**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 one color. We get something like

- But we want

---

**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

---

**Shading**

- Why does the image of a real sphere look like this?

- 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

---

**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

- Translucent surface
- Shadow
- Multiple reflection

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.”
- There are many techniques 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 smoothness and orientation 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.

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.

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.
Phong Shading Model

- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To light 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

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 \( \propto \cos \theta \)
  - \( \cos \theta = \frac{l \cdot n}{|l||n|} \) if vectors normalized
  - There are also three coefficients, \( k_r, k_b, k_g \) that show how much of each color component is reflected

Lambert's Law for Diffuse Reflection

- Purely diffuse object
- \( I = I_r = k_r \cos \theta \)
- \( I_r \): resulting intensity
- \( I_l \): light source intensity
- \( k_r \): (diffuse) surface reflectance coefficient
- \( \theta \): angle between normal & light direction

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

Modeling Specular Reflections

- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased
- \( I \sim k \cos \phi \)
  - \( I \): reflected intensity
  - \( k \): shininess coefficient
  - \( \phi \): angle between normal & incoming direction
  - \( k \): absorption coefficient
The Shininess Coefficient

- Values of $\alpha$ between 100 and 200 correspond to metals.
- Values between 5 and 10 give surfaces that look like plastic.

\[
\alpha = \frac{1}{\cos^2 \phi}
\]

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.

Our Three Basic Components of Illumination

- Diffuse
- Specular
- Ambient

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 $1/(a + bd + cd^2)$ 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:
    - \( k_{dL}, k_{dG}, k_{dB}, k_{sL}, k_{sG}, k_{sB}, k_{aL}, k_{aG}, k_{aB} \)
  - Shininess coefficient \( a \)

Adding up the Components

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

\[
I = k_d I_d \cdot \mathbf{n} + k_s I_s (v \cdot r)^a + k_a I_a
\]

For each color component, we add contributions from all light sources.

Too Intense

With multiple light sources, it is easy to generated values of \( I > 1 \)

One solution is to set the color value to be \( \text{MIN}(I, 1) \)

- An object can change color, saturating towards white
  - Ex. \((0.1, 0.4, 0.8) + (0.5, 0.5, 0.5) = (0.6, 0.9, 1.0)\)

Another solution is to renormalize the intensities to vary from 0 to 1 if one \( I > 1 \).

- Requires calculating all \( I \)'s before rendering anything.
- No over-saturation, but image may be too bright, and contrasts a little off.

Image-processing on image to be rendered (with original \( I \)'s) will produce better results, but is costly.

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

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

Using the halfway vector

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

Computing Reflection Direction

- Angle of incidence = angle of reflection
- Normal, light direction and reflection direction are coplanar
- 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 \( p_0, p_2, p_3 \) or normal \( \mathbf{n} \) and \( p_0 \)
- Normal can be obtained by

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

Normal to Sphere

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

Parametric Form

- For sphere

\[
\begin{align*}
\mathbf{x}(u,v) &= \cos u \sin v \\
\mathbf{y}(u,v) &= \cos u \cos v \\
\mathbf{z}(u,v) &= \sin u
\end{align*}
\]
- Tangent plane determined by vectors

\[
\frac{\partial}{\partial u} = \left[ \frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u} \right] \\
\frac{\partial}{\partial v} = \left[ \frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v} \right]
\]
- Normal given by cross product

\[
\mathbf{n} = \frac{\partial}{\partial u} \times \frac{\partial}{\partial v}
\]
General Case

- We can compute parametric normals for other simple cases
  - Quadrics
  - Parameteric polynomial surfaces
  - Bezier surface patches (Chapter 10)

Shading in WebGL

Objectives

- Introduce the WebGL shading methods
  - Light and material functions on MV.js
  - per vertex vs per fragment shading
  - Where to carry out

- Discuss polygonal shading
  - Flat
  - Smooth
  - Gouraud

WebGL shading

- Need
  - Normals
  - Material properties
  - Lights
  - State-based shading functions have been deprecated (glNormal, glMaterial, glLight)
  - Get computed in application or send attributes to shaders

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 preserved length
- GLSL has a normalization function

Specifying a Point Light Source

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

```javascript
var diffuse0 = vec4(1.0, 0.05, 0.05, 1.0);
var ambient0 = vec4(0.5, 0.5, 0.5, 1.0);
var specular0 = vec4(1.0, 1.0, 1.0, 1.0);
var 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 finite location
  - If w = 0.0, we are specifying a parallel source with the given direction vector
- The coefficients in distance terms are usually quadratic \(1/(a + bd + c \cdot d^2)\) where \(d\) is the distance from the point being rendered to the light source

Spotlights

- Derive from point source
  - Direction
  - Cutoff
  - Attenuation Proportional to \(\cos^\phi\)

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 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 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

Light Properties

```
var lightPosition = vec4(1.0, 1.0, 1.0, 0.0);
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0);
var lightDiffuse = vec4(1.0, 1.0, 1.0, 1.0);
var lightSpecular = vec4(1.0, 1.0, 1.0, 1.0);
```

Material Properties

- Material properties should match the terms in the light model
- Reflectivities
- w component gives opacity (alpha)

```
var materialAmbient = vec4(1.0, 0.0, 1.0, 1.0);
var materialDiffuse = vec4(1.0, 0.8, 0.0, 1.0);
var materialSpecular = vec4(1.0, 0.8, 0.0, 1.0);
var materialShininess = 100.0;
```
Using MV.js for Products

```javascript
var ambientProduct = mult(lightAmbient, materialAmbient);
var diffuseProduct = mult(lightDiffuse, materialDiffuse);
var specularProduct = mult(lightSpecular, materialSpecular);

var t1 = subtract(vertices[b], vertices[a]);
var t2 = subtract(vertices[c], vertices[b]);
var normal = cross(t1, t2);
normal = vec3(normal);

pointsArray.push(vertices[a]);
normalsArray.push(normal);
```

Adding Normals for Quads

```javascript
function quad(a, b, c, d) {
    var t1 = subtract(vertices[b], vertices[a]);
    var t2 = subtract(vertices[c], vertices[b]);
    var normal = cross(t1, t2);
    normal = vec3(normal);
    normalsArray.push(normal);
}
```

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

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

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)
**Polygon Normals**

- Triangles have a single normal
  - Shades at the vertices as computed by the Phong model can almost be the same
  - Identical for a distant viewer (default) or if there is no specular component
- Consider model of sphere
- Want different normals at each vertex

**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 normals around a mesh vertex
  \[
  \mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|}
  \]

**Normal for Triangle**

Plane: \( \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0 \)

\[
\mathbf{n} = \frac{(\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)}{|n_1 + n_2 + n_3|}
\]

Note that right-hand rule determines outward face

**Simple Mesh Format (SMF)**

- Michael Garland
  - http://graphics.cs.uiuc.edu/~garland/
- Triangle data
  - List of 3D vertices
  - List of references to vertex array
  - Define faces (triangles)
- Vertex indices begin at 1

**Calculating Normals**

- Create vector structure (for normals) same size as vertex structure
- For each face
  - Calculate unit normal
  - Add to normals structure using vertex indices
- Normalize all the normals
Gouraud and Phong Shading

- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Compute modified Phong shading model at each vertex
  - Interpolate vertex shades across each polygon

- Phong Shading
  - Find averaged vertex normals
  - Interpolate vertex normals across polygon
  - Compute modified Phong model at each fragment

Comparison

- If the polygon mesh approximates surfaces with 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

Comparison

Per Vertex and Per Fragment Shaders

// vertex shader
attribute vec3 vPosition;
attribute vec3 vNormal;
varying vec3 fColor;
uniform vec3 ambientProduct;
diffuseProduct;
specularProduct;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec3 lightPosition;
uniform float shininess;
void main()
{
    
}

Adding up the Components

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

\[ I = k_d I_d \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^s + k_a I_a \]

For each color component we add contributions from all light sources

Vertex Lighting Shaders I
Vertex Lighting Shaders II

```
// Transform vertex into eye coordinates
vec3 pos = (modelViewMatrix * vec4(vPosition, 1.0)).xyz;

// If light defined in model coordinates,
// transform with modelViewMatrix
// The following assumes light defined in eye/camera frame
vec3 L = normalize(lightPosition - pos);
vec3 E = normalize(-pos);
vec3 H = normalize(L + E);

// Transform vertex normal into eye coordinates
vec3 N = normalize((modelViewMatrix * vec4(vNormal, 0.0)).xyz);
```

Vertex Lighting Shaders III

```
// Compute terms in the illumination equation
vec3 ambient = ambientProduct;
float diffuseTerm = max(dot(L, N), 0.0);
vec3 diffuse = diffuseTerm * diffuseProduct;
float specularTerm = pow(max(dot(N, H), 0.0), shininess);
vec3 specular = specularTerm * specularProduct;
if (dot(L, N) < 0.0) specular = vec3(0.0, 0.0, 0.0);
gl_Position = Projection * vec4(pos, 1.0);

fColor = vec4(ambient + diffuse + specular, 1.0);
```

Vertex Lighting Shaders IV

```
// fragment shader
precision mediump float;
varying vec4 fColor;
void main()
{
    gl_FragColor = fColor;
}
```

Fragment Lighting Shaders I

```
// vertex shader
attribute vec3 vPosition;
attribute vec3 vNormal;
varying vec3 N, L, E;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec3 lightPosition;
```

Fragment Lighting Shaders II

```
void main()
{
    vec3 pos = (modelViewMatrix * vec4(vPosition, 1.0)).xyz;
    L = normalize(lightPosition - pos);
    E = -pos;
    N = normalize((modelViewMatrix * vec4(vNormal, 0.0)).xyz);
    gl_Position = projectionMatrix * vec4(pos, 1.0);
}
```

Fragment Lighting Shaders III

```
// fragment shader
precision mediump float;
uniform vec3 ambientProduct;
uniform vec3 diffuseProduct;
uniform vec3 specularProduct;
uniform float shininess;
uniform vec3 N, L, E;
void main()
{
}
```
void main()
{
  vec3 color;
  vec3 H = normalize( L + E );
  vec3 ambient = ambientProduct;
  float diffuseTerm = max( dot(L, N), 0.0 );
  vec3 diffuse = diffuseTerm*diffuseProduct;
  float specularTerm = specularTerm * specularProduct;
  vec3 specular = specularTerm * specularProduct;
  if ( dot(L, N) < 0.0 ) specular = vec3(0.0, 0.0, 0.0);
  color = ambient + diffuse + specular;
  gl_FragColor = vec4(color, 1.0);
}

• Write code for computing average surface normal per vertex
• Use normal as color (don’t forget the absolute value) and interpolate across your model. This should look like your HW5 output, but with interpolated colors
• Implement Phong shading model in vertex shader with one light. This is produces Gouraud shading algorithm.

• Light and specular color should be white
• First light should be in camera coordinates and placed above camera (e.g. (0,3,0))
• Test with sphere model and shininess of 100. Should get good results with a distinct specular highlight
• Try the lo-res and hi-res bunny models and note the differences in the specular highlights between the two.

• Next implement Phong shading algorithm in fragment shader
• Be sure to use the same material and lighting values for both shading algorithms
• Provide a way to switch between the two
• Provide interface for changing the objects material properties. The three choices should be significantly different.
• Add a second light