Computer Graphics Lighting

Why Do We Care About Lighting?

Lighting "dis-ambiguates" 3D scenes

With lighting

Without 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.

Sometimes they are defined or computed per-vertex like this.

Where Do Surface Normals Come From?

When the triangle is approximating an underlying smooth surface that we know the equation of, we can get them by knowing what the exact normal of the smooth surface would have been. A good example is looking at a sphere from the side:

The sphere we are trying to approximate

Assign the underlying sphere's exact normal vectors to the corners of the triangle

A triangle we are using to approximate the sphere with

When the triangle is part of an arbitrary polyhedron for which we do not have an underlying exact equation, we use vector cross products of the edge vectors to get a vector that is perpendicular to the surface:

\[ n = (P1 - P0) \times (P2 - P0) \]

vector cross product
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 );

// Per-face normal is set before the face is drawn

// Example code

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 );

// Per-vertex normal is set while the face is being drawn

// Example code

glColor3f( 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
glMatrixMode( GL_MODELVIEW );
glTranslatef( tx, ty, tz );
glRotatef( degrees, ax, ay, az );
glScalef( sx, sy, sz );

// Per-face normal is set before the face is drawn

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 );

// Per-vertex normal is set while the face is being drawn

gShadeModel( GL_SMOOTH );

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( );
```
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, you can force OpenGL to do the unitizing for you with:

```cpp
glEnable( GL_NORMALIZE );
```

The OpenGL “built-in” Lighting Model

1. **Ambient** = a constant  
   Accounts for light bouncing “everywhere”

2. **Diffuse** = \( I \cos\theta \)  
   Accounts for the angle between the incoming light and the surface normal

3. **Specular** = \( I \cos^s\phi \)  
   Accounts for the angle between the “perfect reflector” and the eye. The exponent, \( s \), accounts for surface shininess

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 \( E \)
You are all familiar with the Diffuse Lighting effects.

Diffuse Lighting actually works because of spreading out the same amount of light energy across more surface area.

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

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 equation that approximates reflection from a rough surface:

\[
\text{Specular} = I \cos^\theta \\
S = \text{"shininess"} \\
1/S = \text{"roughness"}
\]

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
Lighting Examples

Spot Lights

Colored Lights Shining on Colored Objects

What the light can produce

What the material can reflect

\[ E_R = L_R \cdot M_R \]
\[ E_G = L_G \cdot M_G \]
\[ E_B = L_B \cdot M_B \]

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,
   \[ \text{glLightModelf}( \text{GL\_LIGHT\_MODEL\_AMBIENT}, \text{MulArray3}( .3f, \text{WHITE} ) ) \]
   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.
const float WHITE[] = {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 being set to 1.0 is there on purpose. The reason for that is coming up soon!

Setting the Material Characteristics

Characters for the back-facing surfaces
- glMaterialf(GL_BACK, GL_AMBIENT, MulArray3(.4, 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_BACK, GL_EMISSION, Array3(0., 0., 0.));

Characters for the front-facing surfaces
- 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.);
- glMaterialf(GL_FRONT, GL_EMISSION, Array3(0., 0., 0.));

You can also set the front and back characteristics to be the same value at the same time
- glMaterialf(GL_FRONT_AND_BACK, ...);

How Does OpenGL Define GL_FRONT and GL_BACK?
Vertices are CCW from the point of view of the eye
- GL_FRONT
Vertices are CW from the point of view of the eye
- GL_BACK

A Material-setting Helper Function I Like to Use

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.f, WHITE));
    glMaterialf(GL_BACK, GL_SPECULAR, Array3(0.f, 0., 0.));
    glMaterialf(GL_BACK, GL_SHININESS, 2.f);
    glMaterialf(GL_FRONT, GL_AMBIENT, MulArray3(1.f, rgb));
    glMaterialf(GL_FRONT, GL_DIFFUSE, MulArray3(1.f, rgb));
    glMaterialf(GL_FRONT, GL_SPECULAR, MulArray3(7., WHITE));
    glMaterialf(GL_FRONT, GL_SHININESS, shininess);
}

This code is in your sample code folder in the file setmaterial.cpp
Setting the Light Characteristics

```c
# Enable lighting
enable( GL_LIGHTING );

# Enable light 0
enable( GL_LIGHT0 );

# Set lighting model
glLightModeli ( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );

# Set ambient light
mulArray3(.2, WHITE);

# Set lighting model ambient
glLightModelfv( GL_LIGHT_MODEL_AMBIENT, mulArray3(.2, WHITE) );

# Set lighting model two-sided
glLightModeli( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );

# Set light 0 ambient
Array3( 0., 0., 0. );

# Set light 0 diffuse
LightColor;

# Set light 0 specular
LightColor;

# Set light 0 constant attenuation
1.

# Set light 0 linear attenuation
0.

# Set light 0 quadratic attenuation
0.

You can have multiple lights, nominally 0-7.
```

Setting the Light Position

```c
# Set model view matrix
glMatrixMode( GL_MODELVIEW );

# Load identity
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) );

// specify the shading model:

glShadeModel( GL_SMOOTH );

// load the modelview matrix

glMatrixMode( GL_MODELVIEW );

// load the identity

glLoadIdentity();

// perform the rotations and scaling about the origin:

glRotatef( Xrot, 1., 0., 0. );

// translate the object into the viewing volume:

gluLookAt( XEYE, YEYE, ZEYE, 0., 0., 0., 0., 1., 0. );

// 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) );

// enable lighting:

glEnable( GL_LIGHTING );

glEnable( GL_LIGHT0 );

// draw the objects:

. . .
```

Light Attenuation

```c
// this is here because we are going to do object (and thus normal) scaling:

enable( GL_NORMALIZE );

// perform the rotations and scaling about the origin:

glRotatef( Xrot, 1., 0., 0. );

// translate the object into the viewing volume:

gluLookAt( XEYE, YEYE, ZEYE, 0., 0., 0., 0., 1., 0. );

// specify the shading model:

glShadeModel( GL_SMOOTH );

// enable lighting:

glEnable( GL_LIGHTING );

You can enable and disable lighting “at all”.
(This toggles between using what the lighting equations say and what glColor3f( ) says.)

You can enable and disable each light independently

You can enable and disable each light independently

It is usually good form to disable the lighting after you are done using it

Physics tells us that light energy decreases with the inverse square of the distance,

\[ \frac{1}{d^2} \]

where \( d \) is the distance from the light to the point being lit.

Often, we don’t want any attenuation, that is, we want to see everything. In that case, set

\[ C=1., L=0., Q=0. \]

And sometimes you might want to attenuate linearly. Why? Well, because you can!

In that case, set

\[ C=0., L=1., Q=0. \]
**Sidebar: Why are Light Positions 4-element arrays where the 4th element is 1.0? Homogeneous Coordinates!**

```c
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;
}
```

We usually think of a 3D point as being represented by a triple: (x,y,z).

Using homogeneous coordinates, we add a 4th number: (x,y,z,w).

Graphics systems take (x,y,z,w), perform all transformations, and then divide x, y, and z by w before using them.

\[ X = \frac{x}{w}, \quad Y = \frac{y}{w}, \quad Z = \frac{z}{w} \]

Thus (1,2,3,1), (2,4,6,2), (-1,-2,-3,-1) all represent the same 3D point.

---

**Homogeneous Coordinates let us Represent Points at Infinity**

This is useful to be able to specify a parallel light source by placing the light source position at infinity.

- The point (1,2,3,1) represents the 3D point (1,2,3)
- The point (1,2,3,5) represents the 3D point (2,4,6)
- The point (1,2,3,01) represents the point (100,200,300)

So, (1,2,3,0) represents a point at infinity, along the ray from the origin through (1,2,3).

Points-at-infinity are used for parallel light sources (and some shadow algorithms).

---

**Additional Parameters for Spotlights**

- `glLightf( GL_LIGHT0, GL_SPOT_DIRECTION, Array3(xdir,ydir,zdir) );`
  Specifies the spotlight-pointing direction. This gets transformed by the current value of the ModelView matrix.

- `glLightf( GL_LIGHT0, GL_SPOT_EXPONENT, e );`
  Specifies the spotlight directional intensity. This acts very much like the exponent in the specular lighting equation.

- `glLightf( GL_LIGHT0, GL_SPOT_CUTOFF, deg );`
  Specifies the spotlight maximum spread angle.

---

**Two Light-setting Helper Functions I Like to Use**

```c
void SetPointLight( int ilight, float x, float y, float z, float r, float g, float b )
{
    glLightfv( ilight, GL_POSITION,  Array3(x, y, z) );
    glLightfv( ilight, GL_AMBIENT,   Array3(0., 0., 0.) );
    glLightfv( ilight, GL_DIFFUSE,   Array3(r, g, b) );
    glLightfv( ilight, GL_SPECULAR,  Array3(r, g, b) );
    glLightf ( ilight, GL_CONSTANT_ATTENUATION, 1. );
    glLightf ( ilight, GL_LINEAR_ATTENUATION, 0. );
    glLightf ( ilight, GL_QUADRATIC_ATTENUATION, 0. );
    glEnable( ilight );
}
void SetSpotLight( int ilight, float x, float y, float z, float xdir, float ydir, float zdir, float r, float g, float b )
{
    glLightfv( ilight, GL_POSITION,  Array3(x, y, z) );
    glLightfv( ilight, GL_SPOT_DIRECTION, Array3(xdir, ydir, zdir) );
    glLightf( ilight, GL_SPOT_EXPONENT, 1. );
    glLightf( ilight, GL_SPOT_CUTOFF, 45. );
    glLightfv( ilight, GL_AMBIENT,   Array3(0., 0., 0.) );
    glLightfv( ilight, GL_DIffUSE,   Array3(r, g, b) );
    glLightfv( ilight, GL_SPECULAR,  Array3(r, g, b) );
    glLightf ( ilight, GL_CONSTANT_ATTENUATION, 1. );
    glLightf ( ilight, GL_LINEAR_ATTENUATION, 0. );
    glLightf ( ilight, GL_QUADRATIC_ATTENUATION, 0. );
    glEnable( light );
}
```
Sidebar: Note that we are computing the light intensity at each vertex first, and then interpolating that intensity across the polygon second. That is, you are only using the lighting model at each vertex. You can do an even better job if you interpolate the normal across the polygon first, and then compute the light intensity with the lighting model at each fragment second:

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

Per-vertex

Per-fragment

Before, when we talked about per-vertex normal vectors, we did this:

We can also provide per-vertex colors to do this:

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
**Tricky Lighting Situations**

Watch for these in movies!

- Hair
- Fur
- Feathers

**Sidebar: Beware of Mach Banding**

Notice how these vertical stripes look "scalloped", like a Greek column. But, they are solid-color stripes. What is going on?

**Beware of Mach Banding**

Our vision systems can't handle abrupt changes in intensity.

- Actual Intensity Changes
- Our Perceived Intensity Changes
In fact, our vision systems can't even handle abrupt changes in the slope of intensity.

Flat shading          Smooth shading

This "white line" doesn't really exist—it is an artifact of our vision system!

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