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:

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:

\[ n = (P1 - P0) \times (P2 - P0) \]
### Setting a Per-Face Surface Normal in OpenGL

```c
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();

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

### Setting Per-Vertex Surface Normals in OpenGL

```c
Per-vertex normal is set while the face is being drawn
```

### Flat Shading (Per-face)

```c
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();

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

### Smooth Shading (Per-vertex)

```c
Per-vertex normal is set while the face is being drawn
```
OpenGL Surface Normals Need to be Unitized by Someone

```c
glTranslatef(tx, ty, tz);
glRotatef(degrees, ax, ay, az);
glScalef(sx, sy, sz);
glNormal3f(nx, ny, nz);
```

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:

```c
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

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

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.

You are all familiar with the Specular Lighting effects

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:

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

- \( S \) = "shininess"
- \( 1/S \) = "roughness"

The Three Elements of Built-in OpenGL Lighting:

- Ambient
- Diffuse
- Specular

The Three Elements of Built-in OpenGL Lighting:

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**

- **White Light**
- **Green Light**

\[
\begin{align*}
E_R &= L_R \ast M_R \\
E_G &= L_G \ast M_G \\
E_B &= L_B \ast M_B
\end{align*}
\]

### 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,
   \[\text{glLightModelfv}( \text{GL\_LIGHT\_MODEL\_AMBIENT}, \text{MulArray3}( .3f, \text{WHITE} ) )\]
   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 4\textsuperscript{th} 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

Characteristics for the back-facing surfaces

Characteristics for the front-facing surfaces

You can also set the front and back characteristics to be the same value at the same time

How Does OpenGL Define GL_FRONT and GL_BACK?

Vertices are CCW from the point of view of the eye

A Material-setting Helper Function I Like to Use

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

```c
// Enable lighting
glEnable( GL_LIGHTING );
glEnable( GL_LIGHT0 );

// ModelView lighting
glLightModefv( GL_LIGHT_MODEL_AMBIENT, MulArray3( .2, WHITE ) );
glLightModeli (  GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );

// Light0 configuration
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. );
```

You can have multiple lights, nominally 0-7. The light characteristics are set as shown above. The light position gets transformed by the ModelView matrix at the moment the glLightfv(..., GL_POSITION, ...) function is encountered. It is really important to remember this!

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

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

```
// enable lighting:
gEnable( GL_LIGHTING );
```

You can enable and disable each light independently.

```
// draw the objects:
gDisable( GL_LIGHTING );
```

```
// Setting the Light Position

```c
// 1. if we do this, then the light will be wrt the scene at XLIGHT, YLIGHT, ZLIGHT:
gLightfv( GL_LIGHT0, GL_POSITION,  Array3(XLIGHT, YLIGHT, ZLIGHT) );
```

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

```
// perform the rotations and scaling about the origin:
gRotatef( Xrot, 1., 0., 0. );
gRotatef( Yrot, 0., 1., 0. );
gScalef( Scale, Scale, Scale );
```

```
// 2. if we do this, then the light will be wrt the eye at XLIGHT, YLIGHT, ZLIGHT:
if ( 3. if we do this, then the light will be wrt to the object at XLIGHT, YLIGHT, ZLIGHT:
```

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

Light Attenuation

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

Attenuation = \( \frac{1}{C + Ld + Qd^2} \) where \( d \) is the distance from the light to the point being lit. Physics tells us that light energy decreases with the inverse square of the distance, \( \frac{1}{d^2} \). To emulate this, we would set \( C=0., L=0., Q=1. \). Streetlights and car headlights are good uses for this.

Often, we don’t want any attenuation, that is, we want to see everything. In that case, set \( C=1., L=0., Q=0. \).

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

```
// you can have multiple lights, nominally 0-7:
```

```
// if we do this, then the light will be wrt the eye at XLIGHT, YLIGHT, ZLIGHT:
```c
if ( 3. if we do this, then the light will be wrt to the object at XLIGHT, YLIGHT, ZLIGHT:
```

// specify the shading model:
```c
```

And sometimes you might want to attenuate linearly. Why? Well, because you can! In that case, set \( C=0., L=1., Q=0. \).
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;
}

Sidebar: Why are Light Positions 4-element arrays where the 4th element is 1.0? Homogeneous Coordinates!
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.
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 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. A cutoff angle of 180° indicates that this is really a point light.

Two Light-setting Helper Functions I Like to Use

void SetPointLight( int ilight, float x, float y, float z,  float r, float g, float b )
{
    glLightfv( ilight, GL_POSITION,  Array3( x, y, z ) );
    glLightf( ilight, GL_SPOT_CUTOFF, 180.f );
    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.f );
    glLightf ( ilight, GL_LINEAR_ATTENUATION, 0.f );
    glLightf ( ilight, GL_QUADRATIC_ATTENUATION, 0.f );
    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.f );
    glLightfv( ilight, GL_SPOT_CUTOFF, 45.f );
    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.f );
    glLightf ( ilight, GL_LINEAR_ATTENUATION, 0.f );
    glLightf ( ilight, GL_QUADRATIC_ATTENUATION, 0.f );
    glEnable( ilight );
}
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!)

Sidebar: Smooth Shading can also interpolate vertex colors, not just the results of the lighting model. Before, when we talked about per-vertex normal vectors, we did this:

```gl
    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( );
```

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

```gl
    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 );
    glVertex3f( r2, g2, b2 );
    glEnd( );
```

This is especially useful when using colors for scientific visualization:
Tricky Lighting Situations

Watch for these in movies!

Hair
Fur
Feathers

Notice the lighting in the fur!

Disney
Sony/Columbia Pictures

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?

Our vision systems can't handle abrupt changes in intensity.
Beware of Mach Banding

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.

Perceived Intensity

Actual Intensity