Why Do We Care About Lighting?

Lighting “dis-ambiguates” 3D scenes

Without lighting

With lighting

The Surface Normal

A surface normal is a vector perpendicular to the surface.

Sometimes surface normals are defined or computed per-face, like this:

\[ n = (P_1 - P_0) \times (P_2 - P_0) \]

Sometimes they are defined per-vertex, like this, to best approximate the underlying surface that the face is representing.

Setting a Per-Face Surface Normal in OpenGL

```c
glMatrixMode( GL_MODELVIEW ) ;
glTranslatef( tx, ty, tz ) ;
glRotatef( degrees, ax, ay, az ) ;
glScalef( sx, sy, sz ) ;
glNormal3f( nx, ny, nz ) ;
glColor3f( r, g, b ) ;
gBegin( GL_TRIANGLES ) ;
gVertex3f( x0, y0, z0 ) ;
gVertex3f( x1, y1, z1 ) ;
gVertex3f( x2, y2, z2 ) ;
gEnd( ) ;
```

Setting Per-Vertex Surface Normals in OpenGL

```c
glMatrixMode( GL_MODELVIEW ) ;
glTranslatef( tx, ty, tz ) ;
glRotatef( degrees, ax, ay, az ) ;
glScalef( sx, sy, sz ) ;
gColor3f( r, g, b ) ;
gBegin( GL_TRIANGLES ) ;
gNormal3f( nx0, ny0, nz0 ) ;
gVertex3f( x0, y0, z0 ) ;
gNormal3f( nx1, ny1, nz1 ) ;
gVertex3f( x1, y1, z1 ) ;
gNormal3f( nx2, ny2, nz2 ) ;
gVertex3f( x2, y2, z2 ) ;
gEnd( ) ;
```

Flat Shading (Per-face)

```c

gMatrixMode( GL_MODELVIEW ) ;
gTranslatef( tx, ty, tz ) ;
gRotatef( degrees, ax, ay, az ) ;
gScalef( sx, sy, sz ) ;
gShadeModel( GL_FLAT ) ;
gNormal3f( nx, ny, nz ) ;
gColor3f( r, g, b ) ;
gBegin( GL_TRIANGLES ) ;
gVertex3f( x0, y0, z0 ) ;
gVertex3f( x1, y1, z1 ) ;
gVertex3f( x2, y2, z2 ) ;
gEnd( ) ;
```
Smooth Shading (Per-vertex)

```c
glMatrixMode(GL_MODELVIEW);
glTranslatef(tx, ty, tz);
glRotatef(degrees, ax, ay, az);
glScalef(sx, sy, sz);

// Color

// OpenGL Surface Normals Need to be Unitized by Someone
OpenGL expects the normal vector to be a unit vector, that is: \( nx^2 + ny^2 + nz^2 = 1 \)
If it is not, or if you are using scaling transformations, you can force OpenGL to do the
unitizing for you with:

```c
 glEnable(GL_NORMALIZE);
```

The OpenGL "built-in" Lighting Model

<table>
<thead>
<tr>
<th>1. Ambient = a constant</th>
<th>Accounts for light bouncing &quot;everywhere&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Diffuse = ( I \cos \theta )</td>
<td>Accounts for the angle between the incoming light and the surface normal</td>
</tr>
<tr>
<td>3. Specular = ( I \cos \phi )</td>
<td>Accounts for the angle between the &quot;perfect reflector&quot; and the eye. The exponent, ( \phi ), accounts for surface shininess</td>
</tr>
</tbody>
</table>

Note that \( \cos \theta \) is just the dot product between unit vectors \( L \) and \( n \)
Note that \( \cos \phi \) is just the dot product between unit vectors \( R \) and \( E \)
Diffuse Lighting actually works because of spreading out the same amount of light energy across more surface area.

\[ \text{Diffuse} = I \cos \theta \]

You are all familiar with the Specular Lighting effects. These all have metallic-looking surfaces. What tells you that? It's the shiny-reflection spots.

These are not actually metal. They are wood with special paint that mimics the metallic reflection highlights. We can mimic the same effects digitally!

The Specular Lighting equation is a heuristic that approximates reflection from a rough surface.

\[ \text{Specular} = I \cos^S \phi \]

\[ S = \text{"shininess"} \]
\[ 1/S = \text{"roughness"} \]

The Three Elements of Built-In OpenGL Lighting:

- Ambient
- Diffuse
- Specular

Types of Light Sources:

- Point
- Directional (Parallel, Sun)
- Spotlight
Lighting Examples

Point Light at the Eye

Point Light at the Origin

Spot Lights

Colored Lights Shining on Colored Objects

White Light

Green Light

Too Many Lighting Options
If there is one light and one material, the following things can be set independently:

- Global scene ambient red, green, blue
- Light position: x, y, z
- Light ambient red, green, blue
- Light diffuse red, green, blue
- Light specular red, green, blue
- Material reaction to ambient red, green, blue
- Material reaction to diffuse red, green, blue
- Material reaction to specular red, green, blue
- Material specular shininess

This makes for 25 things that can be set for just one light and one material! While many combinations are possible, some make more sense than others.

Ways to Simplify Too Many Lighting Options

1. Set the ambient light globally using, for example,
   `glLightModelfv( GL_LIGHT_MODEL_AMBIENT, MulArray3( .3f, White ) )`
   I.e., set it to some low intensity of white.
2. Set the light's ambient component to zero.
3. Set the light's diffuse and specular components to the full color of the light.
4. Set each material's ambient and diffuse to the full color of the object.
5. Set each material's specular component to some fraction of white.

```
float White[4] = { 1.,1.,1.,1. };
// utility to create an array from 3 separate values:
float *Array3( float a, float b, float c ){
    static float array[4];
    array[0] = a;
    array[1] = b;
    array[2] = c;
    array[3] = 1.;
    return array;
}
// utility to create an array from a multiplier and an array:
float *MulArray3( float factor, float array0[3] ){
    static float array[4];
    array[0] = factor * array0[0];
    array[1] = factor * array0[1];
    array[2] = factor * array0[2];
    array[3] = 1.;
    return array;
}
The 4th element of the array is being set to 1.0 for no reason. The reason for that is coming up soon.
```
Setting the Material Characteristics

```cpp
void SetMaterial( float r, float g, float b, float shininess ) {
    glMaterialfv( GL_BACK, GL_EMISSION, Array3( 0., 0., 0. ) );
    glMaterialfv( GL_BACK, GL_AMBIENT, MulArray3( .4f, White ) );
    glMaterialfv( GL_BACK, GL_DIFFUSE, MulArray3( 1., White ) );
    glMaterialfv( GL_BACK, GL_SPECULAR, Array3( 0., 0., 0. ) );
    glMaterialf( GL_BACK, GL_SHININESS, 2.f );
    glMaterialfv( GL_FRONT, GL_EMISSION, Array3( 0., 0., 0. ) );
    glMaterialfv( GL_FRONT, GL_AMBIENT, Array3( r, g, b ) );
    glMaterialfv( GL_FRONT, GL_DIFFUSE, Array3( r, g, b ) );
    glMaterialfv( GL_FRONT, GL_SPECULAR, MulArray3( .8f, White ) );
    glMaterialf( GL_FRONT, GL_SHININESS, shininess );
}
```

A Material-setting Shortcut I Like

```cpp
void SetMaterial( float r, float g, float b, float shininess ) {
    glMaterialf( GL_BACK, GL_AMBIENT, MulArray3( .4f, White ) );
    glMaterialf( GL_BACK, GL_DIFFUSE, MulArray3( 1., White ) );
    glMaterialf( GL_BACK, GL_SPECULAR, Array3( 0., 0., 0. ) );
    glMaterialf( GL_BACK, GL_SHININESS, 5. );
    glMaterialf( GL_FRONT, GL_AMBIENT, MulArray3( 1., rgb ) );
    glMaterialf( GL_FRONT, GL_DIFFUSE, MulArray3( 1., rgb ) );
    glMaterialf( GL_FRONT, GL_SPECULAR, MulArray3( .7, White ) );
    glMaterialf( GL_FRONT, GL_SHININESS, 8. );
}
```

Setting the Light Characteristics

```cpp
void SetLight( float r, float g, float b, float shininess ) {
    glLightfv( GL_LIGHT0, GL_AMBIENT,   Array3( 0., 0., 0. ) );
    glLightfv( GL_LIGHT0, GL_DIFFUSE,      LightColor );
    glLightfv( GL_LIGHT0, GL_SPECULAR,  LightColor );
    glLightf( GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1. );
    glLightf( GL_LIGHT0, GL_LINEAR_ATTENUATION, 0. );
    glLightf( GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0. );
    // this is here because we are going to do object (and thus normal) scaling:
    glEnable( GL_NORMALIZE );
}
```

Setting the Light Position

```cpp
void SetLightPosition() {
    // this is true at the moment the glLightfv(...., GL_POSITION,...) function is called.  It is really important to remember this!
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
    // 1. if we do this, then the light will be wrt the scene at XLIGHT, YLIGHT, ZLIGHT:
    glLightfv( GL_LIGHT0, GL_POSITION,  Array3(XLIGHT, YLIGHT, ZLIGHT) );
    // translate the object into the viewing volume:
    gluLookAt(  XEYE, YEYE, ZEYE,  0., 0., 0.,  0., 1., 0. );
    // 2. if we do this, then the light will be wrt the eye at XLIGHT, YLIGHT, ZLIGHT:
    // glLightfv( GL_LIGHT0, GL_POSITION,  Array3(XLIGHT, YLIGHT, ZLIGHT) );
}
```
Computer Graphics

// perform the rotations and scaling about the origin:
glRotatef( Xrot, 1., 0., 0. );
glRotatef( Yrot, 0., 1., 0. );
glScalef( Scale, Scale, Scale );

// 3. if we do this, then the light will be wrt to the object at XLIGHT, YLIGHT, ZLIGHT:
// glLightfv( GL_LIGHT0, GL_POSITION, Array3(XLIGHT, YLIGHT, ZLIGHT) );

// specify the shading model:
glShadeModel( GL_SMOOTH );

// enable lighting:
glEnable( GL_LIGHTING );
glEnable( GL_LIGHT0 );

...
But, for per-fragment, you will need shaders (coming up soon!)

Before, when we talked about normal vectors, we did this:

- glMatrixMode(GL_MODELVIEW);
- glTranslatef(tx, ty, tz);
- glRotatef(degrees, ax, ay, az);
- glScalef(sx, sy, sz);
- glShadeModel(GL_SMOOTH);
- glBegin(GL_TRIANGLES);
  
  - glColor3f(r0, g0, b0);
  - glVertex3f(x0, y0, z0);
  
  - glColor3f(r1, g1, b1);
  - glVertex3f(x1, y1, z1);
  
  - glColor3f(r2, g2, b2);
  - glEnd();

This is especially useful when using colors for scientific visualization:

Smooth Shading can also interpolate vertex colors, not just the results of the lighting model

Before, when we talked about normal vectors, we did this:

- glMatrixMode(GL_MODELVIEW);
- glTranslatef(tx, ty, tz);
- glRotatef(degrees, ax, ay, az);
- glScalef(sx, sy, sz);
- glShadeModel(GL_SMOOTH);
- glBegin(GL_TRIANGLES);
  
  - glColor3f(r0, g0, b0);
  - glVertex3f(x0, y0, z0);
  
  - glColor3f(r1, g1, b1);
  - glVertex3f(x1, y1, z1);
  
  - glColor3f(r2, g2, b2);
  - glEnd();

This is especially useful when using colors for scientific visualization:

Notice how these vertical stripes look "scalloped", like a Greek column. But, they are solid-color stripes. What is going on?
Our vision systems can’t handle abrupt changes in intensity.

In fact, our vision systems can’t handle abrupt changes in the slope of intensity.

Think of the Mach Banding problem as being similar to trying to round second base at a 90º angle.