Shading

Brian Curless
CSE 457
Spring 2014

Basic 3D graphics

With affine matrices, we can now transform virtual 3D objects in their local coordinate systems into a global (world) coordinate system:

To synthesize an image of the scene, we also need to add light sources and a viewer/camera:

Pinhole camera

To create an image of a virtual scene, we need to define a camera, and we need to model lighting and shading. For the camera, we use a pinhole camera.

The image is rendered onto an image plane (usually in front of the camera).

Viewing rays emanate from the center of projection (COP) at the center of the pinhole.

The image of an object point $P$ is at the intersection of the viewing ray through $P$ and the image plane.

But is $P$ visible? This the problem of hidden surface removal (a.k.a., visible surface determination). We'll consider this problem later.
Shading

Next, we'll need a model to describe how light interacts with surfaces.

Such a model is called a shading model.

Other names:
- Lighting model
- Light reflection model
- Local illumination model
- Reflectance model
- BRDF

An abundance of photons

Given the camera and shading model, properly determining the right color at each pixel is extremely hard.

Look around the room. Each light source has different characteristics. Trillions of photons are pouring out every second.

These photons can:
- interact with molecules and particles in the air (“participating media”)
- strike a surface and
  - be absorbed
  - be reflected (scattered)
  - cause fluorescence or phosphorescence
- interact in a wavelength-dependent manner
- generally bounce around and around

Our problem

We're going to build up to a approximations of reality called the {Phong and Blinn-Phong illumination models}.

They have the following characteristics:
- not physically correct
- gives a “first-order” approximation to physical light reflection
- very fast
- widely used

In addition, we will assume local illumination, i.e., light goes: light source -> surface -> viewer.

No interreflections, no shadows.

Setup...

Given:
- a point \( P \) on a surface visible through pixel \( p \)
- The normal \( N \) at \( P \)
- The lighting direction, \( L \), and (color) intensity, \( I_l \), at \( P \)
- The viewing direction, \( V \), at \( P \)
- The shading coefficients at \( P \)

Compute the color, \( I \), of pixel \( p \).

Assume that the direction vectors are normalized:
\[
||N|| = ||L|| = ||V|| = 1
\]
“Iteration zero”

The simplest thing you can do is...

Assign each polygon a single color:

\[ I = k_e \]

where

- \( I \) is the resulting intensity
- \( k_e \) is the emissivity or intrinsic shade associated with the object

This has some special-purpose uses, but not really good for drawing a scene.

[Note: \( k_e \) is omitted in Angel.]

“Iteration one”

Let’s make the color at least dependent on the overall quantity of light available in the scene:

\[ I = k_a + k_d \int_{\Omega} \]

- \( k_a \) is the ambient reflection coefficient.
  - really the reflectance of ambient light
  - “ambient” light is assumed to be equal in all directions
- \( I_{\Omega} \) is the ambient light intensity.

Physically, what is “ambient” light?

[Note: Angel uses \( L_a \) instead of \( I_{\Omega} \)]

Wavelength dependence

Really, \( k_a, k_d \) and \( I_{\Omega} \) are functions over all wavelengths \( \lambda \).

Ideally, we would do the calculation on these functions. For the ambient shading equation, we would start with:

\[ I(\lambda) = k_a(\lambda) I_{\Omega} (\lambda) \]

then we would find good RGB values to represent the spectrum \( I(\lambda) \).

Traditionally, though, \( k_a \) and \( I_{\Omega} \) are represented as RGB triples, and the computation is performed on each color channel separately:

\[ I^p = k_a^p I_{\Omega}^p \]
\[ I^g = k_a^g I_{\Omega}^g \]
\[ I^b = k_a^b I_{\Omega}^b \]

Diffuse reflection

Let’s examine the ambient shading model:

- objects have different colors
- we can control the overall light intensity
  - what happens when we turn off the lights?
  - what happens as the light intensity increases?
  - what happens if we change the color of the lights?

So far, objects are uniformly lit.

- not the way things really appear
- in reality, light sources are localized in position or direction

Diffuse, or Lambertian reflection will allow reflected intensity to vary with the direction of the light.
Diffuse reflectors

Diffuse reflection occurs from dull, matte surfaces, like latex paint, or chalk.

These diffuse or Lambertian reflectors reradiate light equally in all directions.

Picture a rough surface with lots of tiny microfacets.

The microfacets and pigments distribute light rays in all directions.

Embedded pigments are responsible for the coloration of diffusely reflected light in plastics and paints.

Note: the figures above are intuitive, but not strictly (physically) correct.

Diffuse reflectors, cont.

The reflected intensity from a diffuse surface does not depend on the direction of the viewer. The incoming light, though, does depend on the direction of the light source:

\[
\cos \theta = \frac{\mathbf{N} \cdot \mathbf{L}}{||\mathbf{N}|| \cdot ||\mathbf{L}||}
\]

\[
\mathbf{N} \cdot \mathbf{L} = \cos \theta
\]

\[d\mathbf{A} = d\mathbf{A} \cos \theta \]

\[
I \sim (\cos \theta)^+ = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{else} \end{cases}
\]

“Iteration two”

The incoming energy is proportional to \(\cos \theta\), giving the diffuse reflection equations:

\[
I = k_s + k_l \mathbf{L} + k_d \mathbf{L} \cdot \mathbf{N} \cdot \mathbf{L} 
\]

where:

- \(k_s\) is the diffuse reflection coefficient
- \(k_l\) is the (color) intensity of the light source
- \(\mathbf{N}\) is the normal to the surface (unit vector)
- \(\mathbf{L}\) is the direction to the light source (unit vector)
- \(\mathbf{B}\) prevents contribution of light from below the surface:

\[
B = \begin{cases} 1 & \text{if } \mathbf{N} \cdot \mathbf{L} > 0 \\ 0 & \text{if } \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases}
\]

[Note: Angel uses \(L_i\) instead of \(I_i\) and \(f\) instead of \(B\).]
Specular reflection

Specular reflection accounts for the highlight that you see on some objects. It is particularly important for smooth, shiny surfaces, such as:

- metal
- polished stone
- plastics
- apples
- skin

Properties:

- Specular reflection depends on the viewing direction \( \mathbf{V} \).
- For non-metals, the color is determined solely by the color of the light.
- For metals, the color may be altered (e.g., brass)

For a perfect mirror reflector, light is reflected about \( \mathbf{N} \), so

\[
I = \begin{cases} 
1 & \text{if } \mathbf{V} \cdot \mathbf{R} > 0 \\
0 & \text{otherwise}
\end{cases}
\]

For a near-perfect reflector, you might expect the highlight to fall off quickly with increasing angle \( \phi \).

Also known as:

- "rough specular" reflection
- "directional diffuse" reflection
- "glossy" reflection

Phong specular reflection

One way to get this effect is to take \( \mathbf{R} \cdot \mathbf{V} \), raised to a power \( n_s \).

As \( n_s \) gets larger,

- the dropoff becomes (more/less) gradual
- gives a (larger/smaller) highlight
- simulates a (more/less) mirror-like surface

Phong specular reflection is proportional to:

\[
I_{\text{specular}} \sim \mathbf{B}(\mathbf{R} \cdot \mathbf{V})^{n_s}
\]

where \( \langle x \rangle_s = \max(0, x) \), \( \mathbf{B} \cos \phi \)

Blinn-Phong specular reflection

A common alternative for specular reflection is the Blinn-Phong model (sometimes called the modified Phong model.)

We compute the vector halfway between \( \mathbf{L} \) and \( \mathbf{V} \) as:

\[
\mathbf{H} = \mathbf{L} + \mathbf{V} = \mathbf{R}
\]

Analogous to Phong specular reflection, we can compute the specular contribution in terms of \( \mathbf{N} \cdot \mathbf{H} \), raised to a power \( n_s \):

\[
I_{\text{specular}} = \mathbf{B}(\mathbf{N} \cdot \mathbf{H})^{n_s}
\]

where, again, \( \langle x \rangle_s = \max(0, x) \).
"Iteration three"

The next update to the Blinn-Phong shading model is then:

\[
I = k_r + k_f f_a + \beta [k_s (N \cdot L) + k_r (N \cdot H)]
\]

where:

- $k_r$ is the **specular reflection coefficient**
- $n_i$ is the **specular exponent** or shininess
- $H$ is the unit halfway vector between $L$ and $V$, where $V$ is the viewing direction.

[Note: Angel uses $\alpha$ instead of $n_i$, and maintains a separate $L_s$ and $L_o$ instead of a single $L$. This choice reflects the flexibility available in OpenGL.]

---

**Directional lights**

The simplest form of lights supported by renderers are ambient, directional, and point. Spotlights are also supported often as a special form of point light.

We’ve seen ambient light sources, which are not really geometric.

**Directional light** sources have a single direction and intensity associated with them.

Using affine notation, what is the homogeneous coordinate for a directional light?

---

**Point lights**

The direction of a **point light** sources is determined by the vector from the light position to the surface point.

\[
\frac{E - P}{\|E - P\|} = L
\]

Physics tells us the intensity must drop off inversely with the square of the distance:

\[
f_{\text{atten}} = \frac{1}{r^2}
\]

Sometimes, this distance-squared dropoff is considered too "harsh." A common alternative is:

\[
f_{\text{atten}} = \frac{1}{a + Br + Cr^2}
\]

with user-supplied constants for $a$, $b$, and $c$.

Using affine notation, what is the homogeneous coordinate for a point light?

---

**Spotlights**

We can also apply a **directional attenuation** of a point light source, giving a **spotlight** effect.

A common choice for the spotlight intensity is:

\[
f_{\text{spot}} = \begin{cases} 
(L \cdot S)^c & \text{if } L \cdot S \leq \cos \beta \\
0 & \text{otherwise}
\end{cases}
\]

where:

- $L$ is the direction to the point light.
- $S$ is the center direction of the spotlight.
- $\beta$ is the cutoff angle for the spotlight.
- $c$ is the angular falloff coefficient.
“Iteration four”

Since light is additive, we can handle multiple lights by taking the sum over every light.

Our equation is now:

\[ I = k_a + k_d + \sum_{j} \left( \frac{l_j}{a_j + b_j + c_j} \right) \frac{L}{L} \cdot B_i \left[ k_s (N \cdot L) + k_v (N \cdot H) \right] \]

This is the Phong illumination model.

Which quantities are spatial vectors?

Which are RGB triples?

Which are scalars?

Shading in OpenGL

The OpenGL lighting model allows you to associate different lighting colors according to material properties they will influence.

Thus, our original shading equation:

\[ I = k_a + k_d + \sum_{j} \frac{1}{a_j + b_j + c_j} \frac{l_j}{L} \cdot B_i \left[ k_s (N \cdot L) + k_v (N \cdot H) \right] \]

becomes:

\[ I = k_a + k_d + \sum_{j} \frac{1}{a_j + b_j + c_j} \left[ k_s l_j (N \cdot L) + k_v l_j (N \cdot H) \right] \]

where you can have a global ambient light with intensity \( I_a \), in addition to having an ambient light intensity \( I_{cd} \) associated with each individual light, as well as separate diffuse and specular intensities, \( I_{cd} \) and \( I_{cs} \), respectively.

Materials in OpenGL

The OpenGL code to specify the surface shading properties is fairly straightforward. For example:

```c
GLfloat ke[] = { 0.1, 0.15, 0.05, 1.0 };  
GLfloat ka[] = { 0.1, 0.15, 0.1, 1.0 };  
GLfloat kd[] = { 0.3, 0.3, 0.2, 1.0 };  
GLfloat ks[] = { 0.2, 0.2, 0.2, 1.0 };  
GLfloat ns[] = { 50.0 };  
glMaterialfv(GL_FRONT, GL_EMISSION, ke);  
glMaterialfv(GL_FRONT, GL_AMBIENT, ka);  
glMaterialfv(GL_FRONT, GL_DIFFUSE, kd);  
glMaterialfv(GL_FRONT, GL_SPECULAR, ks);  
glMaterialfv(GL_FRONT, GL_SHININESS, ns);  
```

Notes:

- The GL_FRONT parameter tells OpenGL that we are specifying the materials for the front of the surface.
- Only the alpha value of the diffuse color is used for blending. It's usually set to 1.

Shading in OpenGL, cont’d

In OpenGL, this equation, for one light source (the 0th) is specified something like:

```c
GLfloat la[] = { 0.2, 0.2, 0.2, 1.0 };  
GLfloat la0[] = { 0.1, 0.1, 0.1, 1.0 };  
GLfloat la0[] = { 1.0, 1.0, 1.0, 0.1 };  
GLfloat la0[] = { 1.0, 1.0, 1.0, 0.1 };  
GLfloat pos[] = { 1.0, 1.0, 1.0, 0.0 };  
GLfloat a[] = { 1.0 };  
GLfloat b[] = { 0.5 };  
GLfloat c[] = { 0.25 };  
GLfloat s[] = { -1.0, -1.0, 0.0 };  
GLfloat beta[] = { 45 };  
GLfloat w[] = { 2 };  
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, la);  
glLightfv(GL_LIGHT0, GL_AMBIENT, la0);  
glLightfv(GL_LIGHT0, GL_DIFFUSE, la0);  
glLightfv(GL_LIGHT0, GL_SPECULAR, la0);  
glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a);  
glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION, b);  
glLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, c);  
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, s);  
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, beta);  
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, w);  
```
Shading in OpenGL, cont’d

Notes:

You can have as many as GL_MAX_LIGHTS lights in a scene. This number is system-dependent.

For directional lights, you specify a light direction, not position, and the attenuation and spotlight terms are ignored.

The directions of directional lights and spotlights are specified in the coordinate systems of the lights, not the surface points as we’ve been doing in lecture.

Rasterization

After projecting the vertices, graphics hardware “smears” vertex properties across the interior of the triangle in a process called rasterization.

Smearing the z-values and using a Z-buffer will enable the graphics hardware to determine if a point inside a triangle is visible. (More on this in another lecture.)

If we have stored colors at the vertices, then we can smear these as well.

3D Geometry in the Graphics Hardware Pipeline

Graphics hardware applies transformations to bring the objects and lighting into the camera’s coordinate system:

The geometry is assumed to be made of triangles, and the vertices are projected onto the image plane.

Shading the interiors of triangles

We will be computing colors using the Blinn-Phong lighting model.

Let’s assume (as graphics hardware does) that we are working with triangles.

How should we shade the interiors of triangles?
Shading with per-face normals

Assume each face has a constant normal:

\[ I = K_d (N \cdot L)^m \]

For a distant viewer and a distant light source and constant material properties over the surface, how will the color of each triangle vary?

It won't vary
\( \Rightarrow \) faceted appearance

Gouraud interpolation

To get a smoother result that is easily performed in hardware, we can do Gouraud interpolation.

Here's how it works:

1. Compute normals at the vertices. (More on this in a moment…)
2. Shade only the vertices.
3. Interpolate the resulting vertex colors.
**Gouraud interpolation artifacts**

Gouraud interpolation has significant limitations.

1. If the polygonal approximation is too coarse, we can miss specular highlights.

   ![Gouraud interpolation diagram]

2. We will encounter **Mach banding** (derivative discontinuity enhanced by human eye).

This is what graphics hardware does by default. A substantial improvement is to do...

---

**Phong interpolation**

To get an even smoother result with fewer artifacts, we can perform **Phong interpolation**.

Here’s how it works:

1. Compute normals at the vertices.
2. Interpolate normals and normalize.
3. Shade using the interpolated normals.

![Phong interpolation diagram]

---

**Gouraud vs. Phong interpolation**

[Williams and Siegel 1990]

---

**Default pipeline: Gouraud interpolation**

**Default vertex processing:**
- \( L \leftarrow \text{determine lighting direction} \)
- \( V \leftarrow \text{determine viewing direction} \)
- \( N \leftarrow \text{normalize}(n_i) \)
- \( \text{Color}_{\text{gouraud}} \leftarrow \text{shade with } L, N, k_d, k_s, n_i \)
- attach \( \text{Color}_{\text{gouraud}} \) to vertex as "varying"
- \( v_i \leftarrow \text{project } v \text{ to image} \)

**Primitives:** \( v_1, v_2, v_3 \rightarrow \text{triangle} \)

**Fragment processing:**
- color \( \leftarrow \text{Color}_{\text{gouraud}} \)

---
Programmable pipeline:
Phong-interpolated normals!

Vertex shader:
- Attach \( r_v \) to vertex as "varying"
- Attach \( v_v \) to vertex as "varying"
- \( v_i \leftarrow \text{project } v \text{ to image} \)
- Can have varying \( K_d, K_s, v_s \)

\[ v'_1, v'_2, v'_3 \leftarrow \text{triangle} \]

Fragment shader:
- \( L \leftarrow \text{determine lighting direction} \)
- \( V \leftarrow \text{determine viewing direction} \)
- \( N \leftarrow \text{normalize}(n_v^i) \)
- \( \text{color} \leftarrow \text{shade with } L, V, N, K_d, K_s, v_s, n_v^i \)

Choosing Blinn-Phong shading parameters

Experiment with different parameter settings. To get you started, here are a few suggestions:
- Try \( n_s \) in the range \([0, 100]\)
- Try \( K_d + K_s + K_v < 1 \)
- Use a small \( K_v (-0.1) \)

<table>
<thead>
<tr>
<th>Material</th>
<th>( n_s )</th>
<th>( K_d )</th>
<th>( K_s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal</td>
<td>large</td>
<td>small</td>
<td>large</td>
</tr>
<tr>
<td>Plastic</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Planet</td>
<td>0</td>
<td>varying</td>
<td>0</td>
</tr>
</tbody>
</table>

BRDF

The diffuse+specular parts of the Blinn-Phong illumination model are a mapping from light to viewing directions:

\[
f_s = f_s(N, L) + f_s(N, V)
\]

\[
f_s = f_s(L, V)
\]

The mapping function \( f_s \) is often written in terms of incoming (light) directions \( \omega_{in} \) and outgoing (viewing) directions \( \omega_{out} \):

\[
f_s(\omega_{in}, \omega_{out}) \quad \text{or} \quad f_s(\omega_{out} \rightarrow \omega_{in})
\]

This function is called the Bi-directional Reflectance Distribution Function (BRDF).

Here's a plot with \( \omega_{out} \) held constant:

\[
f_s(\omega_{in}, \omega_{out})
\]

BRDF's can be quite sophisticated...

More sophisticated BRDF's

[Cook and Torrance, 1982]

Anisotropic BRDF's (Westin, Arvo, Torrance 1992)

Artistics BRDF's (Gooch)
Summary

You should understand the equation for the Blinn-Phong lighting model described in the "Iteration Four" slide:

• What is the physical meaning of each variable?
• How are the terms computed?
• What effect does each term contribute to the image?
• What does varying the parameters do?

You should also understand the differences between faceted, Gouraud, and Phong interpolated shading.

And you should understand how to compute the normal to a surface of revolution.