CSCI 420 Computer Graphics
Lecture 11

Lighting and Shading

Light Sources
Phong Illumination Model
Normal Vectors
[Angel Ch. 5]

Jernej Barbic
University of Southern California

Outline

- Global and Local Illumination
- Normal Vectors
- Light Sources
- Phong Illumination Model
- Polygonal Shading
- Example

Global Illumination

- Ray tracing
- Radiosity
- Photon Mapping

Tobias R. Metoc

- Follow light rays through a scene
- Accurate, but expensive (off-line)

Raytracing Example

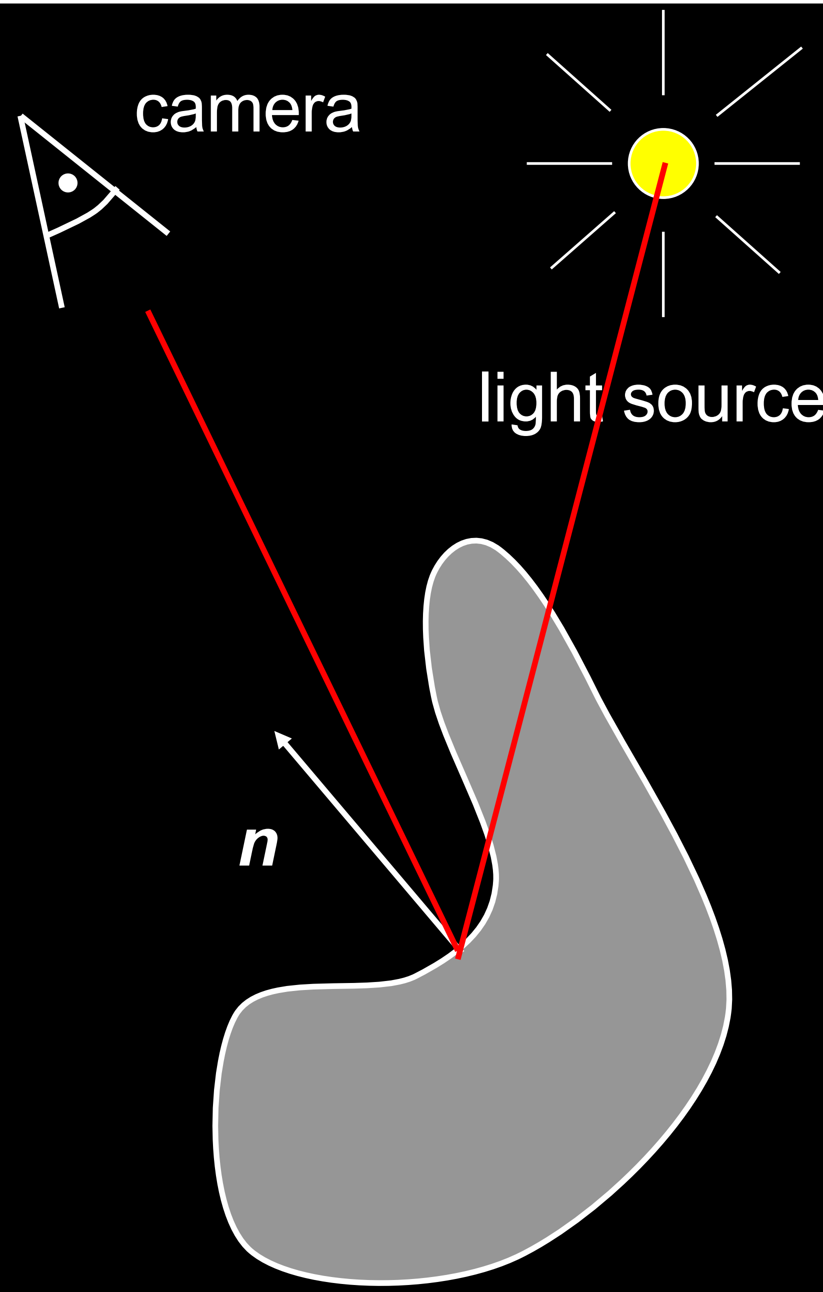
Martin Moeck,
Siemens Lighting

Radiosity Example

Restaurant Interior. Guillermo Leal, Evolucion Visual

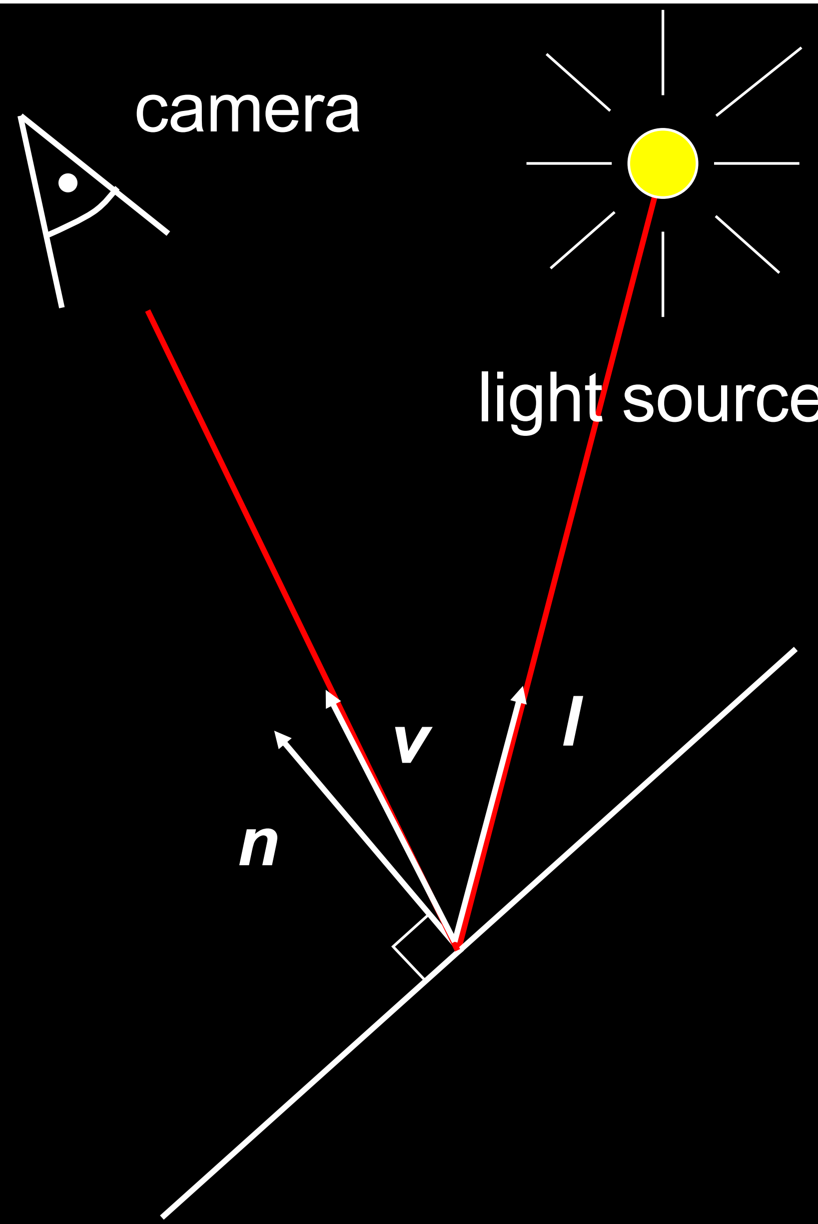
Local Illumination

- Approximate model
- Local interaction between light, surface, viewer
- Phong model (this lecture): fast, supported in OpenGL
- GPU shaders
- Pixar Renderman (offline)



Local Illumination

- Approximate model
- Local interaction between light, surface, viewer
- Color determined only based on surface normal, relative camera position and relative light position
- What effects does this ignore?



Outline

- Global and Local Illumination
- Normal Vectors
- Light Sources
- Phong Illumination Model
- Polygonal Shading
- Example

Normal Vectors

- Must calculate and specify the normal vector
  - Even in OpenGL!
- Two examples: plane and sphere

Normals of a Plane, Method I

- Method I: given by $ax + by + cz + d = 0$
- Let p_0 be a known point on the plane
- Let p be an arbitrary point on the plane
- Recall: $u \cdot v = 0$ if and only if u orthogonal to v
- $n \cdot (p - p_0) = n \cdot p - n \cdot p_0 = 0$

- Consequently $n_0 = [a \ \ b \ \ c]^T$
- Normalize to $n = n_0/|n_0|$

Normals of a Plane, Method II

- Method II: plane given by p_0, p_1, p_2
- Points must not be collinear
- Recall: u x v orthogonal to u and v

- $n_0 = (p_1 - p_0) \times (p_2 - p_0)$

- Order of cross product determines orientation
- Normalize to $n = n_0/|n_0|$

Normals of Sphere

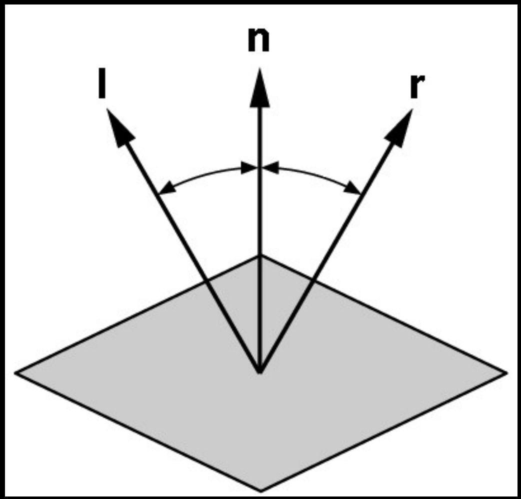
- Implicit Equation $f(x, y, z) = x^2 + y^2 + z^2 - 1 = 0$
- Vector form: $f(p) = p \cdot p - 1 = 0$
- Normal given by gradient vector

$$n_0 = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \\ 2z \end{bmatrix} = 2p$$

- Normalize $n_0/|n_0| = 2p/2 = p$

Reflected Vector

- Perfect reflection: angle of incident equals angle of reflection
- Also: $\boldsymbol{l}$, $\boldsymbol{n}$, and $\boldsymbol{r}$ lie in the same plane
- Assume $|\boldsymbol{l}| = |\boldsymbol{n}| = 1$, guarantee $|\boldsymbol{r}| = 1$



$$\boldsymbol{l} \cdot \boldsymbol{n} = \cos(\theta) = \boldsymbol{n} \cdot \boldsymbol{r}$$

$$\boldsymbol{r} = \alpha\, \boldsymbol{l} + \beta\, \boldsymbol{n}$$

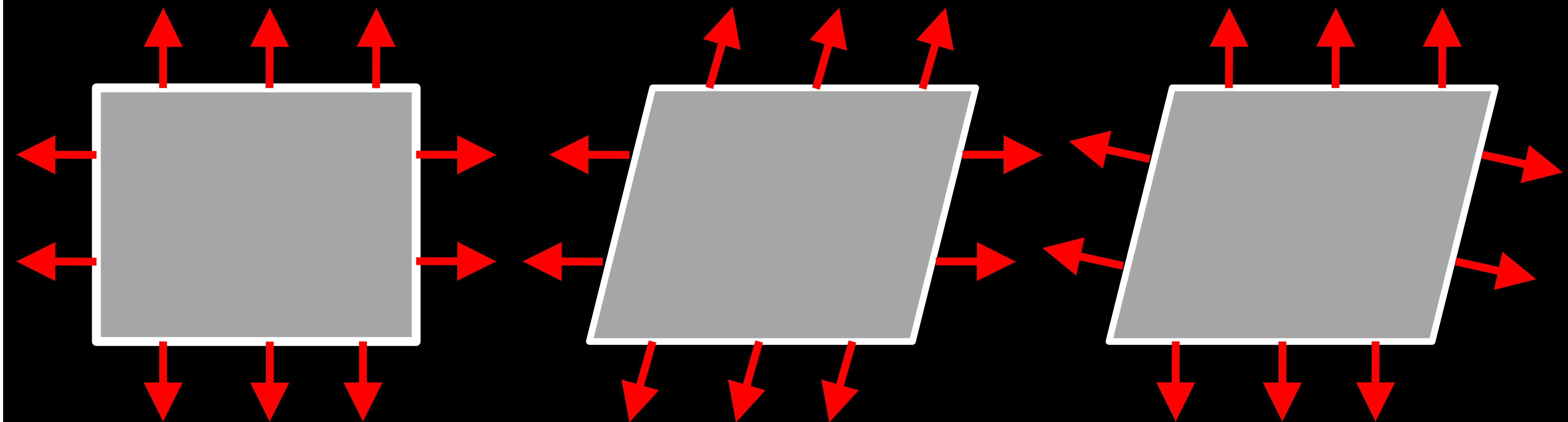
Solution: $\alpha = -1$ and $\beta = 2\,(\boldsymbol{l} \cdot \boldsymbol{n})$

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

Normals Transformed by Modelview Matrix

Modelview matrix M (shear in this example)

Only keep linear transform in M (discard any translation).



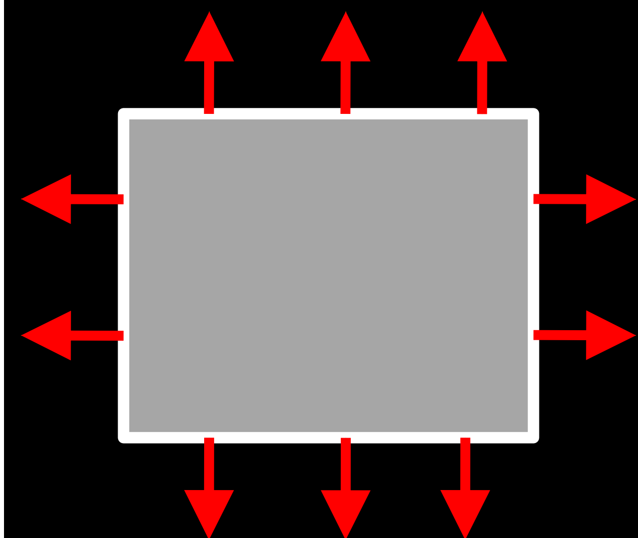
Undeformed

Transformed with M (incorrect)

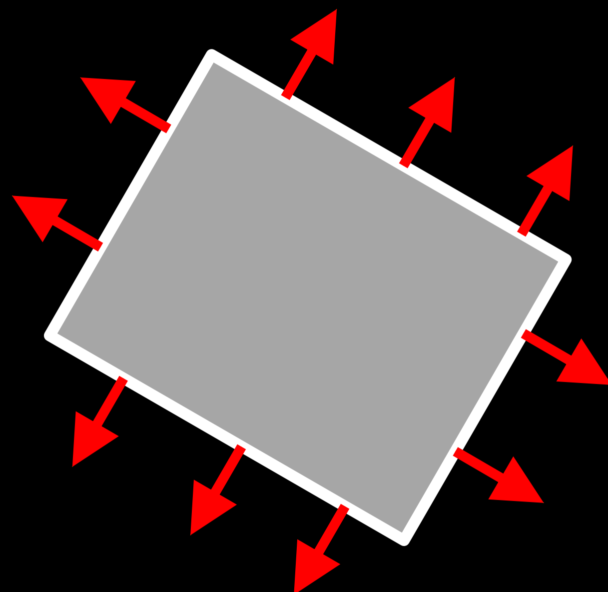
Transformed with $(M^{-1})^T$ (correct)

Normals Transformed by Modelview Matrix

When M is rotation, $M = (M^{-1})^T$



Undeformed



Transformed
with $M = (M^{-1})^T$

(correct)

Normals Transformed by Modelview Matrix (proof of $(M^{-1})^T$ *transform)*

Point (x,y,z,w) is on a plane in 3D (homogeneous coordinates) if and only if

$a x + b y + c z + d w = 0$, or $[a\ b\ c\ d]\ [x\ y\ z\ w]^T = 0$.

Now, let's transform the plane by *M*.

Point (x,y,z,w) is on the transformed plane if and only if

$M^{-1}\ [x\ y\ z\ w]^T$ is on the original plane:

$[a\ b\ c\ d]\ M^{-1}\ [x\ y\ z\ w]^T = 0$.

So, equation of transformed plane is

$[a'\ b'\ c'\ d']\ [x\ y\ z\ w]^T = 0$, for

$[a'\ b'\ c'\ d']^T = (M^{-1})^T\ [a\ b\ c\ d]^T$.

Outline

- Global and Local Illumination
- Normal Vectors
- Light Sources
- Phong Illumination Model
- Polygonal Shading
- Example

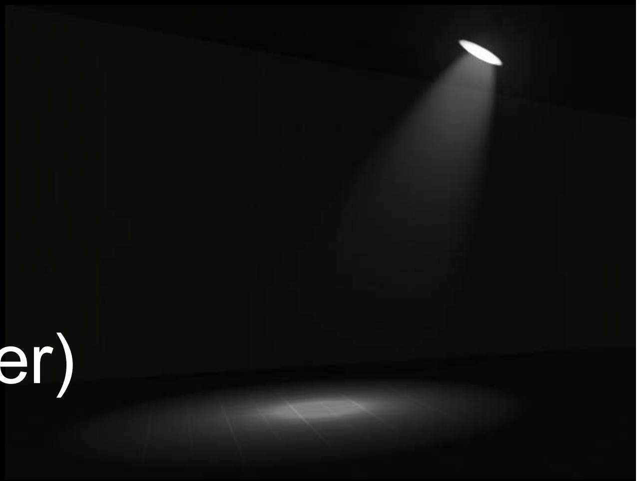
Light Sources and Material Properties

- Appearance depends on
  - Light sources, their locations and properties
  - Material (surface) properties:

  - Viewer position

Types of Light Sources

- Ambient light: no identifiable source or direction
- Point source: given only by point
- Distant light: given only by direction
- Spotlight: from source in direction
  - Cut-off angle defines a cone of light
  - Attenuation function (brighter in center)



Point Source

- Given by a point p_0
- Light emitted equally in all directions
- Intensity decreases with square of distance

$$I \propto \frac{1}{|p - p_0|^2}$$

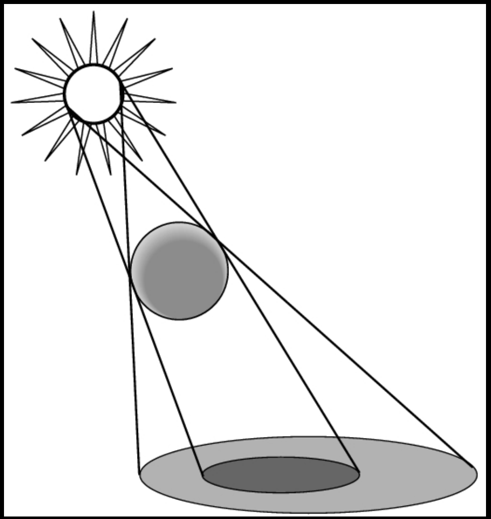
Limitations of Point Sources

- Shading and shadows inaccurate
- Example: penumbra (partial "soft" shadow)
- Similar problems with highlights
- Compensate with attenuation

$$\frac{1}{a + bq + cq^2}$$

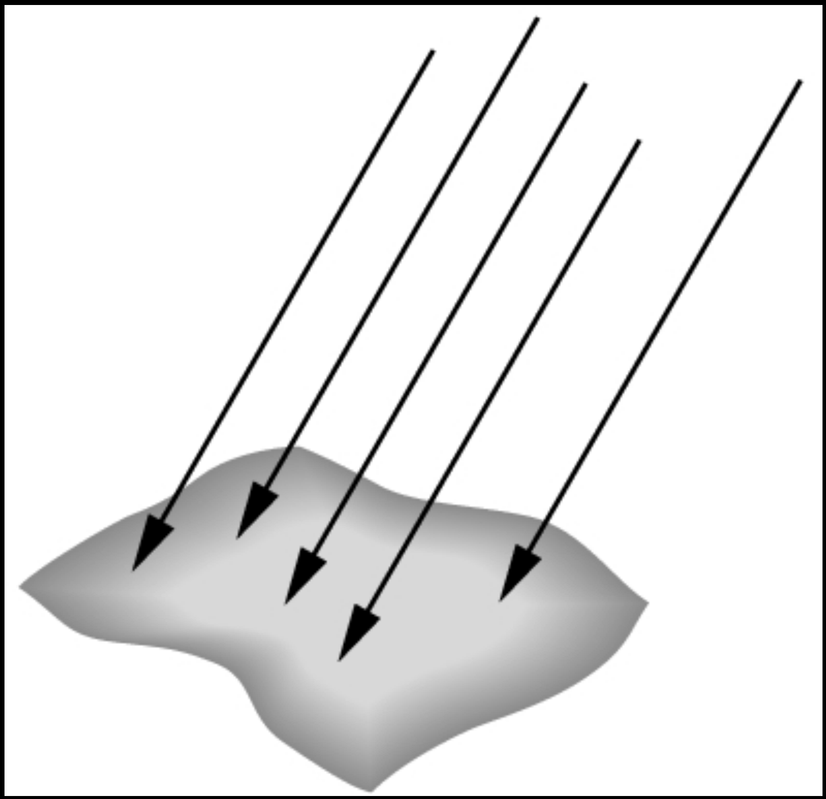
q = distance $|p - p_0|$
a, b, c constants

- Softens lighting
- Better with ray tracing
- Better with radiosity



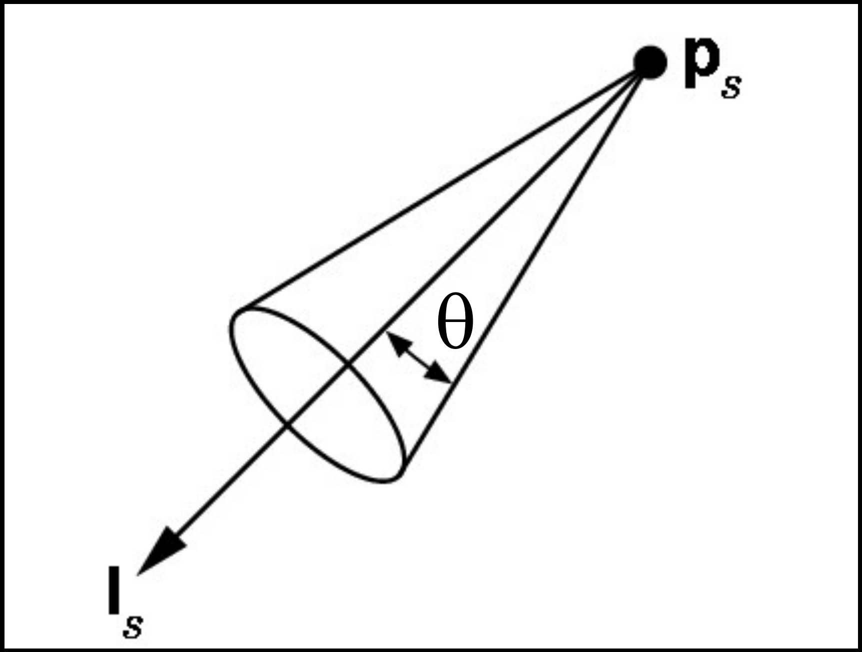
Distant Light Source

- Given by a direction vector [x y z]



Spotlight

- Light still emanates from point
- Cut-off by cone determined by angle θ



Global Ambient Light

- Independent of light source
- Lights entire scene
- Computationally inexpensive
- Simply add [G_R G_G G_B] to every pixel on every object
- Not very interesting on its own.
  A cheap hack to make the scene brighter.

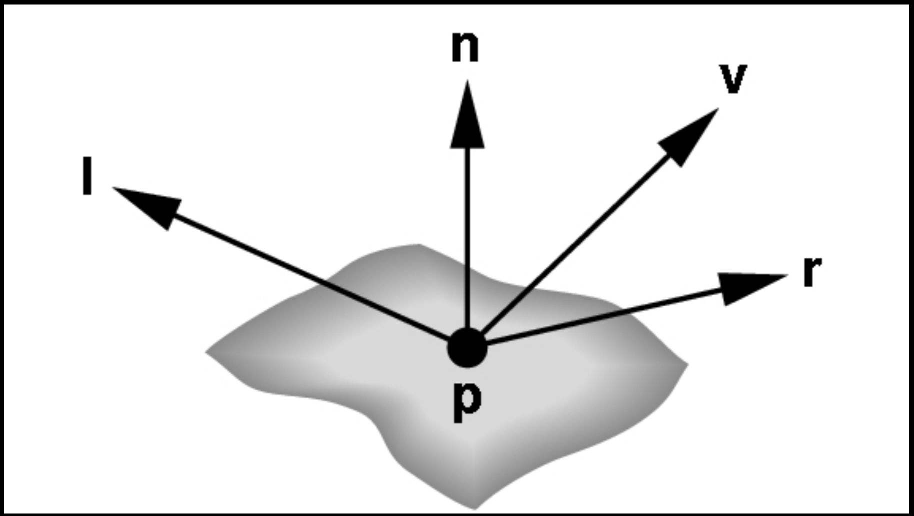
Outline

- Global and Local Illumination
- Normal Vectors
- Light Sources
- Phong Illumination Model
- Polygonal Shading
- Example

Phong Illumination Model

- Calculate color for arbitrary point on surface
- Compromise between realism and efficiency
- Local computation (no visibility calculations)
- Basic inputs are material properties and **l**, **n**, **v**:

l = unit vector to light source
n = surface normal
v = unit vector to viewer
r = reflection of **l** at **p**
(determined by **l** and **n**)



Phong Illumination Overview

1. Start with global ambient light [G_R G_G G_B]
2. Add contributions from each light source
3. Clamp the final result to [0, 1]

- Calculate each color channel (R,G,B) **separately**
- Light source contributions decomposed into
  - Ambient reflection
  - Diffuse reflection
  - Specular reflection
- Based on ambient, diffuse, and specular lighting and material properties

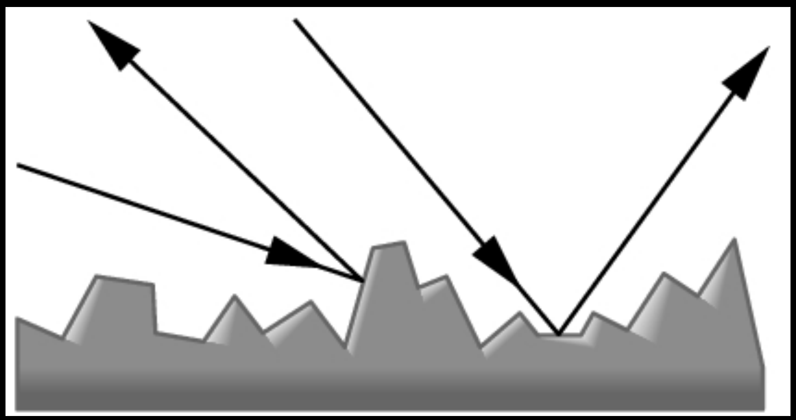
Ambient Reflection

$$I_a = k_a L_a$$

- Intensity of ambient light is uniform at every point
- Ambient reflection coefficient $k_a \geq 0$
- May be different for every surface and r,g,b
- Determines reflected fraction of ambient light
- L_a = ambient component of light source (can be set to different value for each light source)
- Note: L_a is not a physically meaningful quantity

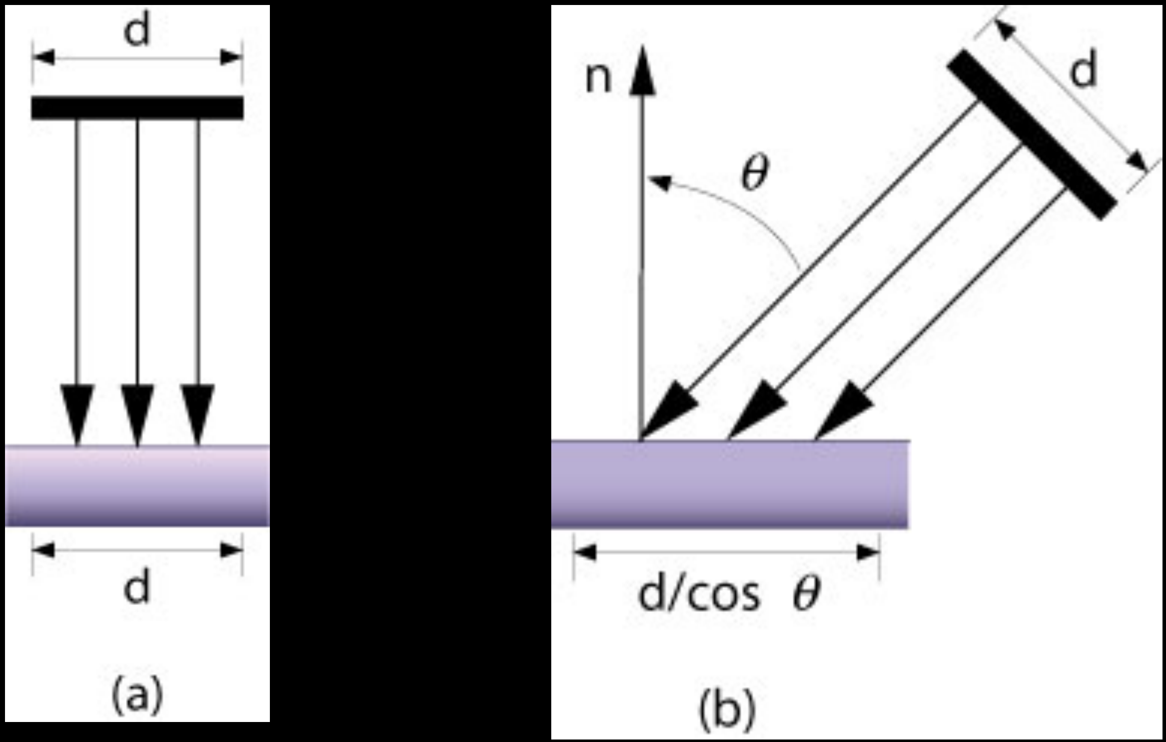
Diffuse Reflection

- Diffuse reflector scatters light
- Assume equally all direction
- Called Lambertian surface
- Diffuse reflection coefficient $k_d \geq 0$
- Angle of incoming light is important



Lambert's Law

Intensity depends on angle of incoming light.



Diffuse Light Intensity Depends On Angle Of Incoming Light

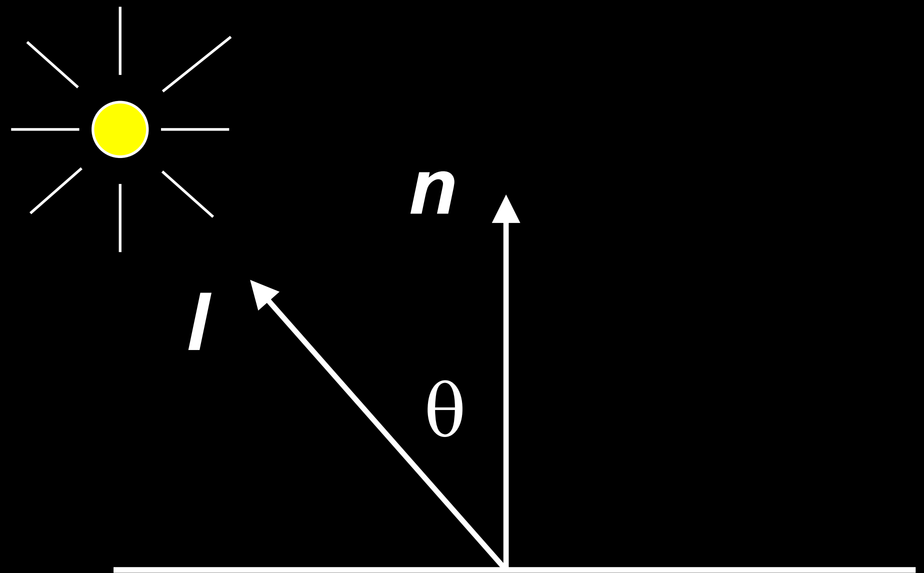
- Recall

  l = unit vector to light

  n = unit surface normal

  θ = angle to normal

- $\cos\theta = \boldsymbol{l} \cdot \boldsymbol{n}$



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

- With attenuation:

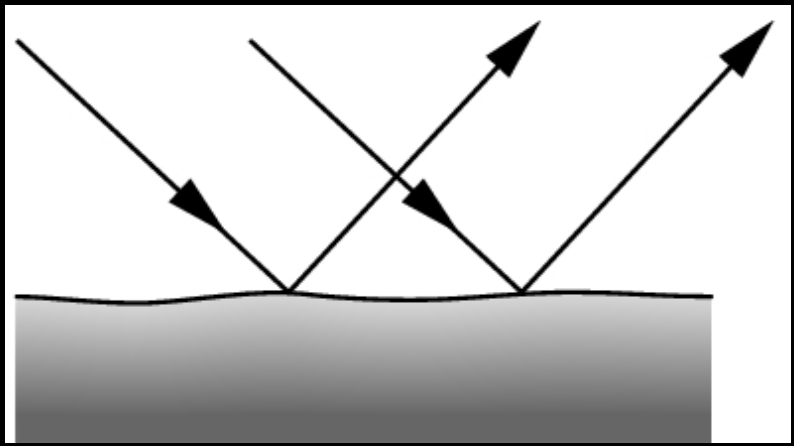
$$I_d = \frac{k_d L_d}{a + bq + cq^2} (l \cdot n)$$

q = distance to light source,

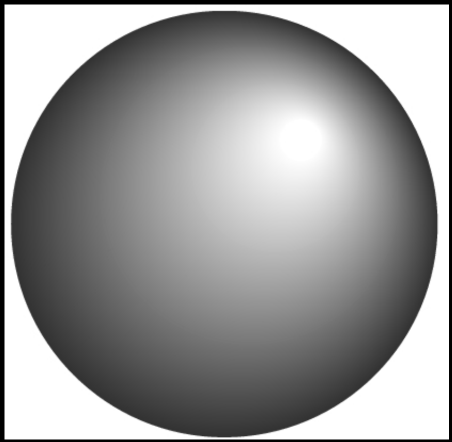
L_d = diffuse component of light

Specular Reflection

- Specular reflection coefficient $k_s \geq 0$
- Shiny surfaces have high specular coefficient
- Used to model specular highlights
- Does not give the mirror effect (need other techniques)



specular reflection



specular highlights

Specular Reflection

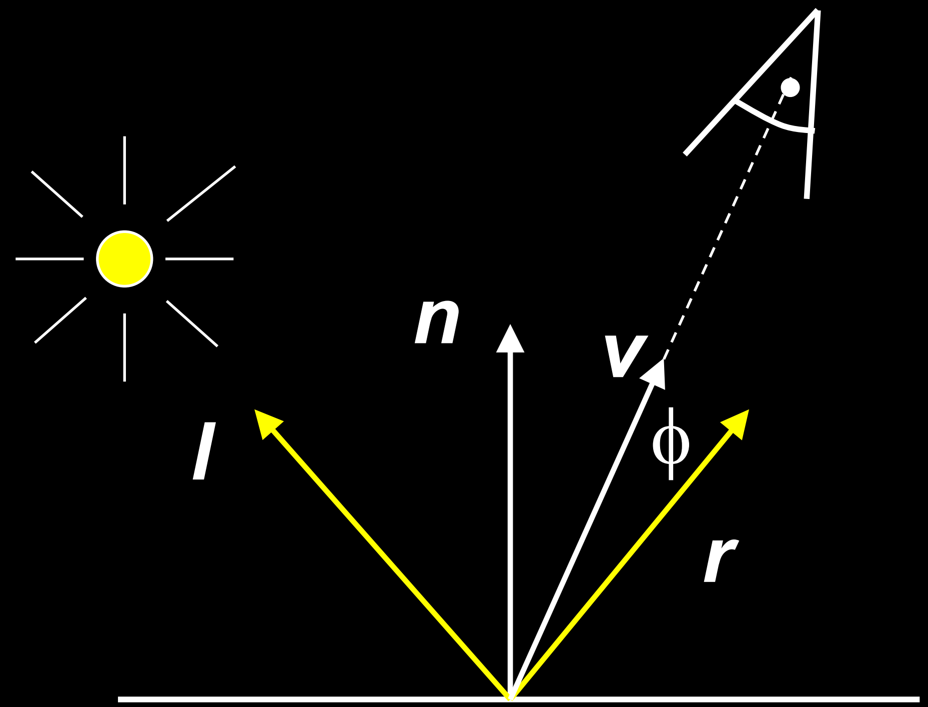
- Recall

  v = unit vector to camera

  r = unit reflected vector

  ϕ = angle between ***v*** and ***r***

- $\cos \phi = \mathbf{v} \cdot \mathbf{r}$

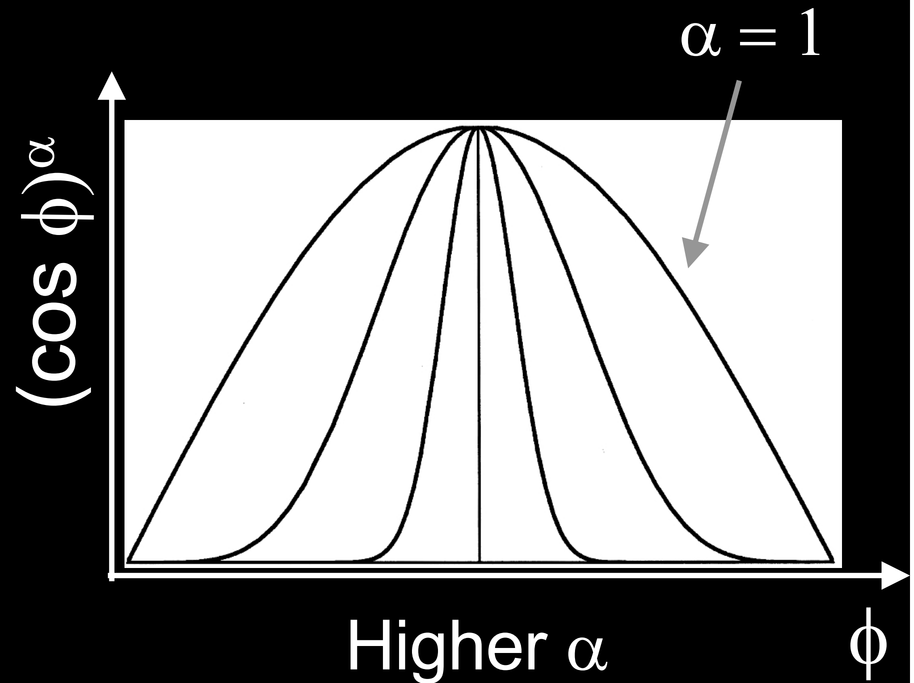


- $I_s = k_s L_s (\cos \phi)^\alpha$

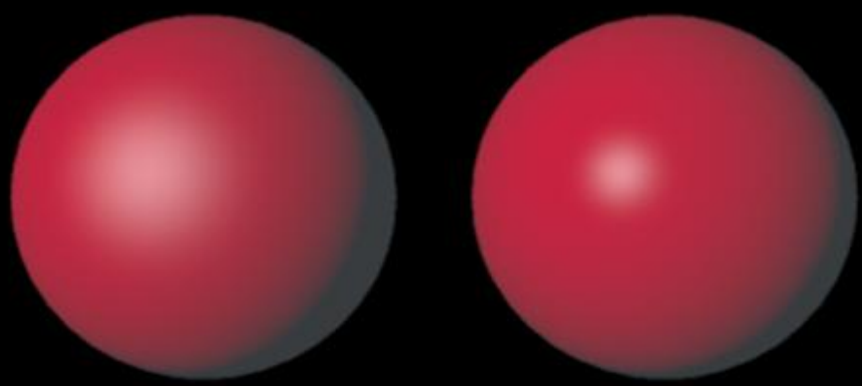
- L_s is specular component of light
- α is shininess coefficient
- Can add distance term as well

Shininess Coefficient

- $I_s = k_s L_s (\cos \phi)^\alpha$
- α is the shininess coefficient



Higher α gives narrower curves



low α high α

Source:
Univ. of Calgary

Summary of Phong Model

- Light components for each color:
  - Ambient (L_a), diffuse (L_d), specular (L_s)
- Material coefficients for each color:
  - Ambient (k_a), diffuse (k_d), specular (k_s)
- Distance q for surface point from light source

$$I = \frac{1}{a + bq + cq^2}\left(k_d L_d (l \cdot n) + k_s L_s (r \cdot v)^{\alpha}\right) + k_a L_a$$

l = unit vector to light

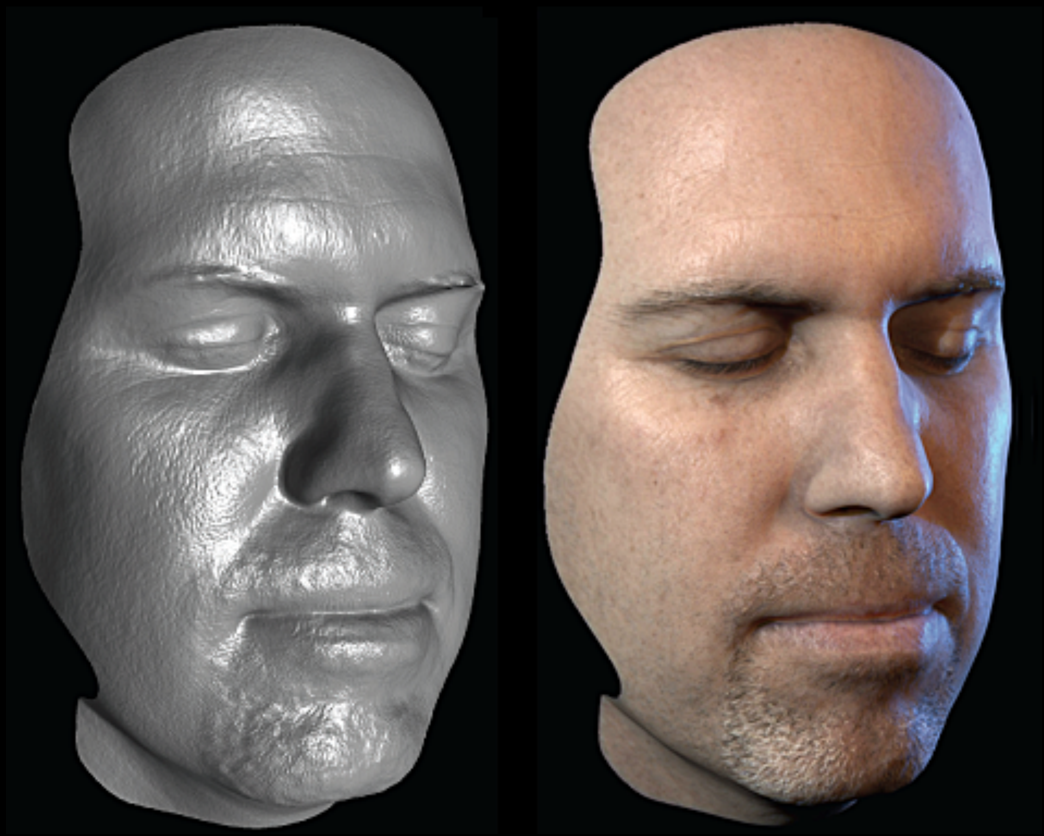
n = surface normal

r = ***l*** reflected about ***n***

v = vector to viewer

BRDF

- Bidirectional Reflection Distribution Function
- Must measure for real materials
- Isotropic vs. anisotropic
- Mathematically complex
- Implement in a fragment shader



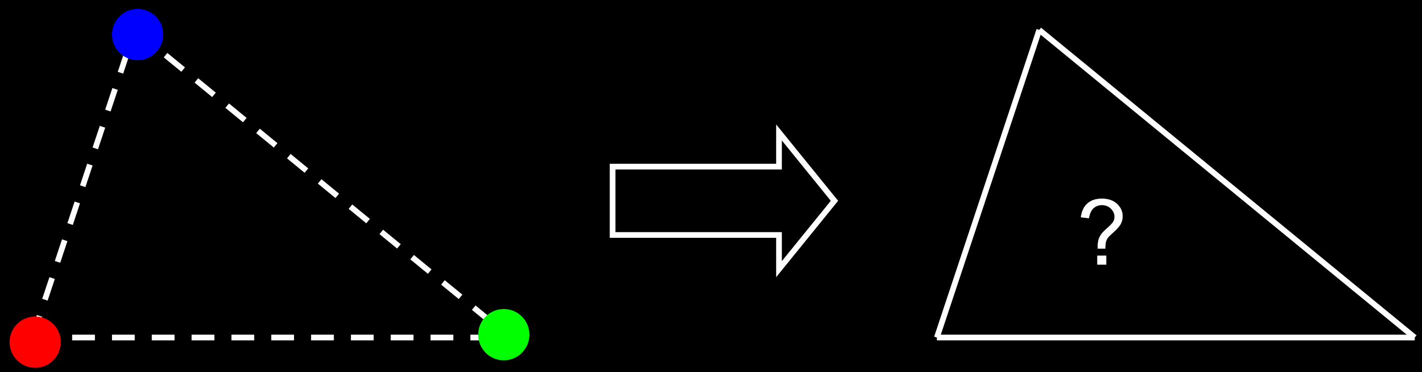
Lighting properties of a human face were captured and face re-rendered; Institute for Creative Technologies

Outline

- Global and Local Illumination
- Normal Vectors
- Light Sources
- Phong Illumination Model
- Polygonal Shading
- Example

Polygonal Shading

- Now we know vertex colors
  - either via OpenGL lighting,
  - or by setting directly via glColor3f if lighting disabled
- How do we shade the interior of the triangle ?

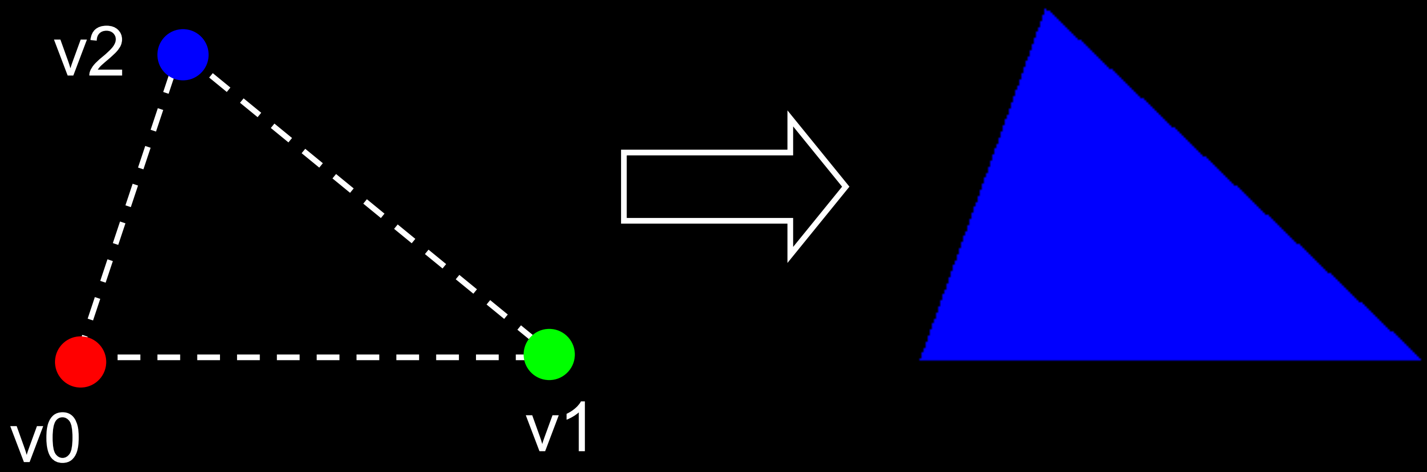


Polygonal Shading

- Curved surfaces are approximated by polygons
- How do we shade?
  - Flat shading
  - Interpolative shading
  - Gouraud shading
  - Phong shading (different from Phong illumination!)

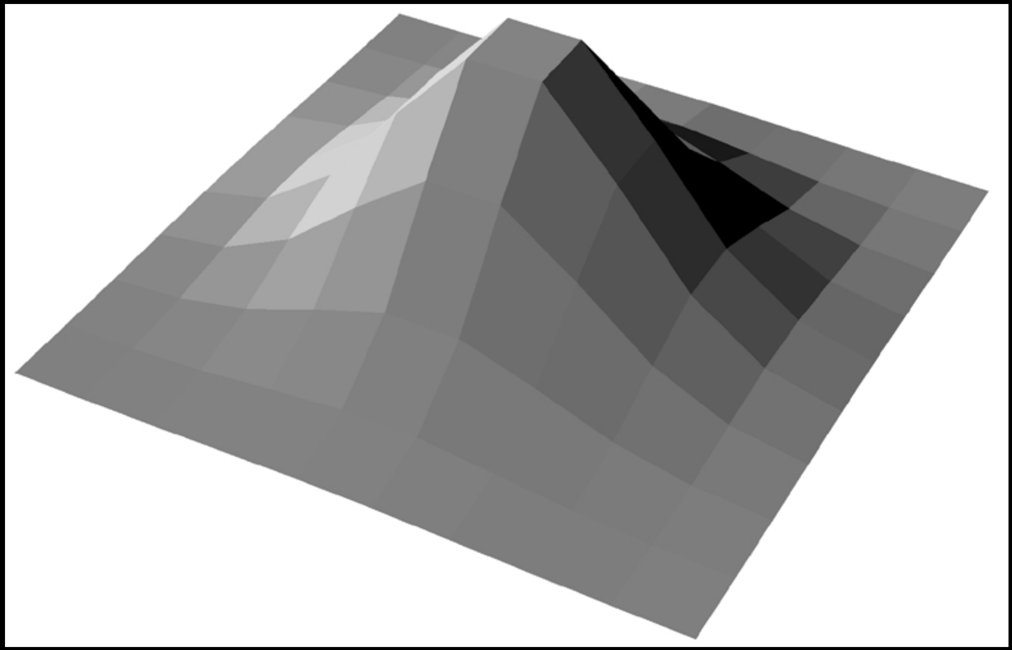
Flat Shading

- Shading constant across polygon
- Core profile: Use interpolation qualifiers in the fragment shader
- Compatibility profile: Enable with glShadeModel(GL_FLAT);
- Color of last vertex determines interior color
- Only suitable for *very* small polygons



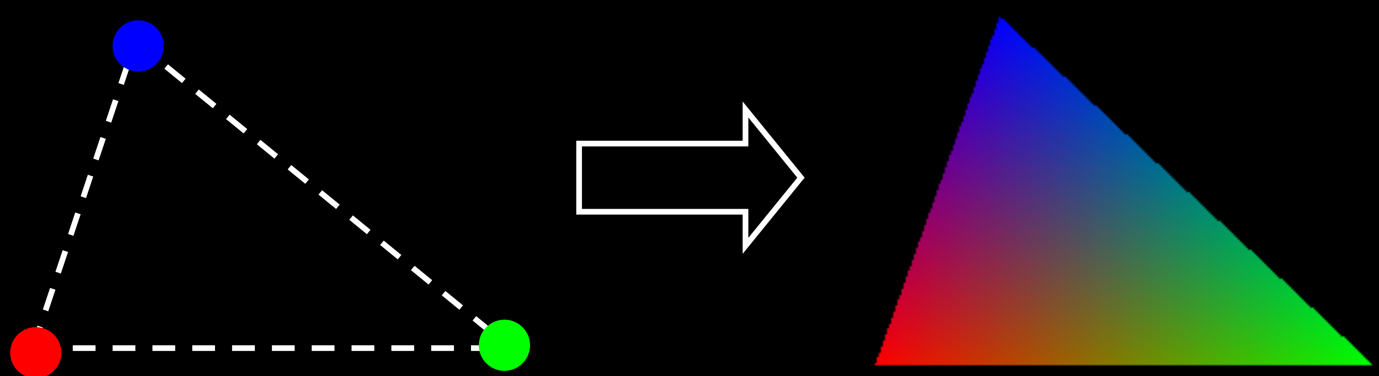
Flat Shading Assessment

- Inexpensive to compute
- Appropriate for objects with flat faces
- Less pleasant for smooth surfaces



Interpolative Shading

- Interpolate color in interior
- Computed during scan conversion (rasterization)
- Core profile: enabled by default
- Compatibiltiy profile: enable with glShadeModel(GL_SMOOTH);
- Much better than flat shading
- More expensive to calculate (but not a problem)



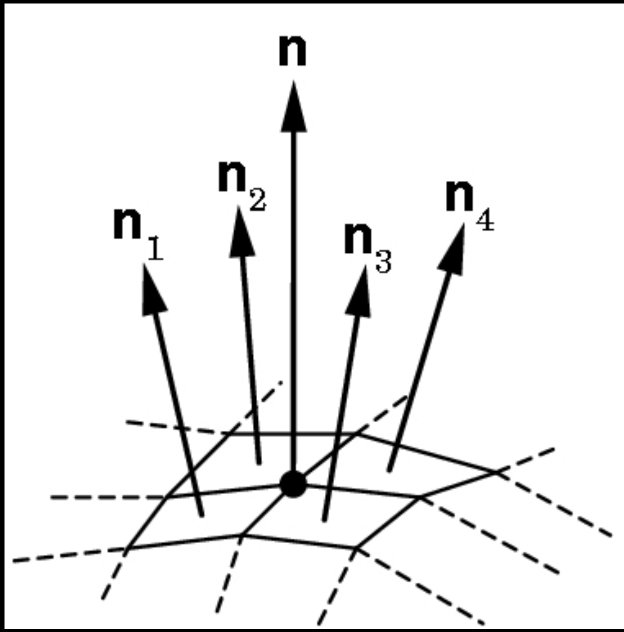
Gouraud Shading

Invented by Henri Gouraud, Univ. of Utah, 1971

- Special case of interpolative shading
- How do we calculate vertex normals for a polygonal surface? Gouraud:
  1. average all adjacent face normals

$$n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1 + n_2 + n_3 + n_4|}$$

  2. use *n* for Phong lighting
  3. interpolate vertex colors into the interior



- Requires knowledge about which faces share a vertex

Data Structures for Gouraud Shading

- Sometimes vertex normals can be computed directly (e.g. height field with uniform mesh)
- More generally, need data structure for mesh
- Key: which polygons meet at each vertex

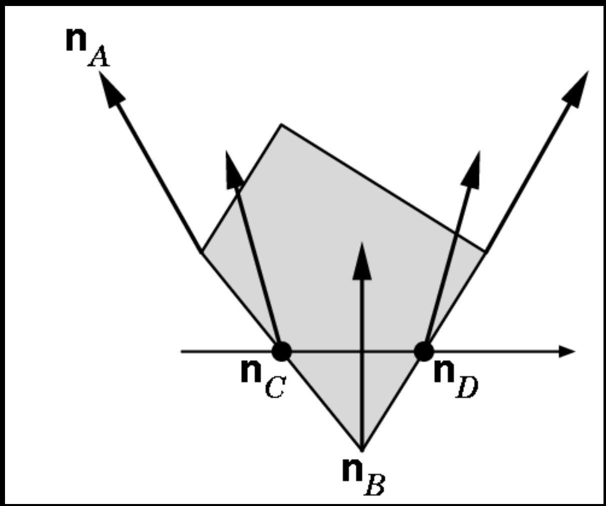
Phong Shading (“per-pixel lighting”)

Invented by Bui Tuong Phong, Univ. of Utah, 1973

- *At each pixel* (as opposed to at each vertex) :
  1. Interpolate *normals* (rather than colors)
  2. Apply Phong lighting to the interpolated normal

- Significantly more expensive

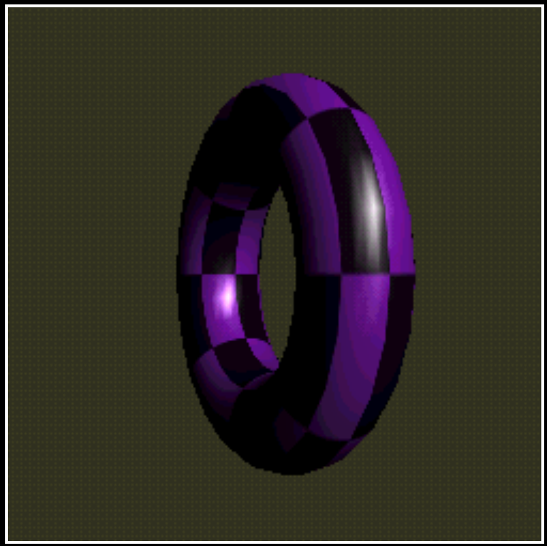
- Done off-line or in GPU shaders (not supported in OpenGL directly)



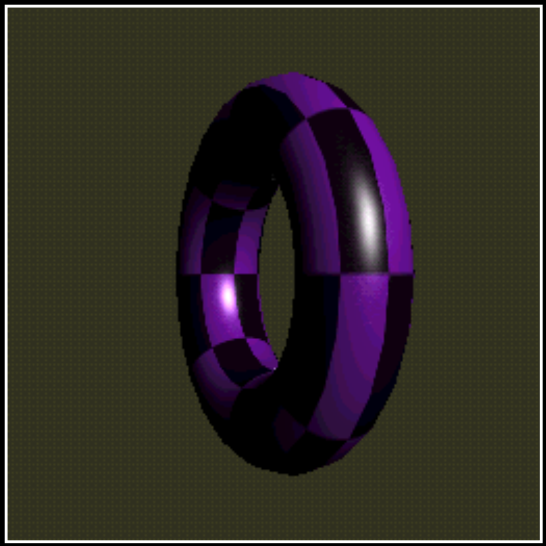
Phong Shading Results

Michael Gold, Nvidia

Single light
Phong Lighting
Gouraud Shading



Two lights
Phong Lighting
Gouraud Shading



Two lights
Phong Lighting
Phong Shading

Outline

- Global and Local Illumination
- Normal Vectors
- Light Sources
- Phong Illumination Model
- Polygonal Shading
- Example

Phong Shader: Vertex Program

```
#version 150

in vec3 position;
in vec3 normal;

out vec3 viewPosition;
out vec3 viewNormal;

uniform mat4 modelViewMatrix;
uniform mat4 normalMatrix;
uniform mat4 projectionMatrix;
```

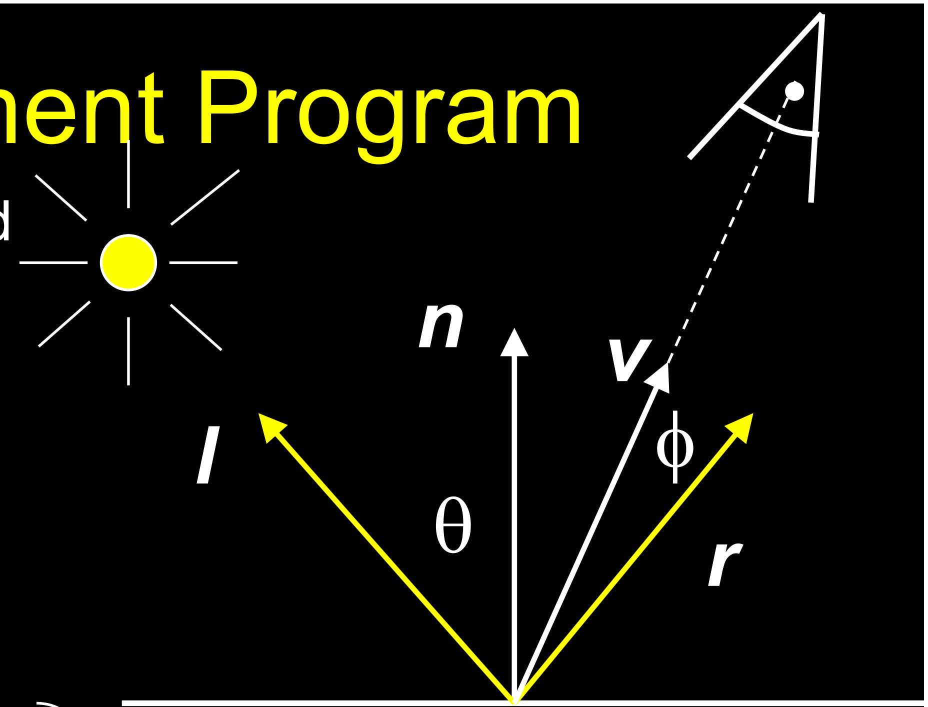
- `in vec3 position;` / `in vec3 normal;` — input vertex position and normal, in world-space
- `out vec3 viewPosition;` / `out vec3 viewNormal;` — vertex position and normal, in view-space; these will be passed to fragment program (interpolated by hardware)
- `uniform mat4 modelViewMatrix;` / `uniform mat4 normalMatrix;` / `uniform mat4 projectionMatrix;` — transformation matrices

Phong Shader: Vertex Program

```
void main()
{
  // view-space position of the vertex
  vec4 viewPosition4 = modelViewMatrix * vec4(position, 1.0f);
  viewPosition = viewPosition4.xyz;

  // final position in the normalized device coordinates space
  gl_Position = projectionMatrix * viewPosition4;
  // view-space normal
  viewNormal = normalize((normalMatrix*vec4(normal, 0.0f)).xyz);
}
```

Phong Shader: Fragment Program




```
in vec3 viewPosition;
in vec3 viewNormal;
```

interpolated from vertex program outputs

```
out vec4 c; // output color

uniform vec4 La; // light ambient
uniform vec4 Ld; // light diffuse
uniform vec4 Ls; // light specular
uniform vec3 viewLightDirection;
```

properties of the directional light

In view space

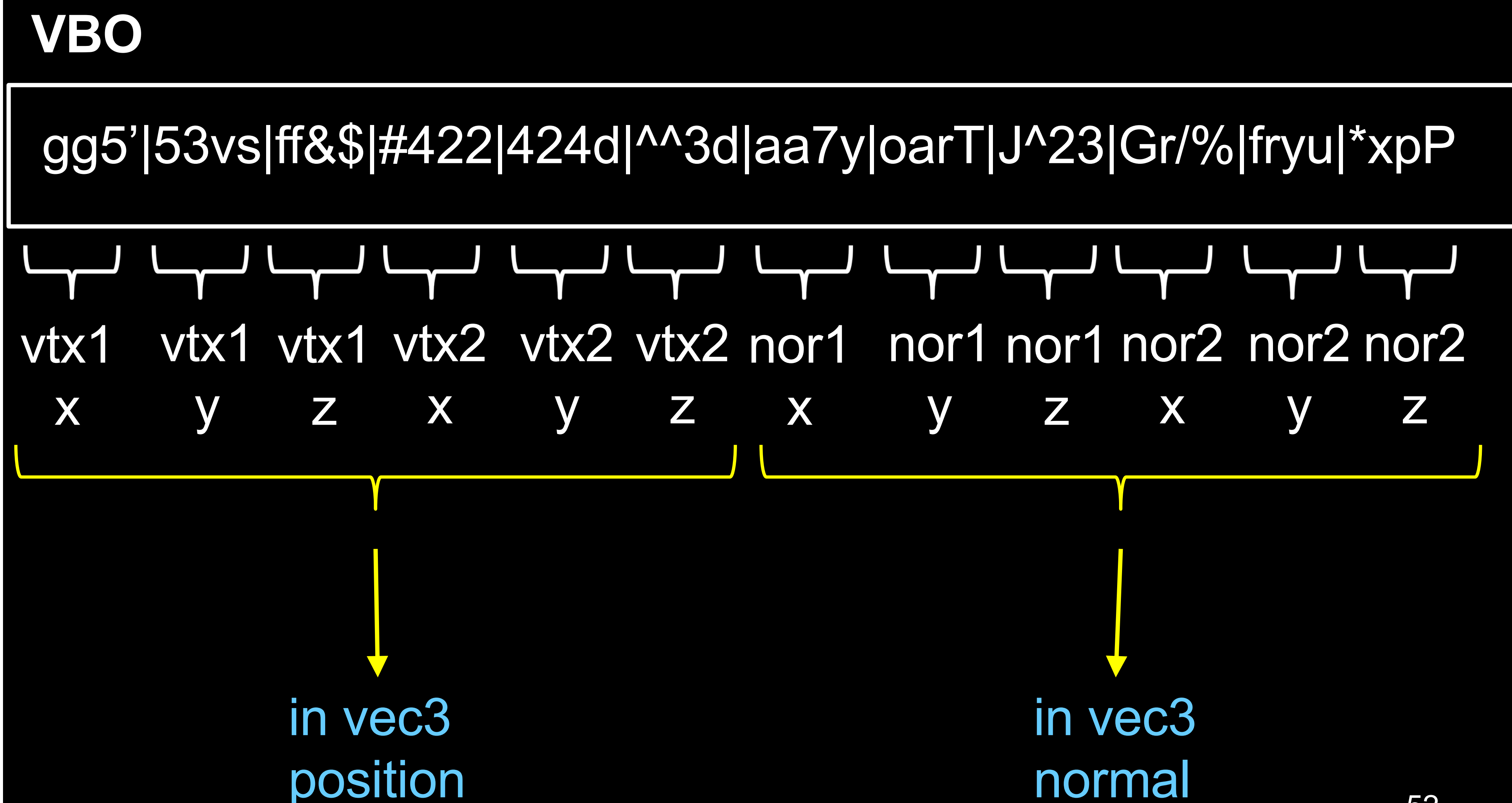
```
uniform vec4 ka; // mesh ambient
uniform vec4 kd; // mesh diffuse
uniform vec4 ks; // mesh specular
uniform float alpha; // shininess
```

mesh optical properties

Phong Shader: Fragment Program

```
void main()
{
  // camera is at (0,0,0) after the modelview transformation
  vec3 eyedir = normalize(vec3(0, 0, 0) - viewPosition);
  // reflected light direction
  vec3 reflectDir = -reflect(viewLightDirection, viewNormal);
  // Phong lighting
  float d = max(dot(viewLightDirection, viewNormal), 0.0f);
  float s = max(dot(reflectDir, eyedir), 0.0f);
  // compute the final color
  c = ka * La + d * kd * Ld + pow(s, alpha) * ks * Ls;
}
```

VBO Layout: positions and normals



VAO code (“normal” shader variable)

During initialization:

```
glBindVertexArray(vao); // bind the VAO

// bind the VBO “buffer” (must be previously created)
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// get location index of the “normal” shader variable
GLuint loc = glGetAttribLocation(program, “normal”);
glEnableVertexAttribArray(loc); // enable the “normal” attribute
const void * offset = (const void*) sizeof(positions);  GLsizei stride = 0;
GLboolean normalized = GL_FALSE;
// set the layout of the “normal” attribute data
glVertexAttribPointer(loc, 3, GL_FLOAT, normalized, stride, offset);
```

Upload the light direction vector to GPU

```
void display()
{
  glClear (GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  openGLMatrix->SetMatrixMode(OpenGLMatrix::ModelView);
  openGLMatrix->LoadIdentity();
  openGLMatrix->LookAt(ex, ey, ez,  fx, fy, fz,  ux, uy, uz);

  float view[16];
  openGLMatrix->GetMatrix(view); // read the view matrix

  // get a handle to the program
  GLuint program = pipelineProgram->GetProgramHandle();
  // get a handle to the viewLightDirection shader variable
  GLint h_viewLightDirection  =
    glGetUniformLocation(program, “viewLightDirection”);
```

Upload the light direction vector to GPU

```
float lightDirection[3] = { 0, 1, 0 }; // the "Sun" at noon
float viewLightDirection[3]; // light direction in the view space
// the following line is pseudo-code:
viewLightDirection = (view * float4(lightDirection, 0.0)).xyz;

// upload viewLightDirection to the GPU
glUniform3fv(h_viewLightDirection, 1, viewLightDirection);

// continue with model transformations
openGLMatrix->Translate(x, y, z);
...

renderBunny(); // render, via VAO
glutSwapBuffers();
}
```

Upload the normal matrix to GPU

```
// in the display function:

// get a handle to the program
GLuint program = pipelineProgram->GetProgramHandle();
// get a handle to the normalMatrix shader variable
GLint h_normalMatrix =
  glGetUniformLocation(program, “normalMatrix”);

float n[16];
matrix->SetMatrixMode(OpenGLMatrix::ModelView);
matrix->GetNormalMatrix(n); // get normal matrix

// upload n to the GPU
GLboolean isRowMajor = GL_FALSE;
glUniformMatrix4fv(h_normalMatrix, 1, isRowMajor, n);
```

Summary

- Global and Local Illumination
- Normal Vectors
- Light Sources
- Phong Illumination Model
- Polygonal Shading
- Example