (1.0, 1.0, 1.0, 1.0);  
glColor3f (0.0, 0.0, 0.0);
```

```
}
```

```
void  
main(int argc, char **argv)
```

A.12 sphere.c (12/12)

```
{  
    n=5;  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB |  
                        GLUT_DEPTH);  
    glutInitWindowSize(500, 500);  
    glutCreateWindow("sphere");  
    myinit();  
    glutReshapeFunc(myReshape);  
    glutDisplayFunc(display);  
    glutMainLoop();  
}
```