Outline

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

Global Illumination

- Ray tracing
- Radiosity
- Photon Mapping
- Follow light rays through a scene
- Accurate, but expensive (off-line)

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: \( u \times v \) 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

\[
\begin{bmatrix}
\frac{\partial f}{\partial x} \\
\frac{\partial f}{\partial y} \\
\frac{\partial f}{\partial z}
\end{bmatrix} = \begin{bmatrix} 2x \\ 2y \\ 2z \end{bmatrix} = 2p
\]
- Normalize \( n_0/|n_0| = 2p/2 = p \)
**Reflected Vector**

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

\[
\begin{align*}
  l \cdot n &= \cos(\theta) = n \cdot r \\
  r &= \alpha l + \beta n \\
  \text{Solution: } \alpha &= -1 \text{ and } \beta = 2 (l \cdot n) \\
  r &= 2 (l \cdot n) n - l
\end{align*}
\]

**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

\[
q = \frac{1}{a + b q + c q^2}
\]

- 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 \(\theta\)

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

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**

- \( L_a = k_a L_a \)

- Intensity of ambient light is uniform at every point
- Ambient reflection coefficient \( k_a \), \( 0 \leq k_a \leq 1 \)
- 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 \( k_d \), \( 0 \leq k_d \leq 1 \)
- Angle of incoming light is important

**Lambert’s Law**

Intensity depends on angle of incoming light.

\[
\cos \theta = l \cdot n
\]

\[
I_d = k_d L_d (l \cdot n)
\]

With attenuation:
\[
t_d = \frac{k_d L_d}{a + bq + cq^2} (l \cdot n)
\]

\( q = \) distance to light source, \( L_d = \) diffuse component of light

**Diffuse Light Intensity Depends On Angle Of Incoming Light**

- Recall
- \( l \) = unit vector to light
- \( n \) = unit surface normal
- \( \theta \) = angle to normal
- \( \cos \theta = l \cdot n \)
- \( I_d = k_d L_d (l \cdot n) \)
- With attenuation:
  \( t_d = \frac{k_d L_d}{a + bq + cq^2} (l \cdot n) \)

**Specular Reflection**

- Specular reflection coefficient \( k_s \), \( 0 \leq k_s \leq 1 \)
- Shiny surfaces have high specular coefficient
- Used to model specular highlights
- Does not give mirror effect
  (need other techniques)

\[
l_s = k_s L_s (\cos \phi)^\alpha
\]

\( L_s \) is specular component of light
\( \alpha \) is shininess coefficient
Can add distance term as well

**Specular Reflection**
Shininess Coefficient

\[ I_s = k_s L_s (\cos \phi)^\alpha \]

\( \alpha \) is the shininess coefficient.

\( \alpha \) gives narrower curves.

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

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

\( I = \) unit vector to light \( r = I \) reflected about \( n \)
\( n = \) surface normal \( v = \) vector to viewer

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 \texttt{glColor3f} if lighting disabled
- How do we shade the interior of the triangle?

BRDF

- Bidirectional Reflection Distribution Function
- Must measure for real materials
- Isotropic vs. anisotropic
- Mathematically complex
- Programmable pixel shading

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

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
- Compatibility 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

Phong Lighting
Gouraud Shading

Outline

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

Phong Shader: Vertex Program

```cpp
#version 150

in vec3 position;
in vec3 normal;

out vec3 viewPosition;  // vertex position and normal, in view-space
out vec3 viewNormal;    // final position in the normalized device coordinates space

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

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

    // view-space normal
    viewNormal = normalize((normalMatrix * vec4(normal, 0.0f)).xyz);
}
```

Phong Shader: Fragment Program

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

out vec4 c;  // output color

uniform vec4 lightAmbient;
uniform vec4 lightDiffuse;
uniform vec4 lightSpecular;
uniform vec3 viewLightDirection;
uniform vec4 matKa;
uniform vec4 matKd;
uniform vec4 matKs;
uniform float matKsExp;

void main()
{
    // camera is at (0,0,0) after the modelview transformation
    vec3 eyedir = normalize(vec3(0, 0, -1) - viewPosition);
    vec3 eyedirRefl = -reflect(eyedir, viewNormal);

    float kd = clamp(dot(viewLightDirection, viewNormal), 0.0f, 1.0f);
    float ks = clamp(dot(viewLightDirection, eyedirRefl), 0.0f, 1.0f);

    // compute the final color
    c = matKa * lightAmbient + matKd * kd * lightDiffuse + matKs * pow(ks, matKsExp) * lightSpecular;
}
```
VBO Layout: positions and normals

```plaintext
VBO

gg5'|53vs|f&$|#422|424d|^^3d|aa7y|oarT|J^23|Gr/%|fryu|*

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>vtx1</td>
<td>vtx1</td>
<td>vtx2</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>y</td>
<td>z</td>
<td>x</td>
</tr>
</tbody>
</table>

in vec3 position
```

VAO code ("normal" shader variable)

During initialization:

```plaintext
gBindVertexArray(vao); // bind the VAO
```

```plaintext
// bind the VBO "buffer" (must be previously created)
gBindBuffer(GL_ARRAY_BUFFER, buffer);
```

```plaintext
// get location index of the "normal" shader variable
GLuint loc = glGetUniformLocation(program, "normal");
```

```plaintext
// set the layout of the "normal" attribute data
gVertexAttribPointer(loc, 3, GL_FLOAT, 0, 0, 0);
```

Upload the light direction vector to GPU

```plaintext
void display()
{
geClear(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");

float lightDirection[3] = { 0, 1, 0 }; // the "Sun" at noon
float viewLightDirection[3]; // light direction in the view space

viewLightDirection = (view * float4(lightDirection, 0.0)).xyz;

// upload viewLightDirection to the GPU
gUniform3fv(h_viewLightDirection, 1, viewLightDirection);

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

... renderBunny(); // render, via VAO

```}

Summary

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