

COMP3201 – Computer Graphics

3. Module 3: Realism and Performance

3.3 Introduction to Lighting and Materials

3.3.1 Introduction

Lighting is an important part of a scene. Many objects look different under different lighting conditions. In fact, some objects don't even look three dimensional until lighting is applied.

Lighting in the real world relies on photons from one or more light sources. At a surface some of these photons are reflected and others are absorbed. Precisely which photons are absorbed or reflected, and the angles at which the incident light is reflected, depends on the material properties of the surface. Some of the reflected photons then reach our eyes. Our eyes possess three different types of cones (photon receivers). Each type of cone is specialised to detecting a certain range of photon energies. Thus our eyes can actually only see three colours: red, green, and blue. All the colours that our brain thinks it sees are made up of differing intensity combinations of those three colours. This allows us to see colours that aren't in the electromagnetic spectrum!

To model the way the real world does lighting is extremely complex and computationally expensive. Current computers can't do real world lighting in real time. OpenGL doesn't try to model the real world exactly but implements a lighting model that approximates the real world to a large degree and is feasible to compute in real time.

OpenGL's version of lighting has light, from a light source, introduced to the scene, propagating until it reaches a vertex. The light interacts with the material properties of the vertex and is reflected to a viewer. Furthermore, OpenGL breaks light into ambient, diffuse, and specular components, and breaks materials into ambient, diffuse, specular, and emissive components. These components do not interact. Note that lighting calculations only occur at vertices. Thus a large polygon can look unrealistic.

However many smaller polygons adds to the computational time required.

3.3.2 Adding Lighting to a Scene

There are five steps to adding lighting to a scene:

1. Turn the lighting calculations on
 - Simply call `glEnable(GL_LIGHTING)`.
2. Define normal vectors for every vertex in every object

- This is achieved with a call to **glNormal*()** before a **glVertex*()**.
3. Define material properties for every vertex
 - This is achieved with calls to **glMaterial*()**.
 4. Create and position the required light sources
 - Light *n* is turned on with **glEnable(GL_LIGHT*n*);** light properties are then set with **glLight*()**.
 5. Define the lighting model (global ambient light and effective location of the viewpoint for the lighting calculation)
 - This is set up with **glLightModel*()**.

Note that lighting and materials are intertwined. If you don't define the material properties for a vertex, there is nothing for the light to interact with. Stated another way, a light source simply injects light into the scene without being part of the scene. That is, if you point a light directly at the camera there is no visual effect. *Light only exists to interact with materials.*

The purpose of the following program is to demonstrate the simplest lighting example possible and show that an object can look either 2 or 3 dimensional depending on the lighting. Specifically it draws a torus on the screen and toggles lighting with the 'l' or 'L' keys.

```

/* light1.c */
#include <GL/glut.h> /* includes gl.h */
#define KEY_ESC 27 /* glut doesn't define this one for some reason */

GLvoid display( GLvoid )
{
    glClear( GL_COLOR_BUFFER_BIT );

    /* Note that glut defines the normals for its 3D objects */
    glutSolidTorus( 0.3, 0.6, 50, 50 );
    glFlush();
}

GLvoid keyboard( GLubyte key, GLint x, GLint y)
{
    static GLboolean lightOn = GL_FALSE;
    switch (key)
    {
        case KEY_ESC: /* exit when escape key is pressed */
            exit(0);
        break;

        case 'l':
        case 'L':
            /* Toggle the light on/off */
            lightOn = !lightOn;
    }
}

```

```

        /* Turn on/off the OpenGL lighting calculations */
        lightOn ? glEnable( GL_LIGHTING ) : glDisable( GL_LIGHTING );
        glutPostRedisplay();
    }
    break;
}

void init( void )
{
    /* What colour is the ambient light */
    GLfloat pAmbientLightModel[] = { 0.3, 0.3, 0.3, 1.0 };

    /* Where is the infinite directional light */
    GLfloat pLight0Pos[] = { 10.0, 10.0, 2.0, 0.0 };

    /* What are the ambient, diffuse, and specular material
    properties for our torus
    */
    GLfloat pMatAmbDiff[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat pMatSpecular[] = { 1.0, 1.0, 1.0, 1.0 };

    /* Get some ambient light */
    glLightModelfv( GL_LIGHT_MODEL_AMBIENT, pAmbientLightModel );

    /* Get an infinite, but directional light (like a sun) */
    glEnable( GL_LIGHT0 );
    glLightfv( GL_LIGHT0, GL_POSITION, pLight0Pos );

    /* Set up a material for our torus */
    glMaterialfv( GL_FRONT, GL_SPECULAR, pMatSpecular );
    glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, pMatAmbDiff );
}

int main( int argc, char *argv[] )
{
    glutInit( &argc, argv );
    glutCreateWindow( argv[0] );
    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );
    init();

    glutMainLoop();
    return 0;
}

```

For the lighting calculation, light and material properties are split into ambient, diffuse, and specular components. Materials can further have an emissive component.

Each of these components is independent. It is important to note that OpenGL approximates each of the light and material properties as if they can be decomposed into red, green, and blue channels. Each channel is then assumed to be independent.

3.3.3 Definitions: Ambient, Diffuse, Specular Light

Ambient Light

This is light whose rays have been scattered so much by the environment that it is now omnidirectional. When ambient light interacts with a surface it is assumed to scatter equally in all directions. This means that no matter where the eye is located, the vertex appears equally bright.

Because ambient light needs no particular direction, OpenGL has the concept of global ambient light as well as the concept of ambient light injected into a scene by a particular light.

Specular Light

Specular light is directional light that behaves as a torch would. The light is directional so a head-on beam gives brighter illumination than a glancing beam. The brightness is also dependent on the eye's position. This light gives objects the "shiny spot".

Diffuse Light

This light behaves like incoming specular light and outgoing ambient. The incoming light is directional so a head-on beam gives brighter illumination than a glancing beam, but once it interacts with the surface, the light is scattered in all directions equally. This means that no matter where the eye is located the vertex appears equally bright.

Specular and diffuse light need a direction and thus are always associated with a particular light source (even if the light is at infinity).

3.3.4 Definitions: Ambient, Diffuse, Specular, Emissive Material Properties

Ambient Material Properties

This defines the overall colour of the object. This colour is most noticeable when the diffuse light is low, i.e., when the object is receiving no direct illumination.

Diffuse Material Properties

This defines the colour of the object under direct illumination, and usually is what we perceive the colour of the object to be.

It should be noted that the reflected light in both ambient and diffuse cases is independent of viewing position.

For almost all cases the ambient and diffuse reflectance (material properties) are the same. Because of this there is a way to set both simultaneously in OpenGL.

```
GLfloat pMatAndDiff[] = { rval, gval, bval, alphaval };  
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, pMatAndDiff );
```

The ambient and diffuse material properties can be set separately by specifying `GL_AMBIENT`, and `GL_DIFFUSE`, in the `glMaterial*()` call.

Specular Material Properties

This is the colour of the highlights that an object gets under specular light. The amount of specular reflection does depend on the viewing angle. As well as the colour of the specular material, OpenGL allows control of the effect that the material has on the specular light, i.e., the size and brightness of the highlight are controllable. The shininess is a scalar value between 0.0 and 128.0 with the higher values representing the smaller, brighter, more focussed, highlight.

These are set by

```
GLfloat pMatSpec[] = { rval, gval, bval, alphaval };  
GLfloat pMatShininess[] = { shinyval };  
glMaterialfv( GL_FRONT, GL_SPECULAR, pMatSpec );  
glMaterialfv( GL_FRONT, GL_SHININESS, pMatShininess );
```

Emissive Material Properties

This is the colour of the light emitted by an object (e.g. headlights, torches, etc). This emitted light (from the material) is not affected by the incoming light. It should also be noted that an emissive material does not inject light into the rest of the scene. Thus a complete model of a headlight needs both a light and an emissive material allocated to give the desired effect.

Emissive properties are set by

```
GLfloat pMatEmissive = { rval, gval, bval, alphaval };  
glMaterialfv( GL_FRONT_AND_BACK, GL_EMISSION, pMatEmissive );
```

The alpha value of a vertex is that of the diffuse material alpha value, no matter what the alpha values supplied for the other material properties are.

The colour produced by lighting a vertex is computed by the following:

```
vertex colour = material emission at that vertex  
               + the global ambient light scaled by the material's ambient  
                 property at that vertex  
               + the ambient, diffuse, and specular contributions from all the  
                 light sources, properly attenuated
```

We reiterate that when lighting is enabled in OpenGL, the colour that a vertex was assigned becomes meaningless as the lighting calculations only depend on the material properties of a vertex, the vertex normal, and the incident light.

3.3.5 Light Source Properties

All the properties associated with a light source are set via the **glLight*** set of commands. The command declarations look like

```
void glLight{if} ( GLenum light, GLenum paramname, TYPE param );  
void glLight{if}v( GLenum light, GLenum paramname, TYPE* param );
```

The **i** version of the command means that TYPE will be a **GLint**, the **f** version of the command means that TYPE will be a **GLfloat**. **v** is the usual vector version.

OpenGL specifies that there must be at least 8 lights available for use. We specify which light this property is setting via the enumerated constant **light**. **light** is one of {**GL_LIGHT0**, ..., **GL_LIGHT7**}.

There are 10 different properties (paramname argument) that a light has. The following 5 properties are set with the vector version of **glLight***

Table 1: Light properties set with glLight*v()

GL_AMBIENT	ambient colour of the light, default = {0.0, 0.0, 0.0, 1.0}
GL_DIFFUSE	diffuse colour of the light, default = white for light0, black for light <i>n</i>
GL_SPECULAR	specular colour of the light, default = white for light0, black for light <i>n</i>
GL_POSITION	position of the light, default = {0.0, 0.0, 1.0, 0.0}, note use w=0.0 for a light at infinity
GL_SPOT_DIRECTION	direction of the spotlight, default = {0.0, 0.0, -1.0}

and these properties use the scalar version of **glLight***

Table 2: Light properties set with glLight*

GL_SPOT_EXPONENT	the intensity variation of the light in the light cone, default = 0.0
GL_SPOT_CUTOFF	half of the spotlight cone angle, default = 180.0
GL_CONSTANT_ATTENUATION	constant attenuation factor, default = 1.0
GL_LINEAR_ATTENUATION	linear attenuation factor, default = 0.0
GL_QUADRATIC_ATTENUATION	quadratic attenuation factor, default = 0.0

Notes on light colour:

Usually the specular component of the light is the same as the diffuse component.

Note that the alpha component of the light is not used unless blending is enabled.

Example

```
GLfloat red[] = {1.0, 0.0, 0.0, 1.0};
```

```
glLightfv( GL_LIGHT2, GL_DIFFUSE, red );  
glLightfv( GL_LIGHT2, GL_SPECULAR, red );
```

Notes on position:

A light source can be considered to be infinitely far away or local. The infinite case is referred to as a directional light, the finite case is a positional light. The rays are considered to be parallel at an object in the infinite case while the exact angle is taken into account in the finite case. The sun would be modelled as a directional light while a room light would be modelled as a positional light.

Example

```
GLfloat pos1[] = {1.0, 1.0, 1.0, 0.0}; /* w=0.0 puts it at infinity */  
GLfloat pos2[] = {-2.0, 0.0, 0.0, 1.0};
```

```
glLightfv( GL_LIGHT2, GL_POSITION, pos1 );  
glLightfv( GL_LIGHT0, GL_POSITION, pos2 );
```

Notes on attenuation:

Attenuation is disabled for directional lights as it makes no sense for an infinitely far light to attenuate its intensity. The attenuation factor that OpenGL uses on a positional light source is calculated by

$$\text{attenuation factor} = \frac{1}{k_c + k_l + k_q d^2}$$

where d is the distance from the light source to the current vertex, k_c is the `GL_CONSTANT_ATTENUATION`, k_l is the `GL_LINEAR_ATTENUATION`, and k_q is the `GL_QUADRATIC_ATTENUATION`. The defaults for these parameters are found in Table 2.

Example

```
glLightf( GL_LIGHT0, GL_LINEAR_ATTENUATION, 2.0 );
```

Notes on spotlight effects:

By default a light source emits light in all directions. It is possible to restrict the range of angles that the light is emitted in. As for the attenuation, it only makes sense to have a positional light as a spotlight. The angle that is specified as the spotlight cut-off angle is *half* of the angle of the cone of light. The maximum value that this angle can be specified to be is 90 degrees (except for the special case of 180 degrees to return to the non-spotlight case).

It is also possible to set the direction that the spotlight is pointing towards.

The intensity of the light throughout the spotlight's cone of light can be controlled with the `GL_SPOT_EXPONENT`. The intensity of the light is highest in the centre and decreases towards the edge by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lit, raised to the power of the exponent. Note that the default value of the exponent is zero and so the angle raised to the exponent is always one in the default case (resulting in a uniform light distribution).

Example

```
GLfloat dir[] = {2.0, -1.0, -4.0};
```

```
glLightf( GL_LIGHT3, GL_SPOT_CUTOFF, 36.0 );  
glLightfv( GL_LIGHT3, GL_SPOT_DIRECTION, dir );  
glLightf( GL_LIGHT3, GL_SPOT_EXPONENT, 0.2 );
```

3.3.6 Material Properties

The default material properties are

```
GL_AMBIENT      {0.2, 0.2, 0.2, 1.0}  
GL_DIFFUSE      {0.8, 0.8, 0.8, 1.0}  
GL_SPECULAR     {0.0, 0.0, 0.0, 1.0}  
GL_SHININESS    0.0  
GL_EMISSION     {0.0, 0.0, 0.0, 1.0}
```

The following are some useful material parameters

Polished Copper

```
Ambient  0.229500, 0.088250, 0.027500, 1.000000  
Diffuse  0.550800, 0.211800, 0.066000, 1.000000  
Specular 0.580594, 0.223257, 0.069570, 1.000000  
Shininess 51.200001
```

Jade

```
Ambient  0.135000, 0.222500, 0.157500, 0.950000  
Diffuse  0.540000, 0.890000, 0.630000, 0.950000  
Specular 0.316228, 0.316228, 0.316228, 0.950000  
Shininess 12.800000
```

More material parameters can be found via **google!**

The last note on materials is about performance. There is a command (`glColorMaterial`) to minimise the cost of changing materials. This command causes the material property to track the value of the current colour. Before this command can be used it must be enabled. It should also be disabled as soon as it is not needed as this command still exacts a performance penalty. The command declaration looks like

void glColorMaterial(GLenum face, GLenum mode);

where face is either GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK, and mode is one of { GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, or GL_AMBIENT_AND_DIFFUSE }.

Example

```
glEnable( GL_COLOR_MATERIAL );
glColorMaterial( GL_FRONT, GL_AMBIENT_AND_DIFFUSE );
/* glColor now changes ambient and diffuse material properties as well as colour */
glColor3f( 0.2, 0.3, 0.4, 0.8 );
/* drawing of objects goes here */
glDisable( GL_COLOR_MATERIAL );
```

3.3.7 Lighting Model

Let us now examine Step 5 of Section 3.3.2, the lighting model. The lighting model encapsulates four items. These items are set with **glLightModel*()**.

1. the global ambient light (this is the light that is independent of any particular light source) (default = { 0.2, 0.2, 0.2, 1.0 }),

Example

```
GLfloat pAmbientLightModel[] = { 0.3, 0.3, 0.3, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, pAmbientLightModel );
```

2. whether or not the viewer of the scene should be considered to be local or infinitely removed (local viewers involve extra angle calculations and hence a performance penalty) (default = GL_FALSE, i.e., infinitely removed),

Example

```
glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
```

3. whether or not the lighting calculations should do both the front and back faces of the polygons or only the front faces (often only front faces are visible so reversing the normals to calculate the back faces correctly imposes an unnecessary performance penalty) (default = GL_FALSE, i.e., front face only),

Example

```
glLightModeli( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );
```

4. whether or not the specular color should be separated from ambient and diffuse colors and applied after the texturing operation (default = GL_SINGLE_COLOR, i.e., not separated)

Example

```
glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR );
```

3.3.8 Normals

A normal vector usually defines the direction that is perpendicular to a surface. In OpenGL a normal can only be specified for a vertex. Hence flat polygons will (usually) share the same normals at the vertices but a curved polygon will (usually) have different normals at the vertices.

The calculation procedure for finding normals has been covered in an earlier lecture.

Normals are set via

```
void glNormal3 {bsidf} ( TYPE nx, TYPE ny, TYPE nz );  
void glNormal3 {bsidf}v( TYPE* pvec );
```

Remember that OpenGL is a state machine so the normal that a vertex is assigned is the last state that the normal was set to via `glNormal3*()`. This means that if you need to specify the same normal to many vertices, then the normal need only be specified once.

Example

```
GLfloat n0[] = { 1.0, 0.0, 0.0 };  
GLfloat n1[] = { 0.0, 1.0, 0.0 };  
GLfloat n2[] = { 0.0, 0.0, 1.0 };
```

```
GLfloat v0[] = { 1.0, 0.0, 0.0 };  
GLfloat v1[] = { 0.0, 1.0, 0.0 };  
GLfloat v2[] = { 0.0, 0.0, 1.0 };
```

```
glBegin( GL_POLYGON );  
{  
    glNormal3fv( n0 );  
    glVertex3fv( v0 );  
    glNormal3fv( n1 );  
    glVertex3fv( v1 );  
    glNormal3fv( n2 );  
    glVertex3fv( v2 );  
}  
glEnd();
```

In OpenGL the normals are only used for lighting calculations. Thus if lighting is not required for an application then there is no need to define a normal. It is important to note that since normals are only used for lighting calculations they do not have to behave as a mathematical normal would. That is, sometimes a desirable lighting effect can be achieved by having normals that aren't actually perpendicular to the surface.

For example consider modelling a sphere in OpenGL. A sphere is constructed using many flat polygons. For a particular flat polygon, what do you assign for the normal at each vertex? If one uses the face normal, the angles of the polygons show up under

lighting. However if you instead assign the normal to be the normal that the sphere you are approximating would have, then under lighting the object looks much more like a sphere.

The normals supplied for the lighting calculations are assumed to be unit normals. Unit normal vectors will remain unit length as long as the only transformations are rotations and translations. Any scaling or shearing will alter the length of the normal vector. If non-unit normals are supplied, or length altering transformations occur, it is possible to have OpenGL automatically normalise the vectors by calling `glEnable(GL_NORMALIZE)`;

Performance tip: If the only length-altering transformations are uniform scalings (i.e., the same scaling value for x, y and z) then the command `glEnable(GL_RESCALE_NORMAL)`; can be used instead of `glEnable(GL_NORMALIZE)`; . This allows OpenGL to recalculate the normals without having to recalculate the length of the normal vector.

3.3.9 Complete Lighting Calculation

The complete lighting calculation is presented on pages 214-5 of the red book.