The OpenGL Light Bible

Taken from www.falloutsoftware.com/tutorials/gl

Let there be light... and light there was!

(*A considerably long and educative process notwithstanding*)

This is one of the most substantial as well as educational tutorials on OpenGL lighting model on this web-site and perhaps, on the Internet as a whole. Therefore the name: The OpenGL Light Bible. I would recommend printing out this tutorial.

Light is important and so you will be able to understand the mechanics of it after reading this tutorial. In the beginning I explain what light is and what types of light are there to be aware of. Toward the end, a more technical, OpenGL-specific text is presented. The complete source code for this tutorial of basic light technicalities is included which demonstrates a simple, flat, non-textured 3D model affected by a light source. For this purpose, previously explained [*.m] file model-loading code is utilized.

So let's begin.

Don't forget that you are a student of Fallout School and today's tutorial is a very important one indeed, perhaps the kind of which you've never been pleased to stumble upon before. It starts with a question: What good is it to use OpenGL light-defining function such as:

```
GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
```

...and declare to yourself: "Ah-ha!, this will create a shiny spot on my 3D object" (*and this alone won't, actually*)

?
Perhaps a very good insight… if your brain is an inanimate piece of matter. However, it is not... so we will use it to its fullest capability with an ambition to infiltrate and break apart the reason behind one of the most important concepts existing in visual representation of objects and matter itself - the light. Let's expand our understanding of how light works in nature and apply it to 3D programming with OpenGL with available API calls.

1. The nature of light

INTRODUCTION
If you want to know and understand what light represents in nature, you will read this section. If you want to program light in OpenGL you will read the following one. But if you want to be an educated human being and programmer, I'm terribly sorry but I think you're going to have to read both.

The main idea behind this and other tutorials on this web-site, www.falloutsoftware.com is to educate the reader. We are not just mindlessly going over the code which is the idea behind the many programming tutorials posted online. To become good at something such as programming it's not enough to just know certain functions and know when to call them. So, just as most common terms behind 3D were explained in the initial OpenGL tutorial - gl0 - Introduction to 3D, this part of tutorial works as an introduction to light. However, it wouldn't be convenient to describe everything about light that is known in a computer programming based document, so I will only emphasize on the most important ideas that will help us understand light and light-programming better.

WHAT LIGHT IS

Light is the most important idea behind visual representation of anything that a human being can visually perceive. The idea of perception of light lies in the fact that what you can see isn't based on the objects that you are viewing but on the rays of light cast from a light source and reflected from those objects. It's important to note that your eyes don't directly see objects as there is no physical correlation between your eyes and those objects.

All of this is theoretical, of course. We use the term light rays to merely abstract a more complex mechanism.

So the light rays commonly originate from an energy source such as the sun or a lamp in your room. It's important to note that theoretically a ray of light travels in a straight line and by the time you visually perceive an object, it is the rays of light reflected or scattered off that object that your eyes absorb. From now on you will have to think of light in terms of theoretical rays as described in this tutorial and not as just "something invisible that makes objects brighter". This becomes more important when you start programming your own 3D light graphics engine, as opposed to using the predefined set of OpenGL light-modeling functions.

So from this, the two rules to understand are:

1. Your eyes are mechanisms created to directly perceive or absorb the *photons*(more about photons in a second) of light and not the objects. And you are, as a programmer, destined to simulate this functionality on the computer screen.
2. A ray of light travels in a straight line.

The second point however, is not entirely correct, but *can* be correct at the same time -- let's discover the theory of light a bit further to understand what is meant by this statement. A side note on what is known about the light by scientists will help…

LIGHT AS BOTH: A STREAM OF PARTICLES AND A WAVE OF ENERGY

There are two ways that light could be thought of as. There is a theory of light particles described by PHOTONS. And there is a theory of light being a WAVE of energy. In ancient Greece the light was thought of as a stream of particles which travel in a straight line and bounce off a wall in a way that any other physical objects do. The fact that light couldn't be seen was based on the idea that the light particles are too small for the eye to see, traveled too fast for the eye to notice them or that the eye was just seeing through them.

In the late 1600s it was proposed that the light was actually a wave of energy and didn't travel exactly in a straight line being a stream of particles. By 1807 the theory of light waves was confirmed with an experiment that demonstrated that the light, passed through a narrow slit radiates additional light outward on the other side of the slit. So it was proven that the light has to travel in a form of a wave in order to spread itself that way, and not in a straight line. It is important to note that a beam of light radiates outward at all times.

The theory of light was developed further by Albert Einstein in 1905. He described the "photoelectric effect". This theory described activity of the ultraviolet light hitting a surface, emitting electrons off that surface. This behavior was supported by an explanation that light was made up of a stream of energy packets called PHOTONS.

To conclude, it is observed that the light can behave as both: a wave and also as packets of energy particles: PHOTONS. There is no solid explanation of the more complex underlying structure of how light works, as of yet.

COLOR OF LIGHT

In this section I describe what light is in more detail. Specifically, how color is being perceived by the viewer. As discovered in the previous section light is a wave of energy. The most apparent quality of light lies in representing color. Color is tightly connected to the concept of light. Let's see how.

The light that can be seen by the human eye is in general a mixture of all kinds of different lights scattered and reflected against the surroundings with different material properties. All physical matter is made up of atoms. The mechanics of reflection of photons off physical matter depends on various things such as the kind of atoms, the amount of each kind and the arrangement of atoms in the object that the photons are being reflected off. Some photons are reflected and some are absorbed. When photons are absorbed they are usually converted to heat. The defining factors of the visual quality of a material lie within this matter absorption-and-reflection concept. The color that the material reflects is observed as that material's color. Also, the more light the material reflects the more shiny it will appear to the viewer.

Each different color is simply energy which can be represented by a wavelength. What's important to note is that color is only a wavelength visible to the eye. A wavelength is measured by the distance between the peaks of the energy wave. Consider this image.

The visible light is contained within the wavelengths ranging from 390 nanometers to 720 nanometers in length. At 390 nanometers the color is violet. A wavelength of 720 nanometers represents the red color. Everything in between is considered the visible light and the range itself is called the spectrum:

The segment of the wavelengths between 390nm and 720nm is called the color spectrum. The wavelengths outside the color spectrum are not visible by the eye. Below 390nm (the wavelength of the violet color) the ultraviolet wavelengths are located. Above 720nm infrared wavelengths can be found. The prefix "ultra" means: beyond, on the other side of; the prefix "infra" stands for inferior to or beneath. ...hence the names.

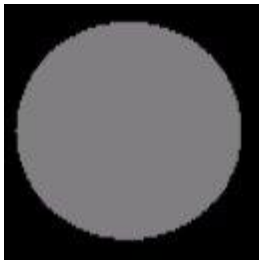
As far as light programming goes, we are only concerned with understanding values within the color spectrum range unless we're programming some sort of an ultraviolet or infra red color wavelength simulation (which in fact we are not).

In graphics programming, and perhaps other crafts which deal with representation of light, a few abstract terms which describe specific effects that light can produce on a surface of an object have emerged. These abstract types of light are described in the following section. Their terminology is crucial to a graphics programmer. Keep in mind that this is by far not the complete set of effects that light in general can produce and serves as a mere approximation.

ABSTRACT TYPES OF LIGHT

The following terms describe different types of light that you must know when programming a 3D application which requires a light source. It is important to understand what effect each of these types of light create on the surface of rendered 3D objects. These terms were created because certain effects that light produces on the objects needed to be described in order to distill the complex mathematical calculations of light. However, this doesn't mean that these exact types of light actually exist in nature, we just think of them as an abstraction of the effects that light can produce when cast on different materials. It would be very time consuming to calculate the real mechanics of light and the way it works in nature so, this common set of light types was generally adopted by OpenGL: AMBIENT LIGHT, DIFFUSE LIGHT and SPECULAR LIGHT. EMISSIVE LIGHT is differentiated from the rest, and is the type of light which is being emitted by an object, whereas the other three types of light are usually used to describe a light source. Let's take a detailed look at each of these types of light:

AMBIENT LIGHT



A 3D sphere lit by AMBIENT LIGHT only; appears to look 2D. Ambient light is the average volume of light that is created by emission of light from all of the light sources surrounding (or located inside of) the lit area. When sun rays pass through the window of a room they hit the walls and are reflected and scattered into all different directions which averagely brightens up the whole room. This visual quality is described by ambient light. Ambient light alone cannot communicate the complete representation of an object set in 3D space because all vertices are evenly lit by the same color and the object appears to be 2-dimensional as seen in the image above. Although the object displayed is actually a 3D sphere, it appears to be flat on the screen, when lit only by ambient light.

DIFFUSE LIGHT

A diffuse light of red color is cast onto a black object defining its 3d shape.

Diffuse light represents a directional light cast by a light source. Diffuse light can be described as the light that has a position in space and comes from a single direction. A flashlight held slightly above the lit by it object can be though of as emitting diffuse light. In the image above a light source casting diffuse light of red color is located on the immediate left side of the object. When diffuse light touches the surface of an object, it scatters and reflects evenly across that surface.

To demonstrate how both AMBIENT and DIFFUSE lights work together to create a more-realistic looking object, imagine a 3D sphere with a dark red ambient light spread over it:

Now, by positioning a diffuse light source on the right side of the sphere, we get the following result:

Notice how the sphere now appears to be 3D.

SPECULAR LIGHT

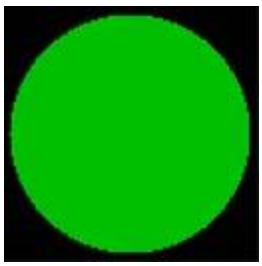
Specular reflection (or specular highlight) is displayed here in addition to the Ambient and Diffuse layers of light. You can observe how the object's 3D representation is greatly augmented by specular light properties.

Just like Diffuse light, Specular light is a directional type of light. It comes from one particular direction. The difference between the two is that specular light reflects off the surface in a sharp and uniform way. The rendering of specular light relies on the angle between the viewer and the light source. From the viewer’s standpoint specular light creates a highlighted area on the surface of the viewed object known as specular highlight or specular reflection. The intensity of the specular reflection is dependent on the material the object is made of and the strength of the light source which contains the specular light component.

EMISSIVE LIGHT

Emissive light is a little different than any other previously explained light components. The emissive light component is responsible for the object's material's property to reflect or absorb light. When applied to an object's material, emissive light acts to simulate the light that is reflected off the object.

With no additional light sources around, an object's color to which only emissive light component is applied has the same visual quality as an object with only ambient light applied to it. However, the mechanics of how any additional diffuse or specular light reacts with the surface of the same object with only emissive light applied to it is different. Let's consider an object which emits an average amount of green color. On the image below emissive light component is applied to the sphere. And as you can see, the result is similar to the effect created by applying ambient light to the same sphere in the example above.



A 3D sphere reflecting green emissive light. The effect is similar to ambient light until additional sources of light are introduced into the scene.

As you already know, a light source can have all of the three components assigned to it which are the ambient light, the diffuse light and the specular light components. Let's see what happens when we apply a light source to the above scene. The light source we are applying has the following properties: red ambient light, red diffuse light and white specular light.

If the above sphere wasn't emitting a light of green color, it would have appeared red in color. However, a green component of emissive light is applied to it. When the light source's "rays" of light hit the sphere's surface, the "source" and "destination" colors merge together producing a yellowish surface. The specular light component of the light source is white. The center of the specular reflection is white in the center, however as it spreads off it merges with the green and red colors, augmenting on yellow (which is green + red). Again, note that if there were no emissive light applied to the sphere, it would have appeared like the sphere shown under the section SPECULAR LIGHT above, all in red with a white specular reflection.

The way OpenGL shades polygons to simulate light and how light properties are assigned to light sources and materials is explained in the following part of this tutorial.

2. Light in OpenGL

Light there was indeed… but how do we program it in OpenGL?

First, let me recite a couple of previously explained functions. I take this straight from the OpenGL tutorial - 2 -Creating an OpenGL Window. But this time I will show you how these functions are used in light programming.

glEnable(int cap); This function has many uses. It can "enable" many features of OpenGL; the feature you want to enable is specified in the flag cap. In the base code, within InitOpenGL we use this command to enable all kinds of things. How is this related to light programming? glEnable is used to enable lighting on the whole when you pass GL_LIGHTING parameter to it. gEnable is also responsible for enabling a particular light source in your 3D scene. This is done by calling glEnable(GL_LIGHTn); where n is the index number of the color you are enabling, ranging from 0 to 7 because OpenGL lets you specify a maximum of eight light sources.

glDisable(int cap); This function disables whatever properties were previously set with glEnable.
glShadeModel(int mode); Selects the polygon shading model. mode is the flag representing the shading mode. This flag can be set to either GL_FLAT or GL_SMOOTH. GL_SMOOTH shading is the default shading model, causes the computed colors of vertices to be interpolated as the primitive is rasterized, assigning different colors to each resulting pixel fragment. GL_FLAT shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting, if lighting is enabled, or it is the current color at the time the vertex was specified, if lighting is disabled.

HOW OPENGL SHADES OBJECTS

The lighting model of OpenGL is based on the Gourad Shading implementation. A specific color is assigned to each of the vertices in a polygon. This color is calculated according to the object's material properties and surrounding light sources. Then the colors at each of the 3 vertices are taken and interpolated across the whole polygon.

OPENGL LIGHTING AND SHADING MODELS

Again, OpenGL has two important functions called glEnable(param) and glDisable(param). While these functions don't enable or disable anything in particular, they are reliant on the parameter passed to them. So in other words, you will be using these functions a lot when programming in OpenGL to turn a particular feature on or off.

To describe light in an OpenGL application you need to perform the following two steps: set the lighting and the shading models. But first, you have to enable the lighting system on the whole:

```
glEnable(GL_LIGHTING);
```

The lighting model is set up with a call to glLightModel and I describe this a bit later.

The shading model is set up with a call to glShadeModel and can be either set to SMOOTH or FLAT model. The SMOOTH shading model specifies the use of Gourad-shaded polygons to describe light while the FLAT shading model specifies the use of single-colored polygons. For this reason the objects shaded using the FLAT model look less realistic, however this mode is less computationally expensive. We're not going to use the FLAT model at all as it's not very visually impressive when it comes to rendering something rather realistic-looking, but you can still experiment with it at your own will. Setting a shading model will be described below.

OpenGL presents the programmer with a set of functions for use in the initialization part of your program to set specific global properties of the program behavior. The visual behavior of the lighting model is specified by the function glLightModel. There are two types of this function. One that uses scalar values as parameters and one for use with vector valued as parameters. Here are the definitions of both functions:

```
glLightModelf(GLenum pname, GLfloat param);          // scalar params
glLightModelfv(GLenum pname, const GLfloat *params); // vector params
```

THE GLOBAL AMBIENT LIGHT MODEL

In addition to specifying the ambient light amount to a light source or a material, it is possible to set the GLOBAL amount of the ambient light which will be cast on all rendered objects. This method of lighting is called the Global Ambient Light and in itself is considered a lighting model.

So, as an example of specifying a lighting model and for the sake of enabling the global ambient light model for the whole scene, a call to the glLightModelfv function is made. The specified parameters are GLOBAL_AMBIENT_LIGHT_MODEL, which tells OpenGL that we want a global ambient light model set; and global_ambient, which is the color of the global ambient light being set:

```
Glfloat global_ambient[] = { 0.5f, 0.5f, 0.5f, 1.0f };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);
```

This is normally done during the initialization part of your OpenGL application. It's important to note that the default global ambient light intensity is R=0.2, G=0.2, B=0.2, Alpha=1.0.

SMOOTH AND FLAT SHADING MODELS

After setting up global ambient light the next step is usually setting up the shading model. This is done by calling glShadeModel. As was described previously, we're going to use the smooth shading model in this tutorial and all consequent additions to the OpenGL base code. So, finally, to set it up we make the following call:

```
glShadeModel(GL_SMOOTH);
```

After this call, all of the polygons will be smoothly shaded by using the Gourad-shading technique and according to the nearby light sources and polygon's material properties.

Okay, now... OpenGL uses Gourad-shading while in theory, shading can be handled in one of two ways. In the traditional Gourad-shading technique the illumination is computed exactly at the vertices and the values are interpolated across the polygon. In Phong-shading technique, the normal vectors are given at each vertex, and the system interpolates these vectors in the interior of the polygon. Then this interpolated normal vector is used in the lighting equation. Phong-shading produces more realistic images, but takes considerably mode time to compute for the reason that 3 normals are used per polygon, as opposed to just 1.

Normal vector calculations are crucial for an acceptable lighting result.

OpenGL doesn't calculate normal vectors for you and this will be mentioned again below when we get to the COMPUTING SURFACE NORMALS section. Let me tell you however, about where and when the normal vector is specified. When you are composing an object, just before you specify one of its consequent vertices with a call to glVertex (as was explained in tutorial 2.5 - "Introduction to OpenGL primitives") for light to work properly you also have to specify the normal vector for that vertex with the glNormal command. If you don't specify the normal vector, the normal will contain its default value of (0.0, 0.0, 1.0) at every vertex. This, of course, will work but won't be acceptable in most cases since a more correct normal vector is preferred which should be calculated by you.

The normal vector calculation and vertex-assignment will be explained in time, for now let's delve into mechanics behind light properties.

THE TWO TYPES OF DEFINING LIGHT PROPERTIES IN OPENGL
(LIGHT SOURCE and SURFACE MATERIAL)

This section exists here because I want to explain something before describing how to define light sources and surface material reflective properties.

Theoretically, there are two types of light properties you should consider when programming the lighting model in OpenGL. The first type of light properties is the one that describes a *light source* and the second type of light properties is the one that describes the light reflected by the *material* of an object's surface. The same functionality goes into defining a light (which is done by defining its color), but the type of light you are defining can either be a light source or a surface material property. And the way light/color works in both of these cases is different.

The color of each light source is characterized by the color, in RGBA format, that it emits and is defined by using the function: glLight. The properties of the material the object is made of are defined by calling the function: glMaterial, and are characterized by the amount of light that the material reflects. The surface material properties are also characterized by the RGBA color format.

Now, let's take a look at how to define a light source in OpenGL...

DEFINING A LIGHT SOURCE

OpenGL allows a maximum of 8 light sources in a scene at once. Each of the light sources can be either enabled or disabled. All of the 8 light sources are initially disabled, and are enabled with a call to glEnable.

For each of the 8 light sources OpenGL has a name in the following format: GL_LIGHT*n*

Where *n* is the index number of the light source you are specifying. *n* can be a value ranging from 0 to 7. It should be obvious that to specify a light source number 1 you would use GL_LIGHT0. By order, the light source number 8 is specified as GL_LIGHT7.

To define a light source, OpenGL presents the programmer with a basic set of functions, specification of which divides itself into these already familiar to us light groups: Ambient Light, Diffuse Light, Specular Light and Emissive Light. For each of these types of light OpenGL #defines four models: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR and GL_EMISSIVE respectively.

So how to set a specific light type and enable it in your 3D scene? For each one of the light sources you need to call the glLightfv function with parameters which specify the what, and more importantly the

how. You've actually seen this in action already; as amusingly described above. To add a component of specular light to a light source, you would make the following function call:

```
GLfloat specular[] = {1.0f, 1.0f, 1.0f , 1.0f};
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
```

The whats here are the GL_LIGHT0 light source and the specular shading model described by the GL_SPECULAR parameter and the how is the GLfloat specular parameter which defines properties of the specular reflection you are setting up for the GL_LIGHT0 light source. specular[] is a 4-parameter array. The parameters are described as:

```
specular[] = { floatRed, floatGreen, floatBlue, floatAlpha};
```

The first three parameters are the RGB values which can range from anywhere between 0.0f and 1.0f. 0.0f being no color, and 1.0f being full color. 0.5f would identify an average between the very bright color and the very dim color. But if you've read the OpenGL - 4 - color tutorial you should already be familiar with how to assign a color to a vertex in a polygon. Here, this is done in the exact same way, the difference is that you are assigning the color to a light source and in this case it is the color of the specular reflection you are assigning to GL_LIGHT0.

Another diminutive difference when defining a light source (as opposed to polygon color) is that of the addition of the new variable 'floatAlpha' as the fourth parameter in the array. In reality you would only specify this parameter for an EMISSIVE light component to modify an object's material's alpha value or if you're programming some kind of a advanced special effect, but more on this in later tutorials. An emissive light property is typically assigned to a polygon, not a light source. Here it is shown just to tell you that there is a fourth parameter which can also exists when creating a light component. The floatAlpha parameter is the polygon's ALPHA value and should be set to 1.0f for now or simply skipped because its default value is 1.0f in any case.

So what is this alpha parameter anyway? The alpha parameter is used to define the translucency factor of an object and could be used to simulate materials made out of translucent matter such as glass. Alpha is also used to blend the lighted areas with the texture of a 3D model. Texture-mapping will be explained in the following tutorials. The color format which describes the RGB values of color as well as the alpha value is referred to as the RGBA format. We will eventually get to this in detail later. For now lets concentrate on how to specify the full range of properties for a light source.

The same glLightfv mechanism is used to specify any of the other three shading models, if required by your light engine. For example, to set and enable the Ambient Light component of a light source so that it emits Ambient Light of moderately pale white color (R=0.5f, G=0.5f, B=0.5f) you could use the following call and parameters:

```
// note, no alpha is required since it has no use in specifying a light
// source (in this case). Keep in mind however, that its default value
// is 1.0f anyway
GLfloat ambient[] = { 1.0f, 1.0f, 1.0f };
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
```

But assigning ambient, diffuse and specular types of light to a light source is not usually enough. You also have to specify the position of the light source. And this is, similarly to defining light components, done with the glLight function in the following way:

```
Glfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };
GlLightfv(GL_LIGHT0, GL_POSITION, position);
```

This rounds up our discussion of assigning light properties to a light source. In the following part I will explain how to create a fully functional light source and finally enable it in the scene.

ENABLING (AND DISABLING) A LIGHT SOURCE

And finally, if you want to use the newly created light source, you have to enable it with a call to glEnable:

```
glEnable(GL_LIGHT0);
```

All of the provided 8 OpenGL light sources are disabled by default. And just for convenience, if you want to turn off an enabled light source, you would use the glDisable function:

```
glDisable(GL_LIGHT0);
```

The following is an example of how you would set up all essential parameters of a light source. The are really no solid rules as to what parameters to assign to a light source, the light engine programmer is required to tweak the values until things just look good. Experiment with your light sources' parameters and eventually you will understand how changing them affects the visual outcome.

```
// Somewhere in the initialization part of your program…
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Create light components
Glfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
Glgloat diffuseLight[] = { 0.8f, 0.8f, 0.8, 1.0f };
Glfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
Glfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };

// Assign created components to GL_LIGHT0
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Well, and that's that. The light source GL_LIGHT0 is enabled and from now on will start to shine light onto the objects in the surrounding 3D space. Now that you are able to create and assign light components to an OpenGL light source, let's take a look at what materials represent in the real world and how to assign material properties to polygons.

MATERIALS IN REAL WORLD

In nature, objects are lit by the sun which emits light of white intensity. White, as we know, is a combination of all colors. When this white light hits the surface of an object, some wavelengths of light are reflected and some are absorbed. The light that is reflected defines the color of the object we're viewing. Different objects are made out of different matter which occupies different reflectance properties. For example, a red ball reflects only the red particles of light and absorbs all of the others. Under white light, all objects appear to be of their "natural" color because the white light contains all of the colors together, so the object always has a color to reflect. However, try viewing the same red ball in a room with a blue light bulb as the only light source and the ball will appear to be black because there is no red color to reflect.

DEFINING SURFACE MATERIAL PROPERTIES

Before going further into the surface reflective property mechanics, I have to point something out as this can confuse some of the beginner-level OpenGL programmers.

In OpenGL, by assigning a material property to an object (defined by RGB color format), you are theoretically assigning the color which is reflected by that object. When assigning a color to an object with the glColor command (as described in OpenGL tutorial - 4 - Color, and lighting cannot be enabled or used in this case, you are only assigning a color simulation of the object) you are merely assigning the property which describes that object's color, it will not react with light and will remain just that - the object's color. But in that case lighting is assumed to be turned off anyway, so this should make sense. This is the difference between assigning a color to an object and assigning a material property to an object.

Keep in mind that when you enable lighting with glEnable(GL_LIGHTING) you are generally expected to be assigning material properties with the glMaterial command as shown in the following code sample:

```
float mcolor[] = { 1.0f, 0.0f, 0.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mcolor);
// now, draw polygon as its material properties will be affected by the
glMaterialfv call.
```

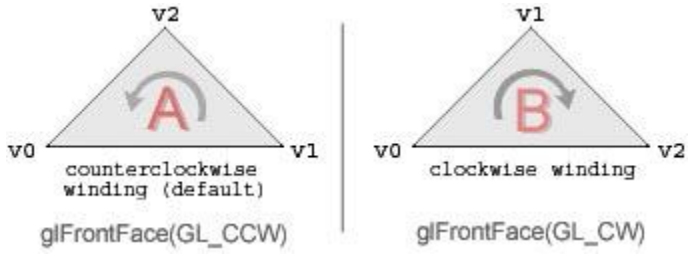
This version of glMaterial postfixed by fv takes vector-based color coordinates. The GL_AMBIENT_AND_DIFFUSE parameter specifies that mcolor will be applied to both ambient and diffuse components of the material. This is done for convenience because in most cases Ambient and Diffuse properties of a material should be set to the same color.

POLYGON WINDING

The first parameter of the glMaterialfv command described above (GL_FRONT) indicates which face of the polygon should reflect the light specified by mcolor. This requires a bit more explanation. Apparently, there are two sides to a polygon - front and back. But how exactly do we know which side is the front (defined in OpenGL as GL_FRONT) and which side is the back (defined as GL_BACK) of a polygon? The answer is simple. There are two ways of specifying a polygon in 3D space. By specifying vertices one by one in clockwise direction and specifying vertices in counterclockwise direction. The direction you specify the vertices in is what describes which side is the front and which is the back. OpenGL lets you specify these rules with the glFrontFace command. The mechanics which describe polygon defining direction is called polygon winding.

```
glFrontFace(GL_CCW);
```

This sets the default behavior. The CCW in GL_CCW is an abbreviation for CounterClockwise Winding. Let's take a look at the following image to better visualize polygon winding and move on with the surface material properties enlightenment.



To define a polygon in counterclockwise direction, you first specify the first vertex - v0, continue defining its base by v1 and concluding the polygon with v2 as its peak point as seen in the example A. With this configuration, the polygon will be defined as having counterclockwise winding and the visible, front side will be facing the viewer if GL_CCW (which is the default setting) is set with glFrontFace. If you try defining a polygon in clockwise direction, as seen in the example B, and GL_CCW is enabled, you will not see the polygon because it's face will be pointing away from the viewer. GL_CW flag is used in that situation.

There is a possibility to apply light to both of the polygon sides. If you're doing that, you would like to set the first parameter of glMaterial to GL_FRONT_AND_BACK.

DEFINING SURFACE MATERIAL PROPERTIES (CONTINUED)
glMaterial vs. color tracking

So now that we're familiar with polygon winding, let's go over this again quickly. To define reflective material properties of the polygon's surface you have to specify them by calling the glMaterial command prior to defining the polygon's vertices:

```
// Evaluate the reflective properties of the material
float colorBlue[] = { 0.0f, 0.0f, 1.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, colorBlue);

// Draw a polygon with current material properties being set
glBegin(GL_TRIANGLES);
glVertex3f(-1.0f, 0.0f, 0.0f);
glVertex3f(0.0f, -1.0f, 0.0f);
glVertex3f(1.0f, 0.0f, 0.0f);
glEnd();
```

What I haven't mentioned until now, is that this way of defining material properties gets tedious when dealing with a large number of polygons. It is an acceptable way of defining materials but a more convenient alternative is called color tracking. With this method you are able to specify material properties by merely calling the glColor command prior to each object or polygon for which you're specifying those material properties. Just using glColor without enabling color tracking won't do anything. To use color tracking you have to enable it. And that is done with the following command:

```
// enable color tracking
```

```
glEnable(GL_COLOR_MATERIAL);
```

Also, you have to predefine the material properties which will be consequently assigned to objects when you define them with the glColor command. This is done by making the following call:

```
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

Here, we tell OpenGL that each time we will use the glColor command, prior to defining an object or a polygon with a series of glVertex calls, the ambient and diffuse qualities of the material properties specified by the glColor command will be assigned to that object/polygon. As an overview, here is how we would use color tracking to define the same material properties of a polygon/triangle as one explained above, whenw e used glMaterial:

```
// ...Somewhere during initialization...

// enable color tracking
glEnable(GL_COLOR_MATERIAL);
// set material properties which will be assigned by glColor
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);


// ...Somewhere in the main rendering loop of your program...
// Draw a polygon with material properties set by glColor

glColor3f(0.0f, 0.0f, 1.0f); // blue reflective properties
glBegin(GL_TRIANGLES);
glVertex3f(-1.0f, 0.0f, 0.0f);
glVertex3f(0.0f, -1.0f, 0.0f);
glVertex3f(1.0f, 0.0f, 0.0f);
glEnd();
```

Keep in mind, the material properties of the object specify the kind and the amount of light that is reflected off that object. The three components of light properties which are applied to the object to define its material's appearance are the AMBIENT, DIFFUSE and SPECULAR components. The components of a material color are set with the glMaterial command in a similar way that a color of a polygon is set with glColor. glMaterial is called before you specify a vertex in a polygon to set reflective properties for that vertex and all other consequent vertices. glColor, also can be used to specify material properties, but you have to enable color tracking for this functionality to work.

MATERIAL PROPERTY (DE)COMPOSITION

The AMBIENT component defines the overall color of the object and has the most effect when the object is not lit. The DIFFUSE component has the most importance in defining the object's color when a light is emitted onto its surface. It is convenient to always set AMBIENT and DIFFUSE components of the material to the same value unless you are doing something other than ordinary lighting. This has been previously demonstrated above. What we haven't talked about yet is the specular or shininess component of the material properties.

It is enough to define ambient and diffuse color components of a material for the object to appear realistically lit by a theoretical light source. However, this is not always the case. For example when

modeling an object made out of metal it is desired for that object to have more emphasized reflective properties for it to look more like metal. These properties divide themselves into two categories: the SPECULAR component and the SHININESS component.

The SPECULAR REFLECTION component

The SPECULAR REFLETION component of the material defines the effect the material has on the reflected light. This functionality can be obtained by defining the GL_SPECULAR component of the object's material. This property adds a glossy surface to the object you are modeling.

The SPECULAR EXPONENT (shininess) component

An extension to the SPECULAR REFLECTION component lies in the use of shininess, or the SPECULAR EXPONENT (Keep in mind, shininess is the second feature of the specular light component and not a light-type by itself). By defining GL_SHININESS property of a material you control the size and brightness of the specular reflection. The specular reflection is the very bright white spot which appears on objects with materials of high reflectivity. You really have to fiddle with both parameters to see exactly what effect they produce on a surface of a material.

So to completely define the **specular** component of a material you have to define both GL_SPECULAR and GL_SHININESS properties.

It is important to know that for the specular reflections to work you have to set both: the light source's specular light component and the object's material specular reflectance component. I already explained how to assign light properties to a light source, including the specular light component in DEFINING A LIGHT SOURCE. So, here I will only describe how to add a specular component to an object's material properties:

```
float specReflection[] = { 0.8f, 0.8f, 0.8f, 1.0f };
glMaterialfv(GL_FRONT, GL_SPECULAR, specReflection);
```

This property, as you can see, is set in the exact same way as you set any other properties of a material with the glMaterial command. After calling this command all consequent object compositions will contain specular reflection properties, until you make another call to glMaterial to modify the current specular component.

As previously explained, in some cases it is desired to set the specular exponent property of the specular reflectance. And additionally, this is done by assigning the GL_SHININESS model to the global material property in the already familiar way:

```
glMateriali(GL_FRONT, GL_SHININESS, 96);
```

Here, we are using the version of glMaterial command which utilizes the i postfix, meaning the value of the shininess factor will be evaluated by an integer variable. The integer parameter's range is specified by a value anywhere between 1 and 128. 128 being the brightest intensity. So, by setting this parameter to 128, a very bright specular highlight is achieved. In this case I set it to a slightly less bright value (I assure you, it has nothing to do with the fact that I dislike direct sunshine or a lamp turned on during a tv session).

With practice you will realize where exactly to place a call to glMaterial command to modify your object's material properties in any way you want. It is also useful to look at the source code for this tutorial which gives you a raw example of how all of the explained things work together.

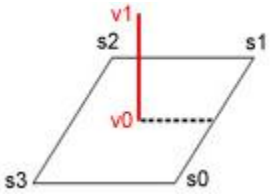
Well that was a lot of ground to be covered and consequently consumed by your brain. I am, in fact, not sorry for such a perturbation placed onto your brain cells which by now are probably screaming with an excessive amount of electric charges.

AND JUST WHEN YOU THOUGHT YOU WERE DONE...

...there is yet another important concept to understand when it comes to programming light in OpenGL. Even if all of your light sources are set in place and all material objects are defined to what you want them to be, it's still not quiet enough. For the light equations to work properly, it is required to know the direction which the polygons in your model are facing. This direction is defined by a normal (or a perpendicular, if you will) vector in relation to the polygon's surface. The next topic covers this mechanism.

COMPUTING SURFACE NORMALS

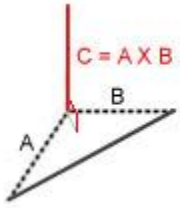
In order to display light correctly you are required to compute normal vectors for each polygon in an object.



The theoretical normal vector of the surface s0, s1, s2, s3 is indicated in red, defined by two points v0 and v1.

A normal of a polygon is a perpendicular vector in relation to the polygon's surface and is very useful in many implementations of 3D computer graphics when dealing with surface direction mechanics. Sorry, but OpenGL doesn't compute normal vectors for you and this is something you have to do on your own.

Since all models in your 3D scene will be made out of polygons, it is convenient to have a function which calculates the normal vector of a polygon. A normal vector of a polygon is the cross product of two vectors located on the surface plane of the polygon (in the case below that polygon is a triangle). And what we need to do is take any two vectors located on the polygon's plane and calculate their cross product. The cross product will be the resulting normal vector. Mathematically the cross product of two vectors A and B is represented as A X B. Take a look:



A normal vector C as the cross product of two vectors A and B.

The shown triangle's vectors A and B are taken. The result (the red vector) is the normal vector which defines what way the triangle is facing. The following function calculates the normal vector of a given polygon. The parameters are vertex_t v[3]; which defines the 2 vectors that lie on the polygon's plane and vertex_t normal[3]; which will hold the resulting normal vector. Keep in mind that if you are using counterclockwise winding (as this is the default OpenGL behavior), by order you must specify the points of v[3] in counterclockwise direction as well.

```
// This is how a vertex is specified in the base code
typedef struct vertex_s
{
    float x, y, z;
} vertex_t;

// normal(); - finds a normal vector and normalizes it
void normal (vertex_t v[3], vertex_t *normal)
{
    vertex_t a, b;

    // calculate the vectors A and B
    // note that v[3] is defined with counterclockwise winding in mind
    // a
    a.x = v[0].x - v[1].x;
    a.y = v[0].y - v[1].y;
    a.z = v[0].z - v[1].z;
    // b
    b.x = v[1].x - v[2].x;
    b.y = v[1].y - v[2].y;
    b.z = v[1].z - v[2].z;

    // calculate the cross product and place the resulting vector
    // into the address specified by vertex_t *normal
    normal->x = (a.y * b.z) - (a.z * b.y);
    normal->y = (a.z * b.x) - (a.x * b.z);
    normal->z = (a.x * b.y) - (a.y * b.x);

    // normalize
    normalize(normal);
}
```

The final step of this function is to normalize the resulting vector and this is something I haven't talked about yet. Normalization of the normal vector is explained in the following section.

THE NEED FOR NORMALIZATION

To normalize a normal vector means to reduce its length to unit size. A unit size is just 1. All of the calculated normal vectors are required to be a length of 1 to work properly with OpenGL lighting system. It would be possible to tell OpenGL to normalize normal vectors for us with a call to glEnable(GL_NORMALIZE); but believe me, that would be more computationally expensive than doing normalization within your own code.

So how exactly the normalization process is done? As stated above, all we have to do is reduce the size of a given normal vector to a length of 1. To accomplish that, first you have to find the length of a normal vector. To find the length of any vector, you take all of the coordinate components (x, y and z) of that vector and square them. Add all of the squared components and find the square root of that sum. This sum will be the length of the vector. Afterwards, divide each coordinate component of the vector by its derived length and you will get a vector which points in the same exact direction but of unit length. And the function normalize does precisely that:

```
// This is how a vector is specified in the base code
// The origin is assumed to be [0,0,0]
typedef struct vector_s
{
    float x, y, z;
} vector_t;

void normalize (vector_t *v)
{
    // calculate the length of the vector
    float len = (float)(sqrt((v.x * v.x) + (v.y * v.y) + (v.z * v.z)));

    // avoid division by 0
    if (len == 0.0f)
        len = 1.0f;

    // reduce to unit size
    v.x /= len;
    v.y /= len;
    v.z /= len;
}
```

THE FINAL SHADING FORMULA EXPLAINED

As this tutorial coming to an end, let's overview some light mechanics and then I will let you delve right into the source code.

The final appearance of shading on the object is reliant on the combination of the global ambient light; ambient light, diffuse and emissive lights emitted by surrounding light sources and the lit object's material properties:

The lit color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source.

Each light source contributes the sum of three terms: ambient, diffuse, and specular. The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity. The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source. The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material.

All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with 0 if they evaluate to a negative value. Finally, the alpha component of the resulting lit color is set to the alpha value of the material diffuse reflectance.

And that's all. This concludes our discussion of light. In the following tutorial we will discover the next progression of representing the visual quality of objects rendered on the screen - texture-mapping.

SOURCE CODE

In conclusion, the source code which demonstrates all of what's been described in this tutorial. I can't stress the importance of reading additional OpenGL literature. More than half of the information I know comes from those books. In the source code, a simple object is loaded using the object loader from OpenGL tutorial 6, the light shading models are set up and enabled, the light sources are placed at specific locations in the 3D world, the user is able to utilize the mouse movement to rotate the object on its axis demonstrating how the light interacts with the surface. Enjoy.