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

- 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
  - Exist 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 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

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

**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 \( \sim \cos \theta_i \)
  - \( \cos \theta_i = l \cdot n \) if vectors normalized
- There are also three coefficients, \( k_r, k_b, k_g \) that show how much of each color component is reflected

**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.

\[ I_r \sim k_s I \cos^\alpha \phi \]

The Shininess Coefficient

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

\[ \cos^\alpha \phi \]

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.

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 $\alpha$

Adding up the Components

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

$$I = k_d I_d l \cdot n + k_s l_s (v \cdot r)^\alpha + k_a I_a$$

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 \( v \)
  \[
  h = \frac{l + v}{|l + v|}
  \]

Using the halfway angle

- Replace \((v \cdot r)^\alpha\) by \((n \cdot h)^\beta\)
- \( \beta \) is chosen to match shineness
- 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

- \( l \) and \( v \) are specified by the application
- Can compute \( r \) from \( l \) and \( n \)
- Problem is determining \( n \)
- For simple surfaces, it can be determined
  – how depends on underlying representation of surface
- OpenGL leaves determination of normal to application
  – Exception for GLU quadrics and Bezier surfaces
Plane Normals

- Equation of plane: $ax+by+cz+d = 0$
- From Chapter 4 we know that plane is determined by three points $p_0$, $p_2$, $p_3$ or normal $n$ and $p_0$
- Normal can be obtained by

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

Normal to Sphere

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

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
  $$\frac{\partial p}{\partial u} = \left[\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right]^T$$
  $$\frac{\partial p}{\partial v} = \left[\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}\right]^T$$
- Normal given by cross product
  $$n = \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v}$$

General Case

- We can compute parametric normals for other simple cases
  - Quadrics
  - Parameteric polynomial surfaces
    - Bezier surface patches (Chapter 11)
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 preserve length
  - `glEnable(GL_NORMALIZE)` allows for autonormalization at a performance penalty

Normal for Triangle

\[
\text{plane } \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0
\]
\[
\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)
\]
\[
\text{normalize } \mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|
\]

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

```c
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};
```

```c
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
gllightv(GL_LIGHT0, GL_POSITION, light0_pos);
gllightv(GL_LIGHT0, GL_AMBIENT, ambient0);
gllightv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
gllightv(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
  ```c
  a = 0.80;
gllightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a);
  ```

Spotlights

- Use `glLightv` 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
  ```c
  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

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 $n = p$
- 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

\[ n = \frac{(n_1 + n_2 + n_3 + n_4)}{|n_1 + n_2 + n_3 + n_4|} \]

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

Comparison

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

Next time: Scene graphs, Object oriented modeling