Chapter 11

Color and Light

Our objects so far have mostly looked as if they plan to spend the afternoon home watching the game. It's time now to dress up and go party. The goal for this chapter is to learn how to use light sources to illuminate a scene and complementarily define material properties of objects to determine how they appear when lit.

We begin with a brief discussion of the theory of vision and color models in Section 11.1, learning particularly about the RGB color model so important in CG, as well as a few other models which pop up occasionally, such as CMY, CMYK and HSV. In Section 11.2 we study Phong's lighting model and how it conceives of light coming off an object as comprised of three components – ambient, diffuse and specular – based on the nature of their reflectance. This section concludes with a formula to derive the RGB intensities of the light reflected at a vertex based on Phong's model.

We move on to OpenGL in Section 11.3 and see how faithfully it implements Phong's model. And, we begin extensively to experiment and code. In Section 11.4 we describe OpenGL's so-called lighting model – not to be confused with Phong's lighting model – which sets certain environmental parameters. Directional light sources, located far from the scene, and positional lights, located in or near it, are discussed in Section 11.5, as is the related notion of attenuation of light over distance. Spotlights are the topic of Section 11.6. At this point we have all the parts needed to formulate in Section 11.7 the famous lighting equation that OpenGL actually implements to calculate color intensities at a vertex.

We discuss the two so-called shading models OpenGL offers, smooth and flat, in Section 11.8. The former familiarly interpolates the vertex colors through a primitive, while the latter is a somewhat idiosyncratic discrete coloring scheme. Animation of light sources is the topic of Section 11.9.

Specifying appropriate surface normals is critical to good lighting.

OpenGL can sometimes help with automatic normals, but often the user is on her own and the task can require a fair amount of calculation. Before we begin with normal computation proper, we have an optional introduction in Section 11.10 to the calculus of partial derivatives, to the extent required to calculate tangent planes and normals to the kinds of surfaces typical in CG. We recommend it be skipped at first and consulted subsequently if need be.

The long Section 11.11 is devoted to computing and applying surface normals to lighting. It begins by following the informal taxonomy of 2D objects introduced in Section 10.2, and moves on to Bézier and quadric surfaces for which automatic normals are available.

Section 11.12 contains a discussion of an alternate shading model proposed by Phong, which is more sophisticated (and more computation-intensive) than OpenGL's smooth shading. We conclude in Section 11.13.

11.1 Vision and Color Models

We begin with a little of the physics and biology underlying color and its perception.

Electromagnetic (EM) radiation consists of oscillating electric and magnetic fields moving through space. It is produced by the motion of electrically charged particles. From a physics point of view, EM radiation can be treated dually as waves or a stream of massless particles called photons traveling through a vacuum at the speed of light. EM radiation is characterized by its frequency or, equivalently, wavelength, the inverse of frequency. The EM spectrum consists of EM radiation of all possible frequencies, of which visible light is a very small part. See Figure 11.1.

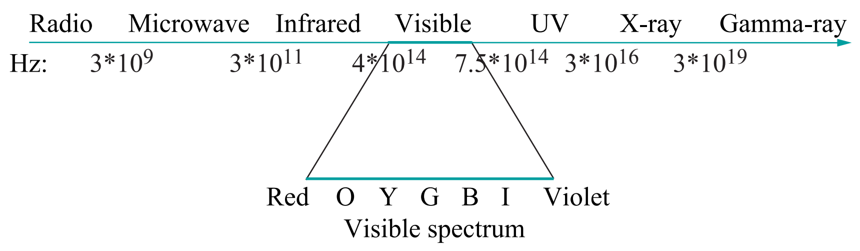


Figure 11.1: EM spectrum indicating approximate frequency ranges in Hz.

Visible light emitted from a source is rarely pure, i.e., of one particular frequency. Rather, there is an intensity distribution across the entire visible spectrum, and the perceived color depends on the particular distribution. Light from a source with an intensity distribution, for example, as in Figure 11.2(a), would be perceived as blue, as this color's intensity dominates, while one with the distribution of Figure 11.2(b) would appear white, because white is a mix of all colors in the visible spectrum.

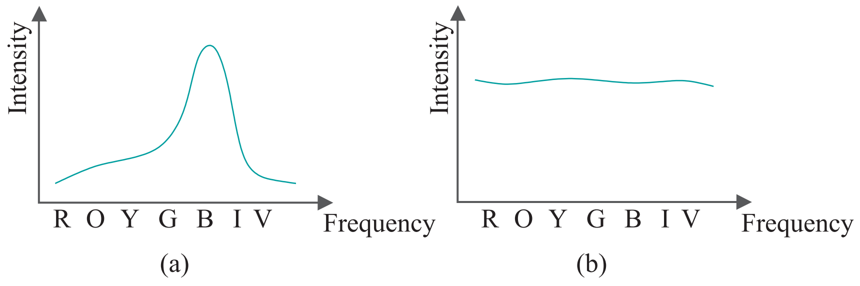


Figure 11.2: Intensity distributions across the visible spectrum: (a) appears blue (b) appears white.

We humans can see because of millions of light-sensitive cells embedded in the retinas of our eyes (see Figure 11.3 for a simplified anatomy). These cells are of two kinds, rod and cone. Rod cells are sensitive to low-intensity light, but not its frequency, which accounts for our night vision, as well as the fact that we have particular difficulty distinguishing colors in the dark. Cone cells, on the other hand, are stimulated only by fairly bright light, but can efficiently distinguish frequencies in the visible light spectrum, enabling us to perceive color. In fact, there are three kinds of cones – red, green and blue – according to the color of the light that most stimulates them. This is the basis of the *tristimulus* theory of human vision that perceived color is the net effect of the stimulation of these three kinds of cells.

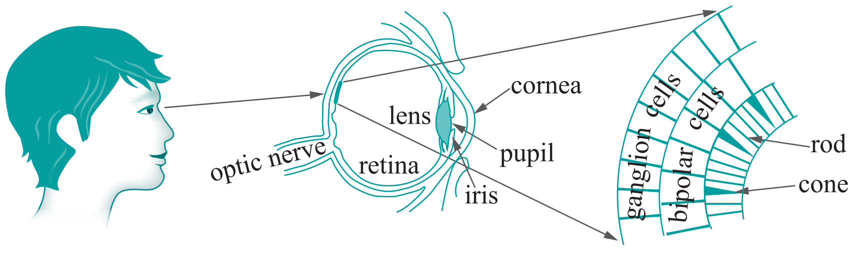


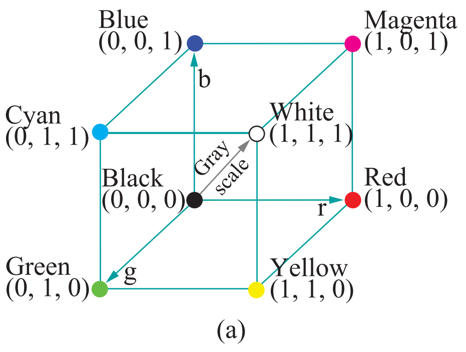
Figure 11.3: The eye.

11.1.1 RGB Color Model

A consequence of the tristimulus theory is the ubiquitous *RGB color model*: each color is represented as a sum of the three *primary colors*, red, green and blue, and each with a certain intensity, typically a value between 0 and 1 (for this reason RGB is called an *additive color model*). A color is denoted by a *color tuple* (r, g, b), where each component is the respective primary color's intensity.

Note: An intensity distribution curve, as those in Figure 11.2, one corresponding to each primary color, has been standardized by the International Commission on Illumination (CIE, from its French name Commission Internationale de L'Éclairage), as also a standard to convert intensity distributions across the visible spectrum to RGB triplets.

The RGB color space can be depicted as a cube, called a *color cube*, with axes corresponding to R, G and B values (see Figure 11.4(a)). The origin $(0, 0, 0)$ of the cube corresponds to black, while its diagonally opposite vertex $(1, 1, 1)$ to white, which, of course, is the maximal equal mix of red, green and blue. The other three diagonally opposite pairs each corresponds to a primary color and its complement (the complement of a color being that which with it combines to produce white). The straight line segment from black to white, each point (x, x, x) of which has equal parts of the primary colors, represents the gray scale. Figure 11.4(b) is a popularly drawn Venn diagram, where discs correspond to primary colors and their intersections are colored according to the mixing of the primaries.



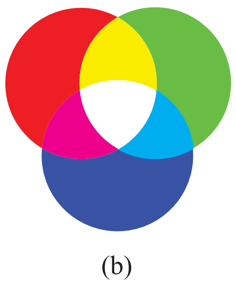


Figure 11.4: (a) The RGB color cube (b) Venn diagram combining colors.

The mechanics of the "addition" of colors in the RGB model is interesting. The color cube, for instance, indicates that an equal mix of red (which is $(1, 0, 0)$) and green ($(0, 1, 0)$) produces yellow ($(1, 1, 0)$). The reason for this is that the sensation produced in the human eye by a mix of two lights, one whose red frequency dominates and another whose green dominates, is similar to that produced by a single light dominant in the yellow frequency. This is a consequence of how our optic nerves react to the stimulation of particular combinations of cone cells, and *not* because the frequencies of red and green combine according to some law of physics to produce that of yellow! The RGB model, therefore, rests more on the biology of human vision than the physics of light.

RGB Color Model and Computer Graphics

The RGB model is implemented in millions of color display units around us, including computer monitors, both CRT and LCD. A CRT (cathode-ray tube) monitor has phosphors of the three primary colors located at each one of a rectangular array of pixels, and three electron guns that each fires a beam at phosphors of one color. A mechanism to aim and control their intensities causes the beams to travel together row by row, striking successive pixels in order to excite the RGB phosphors at each to values specified for it in the color buffer. See Figure 11.5(a).

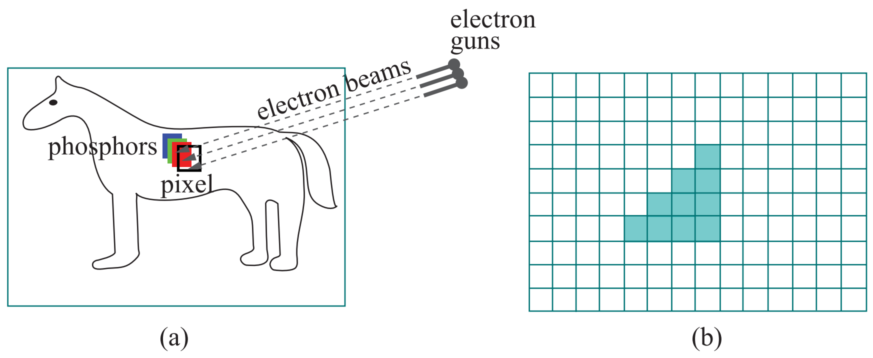


Figure 11.5: (a) Color CRT monitor with electron beams aimed at a pixel with phosphors of the 3 primaries (b) A raster of pixels showing a rasterized triangle.

Pixels in an LCD (liquid crystal display) monitor, on the other hand, each consist of three subpixels made of liquid crystal molecules, which separately filter through light of only one primary color. The amount of primary color emerging through each subpixel is controlled by an electric charge, whose intensity in turn is determined by the corresponding value in the color buffer.

From the point of view of OpenGL and, indeed, most CG theory, what matters most is that the pixels in a monitor, CRT or LCD, are, in fact, arranged in a rectangular array, called a *raster* (as depicted in Figure 11.5(b)). The number of rows and columns in the raster determines the monitor's resolution. For, this layout is the basis of the lowest-level CG algorithms, the so-called raster algorithms, which actually select and color the pixels to represent user-specified shapes such as lines and triangles on the monitor. Figure 11.5(b), for example, shows the rasterization of a right-angled triangle (with terrible jaggies because of the low resolution). We'll be studying raster algorithms in fair depth ourselves in Chapter 14.

Furthermore, a memory location called the *color buffer*, either in the CPU or graphics card, contains, typically, 32 bits of data per raster pixel – 8 bits for each of RGB and 8 for the alpha value (used for blending). It is the RGB values in the color buffer which determine the corresponding raster pixel's color intensities. The values in the color buffer are read by the raster – at which time the raster is said to be refreshed – at a rate called the monitor's refresh rate. Beyond this, the technology underlying a particular display device matters little practically in computer graphics.

11.1.2 CMY and CMYK Color Models

The *CMY color model*, whose augmentation CMYK is typically used in color printing, is a *subtractive color model*. CMY stands for cyan, magenta and yellow, and they are, respectively, the complements of red, green and blue. For example, cyan reflects blue and green but absorbs (or subtracts) red. Likewise, magenta and yellow subtract green and blue, respectively. Accordingly, cyan, magenta and yellow are referred to as the *subtractive primaries*. The color cube and Venn diagram for the CMY color model are depicted in Figure 11.6.

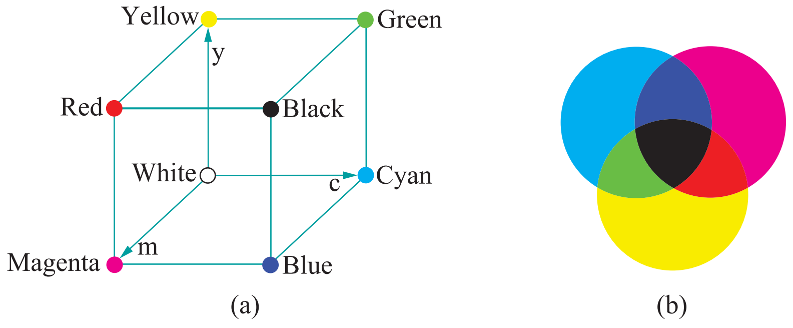


Figure 11.6: (a) The CMY color cube (b) Venn diagram of the CMY model.

Going between the RGB and CMY color spaces is simple:

$$\begin{bmatrix} c \\ m \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} r \\ g \\ b \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} c \\ m \\ y \end{bmatrix} \tag{11.1}$$

Ink of the color of each of the three subtractive primaries is coated as a grid of dots (called a *screen*) on a sheet of paper during printing. The relative proportions of CMY ink at each dot determines the amounts of light of various frequencies subtracted there; the remaining light emerges through the ink layers and imparts to the dot its perceived color.

However, in practice, the combination of CMY ink to produce RGB color does not work as well as the equations (11.1) might suggest. The CMY pigments in printer toner cartridges are never pure enough that an equal mix produces 100% black or even proper shades of gray. In addition to this technical problem there is an economic one too: making blacks and grays, by far the most common colors in printing, by mixing colored inks is expensive.

Modern color printers, accordingly, supplement their CMY inks with a black ink to directly produce both blacks and the gray scale, the resulting process called *four-color printing*.

The CMY model augmented with black is called the *CMYK color model* (for reasons to do with printing terminology black is denoted by K rather than B). Conversion formulae between the CMYK color space and the RGB and CMY color spaces are more complicated than those between the latter two and we'll not discuss them here. However, drawing and image editing programs, such GNU's GIMP (freeware [49]), which offer both RGB and CMYK models will automatically convert between the two.

A practical point to keep in mind is that mapping from RGB to CMYK is often device-dependent and rarely 100% accurate, which is why CMYK print-outs are frequently significantly different from the original RGB display. Moreover, the space of colors representable in the RGB and CMYK color models – their *gamuts* – are not identical either, so some colors simply cannot be transferred exactly from monitor to paper (or vice versa).

11.1.3 HSV (or HSB) Color Model

The RGB color model, though pretty much ingrained into applications around us, is not particularly intuitive for the mixing of colors. For example, what RGB values should an artist combine for a jungle green, sunset orange, ocean blue, ...? The *HSV color model* was created by Alvy Ray Smith (one of the co-founders of Pixar Corporation) in 1978 as a more user-friendly alternative for designers. HSV is the abbreviation for hue, saturation and value. The model is also called *HSB*, where B stands for brightness.

The HSV model gets past the problem of having to numerically mix primaries by allowing the designer to choose a color's "coloredness" (that which we perceive as jungle green, sunset orange, ocean blue, etc.) directly with a *single* parameter, the *hue*. The hue parameter space is circular and often called the *color wheel*.

Here's how the color wheel is derived. See Figure 11.7. Begin with a triangle with corners representing the red, green and blue hues. Double the number of vertices to make a hexagon and fill in the middle hues, yellow, cyan and magenta (e.g., yellow is an equal mix of red and green, so midway between them). Again double to a dodecagon and add new hues by interpolating between previous ones. Continuing the process leads to a continuum of hues in a circle. A position on this circle – or, the color wheel as it's called – thus represents a particular hue. Typically, red, green and blue are located at $0°$, $120°$ and $240°$, respectively.

The hue parameter by itself is insufficient to specify a color. Two other parameters are required as well. The *saturation* of a color, typically given as a percentage, represents its purity. The higher the saturation the richer and more vibrant the color appears; conversely, less saturated colors (called *desaturated*) appear faded and grayish. The final parameter is *value*, given

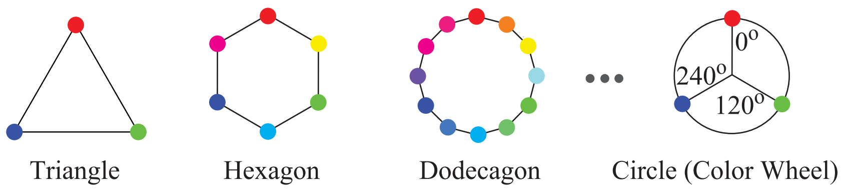
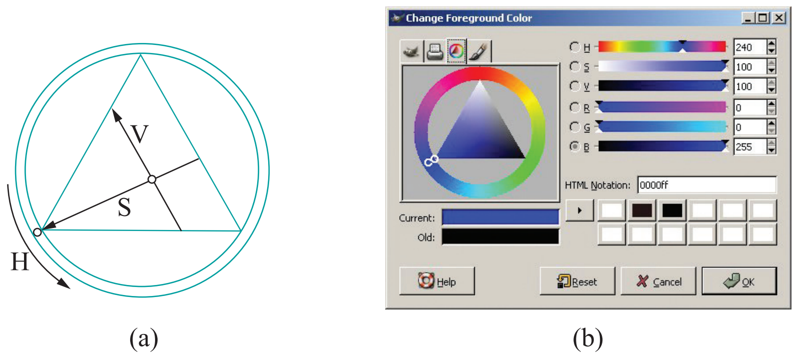


Figure 11.7: Hues on a triangle, hexagon, dodecagon and circle (color wheel).

as a percentage as well, representing a color's intensity or brightness.

The saturation and value amounts of a color are often specified by positioning a point on an equilateral triangle inscribed in the color wheel, with a vertex of that triangle located at and turning with the color's hue position on the wheel. As indicated in Figure 11.8(a), value changes parallel to the edge opposite the hue vertex, while saturation increases with distance from the opposite edge. Figure 11.8(b) shows GIMP's color dialog box setting a 100% blue in HSV mode.



H 240
S 100
V 100
R 0
G 0
B 255
HTML Notation: 0000ff

Figure 11.8: (a) The outer small hollow circle is positioned on the colored wheel to set the H value and the inner one inside the triangle to set S and V values. (b) The Gimp color dialog box making a 100% blue using the HSV color wheel option.

11.1.4 Summary of the Models

In drawing applications the predominant model is RGB and we'll really not have use for any other through the rest of this book. It's useful, though, to have a nodding acquaintance with models that may occasionally pop up elsewhere. With CMY, CMYK and HSV we have covered the ones the user is most likely to encounter. CMYK does, in fact, become particularly important when one goes from drawing to printing. There are other color models not used as much, such as Lab (an option in Adobe's Photoshop) and HLS (for hue, lightness, saturation).

The gold standard among color models was established by the CIE in 1931. It's called the *CIE XYZ model* (also the *CIE 1931 model*) where the X, Y and Z parameters represent, respectively, three theoretical primaries, each corresponding to a particular intensity distribution standardized by the CIE. Although not seen in practical interfaces, the CIE XYZ color model is used to calibrate implementations of the other ones.

11.2 Phong's Lighting Model

A model of interaction between light sources and objects is called a *lighting model* (or *reflection model*, or *illumination model*). In 1975 Vietnamese computer scientist Phong Bui Tuong [105] invented a particular lighting model, now known by his name, which is currently the one most widely used in practice. Despite the subsequent development of more authentic lighting models, e.g., Cook-Torrance [28], ray tracing, etc., Phong's has endured in popularity, especially because it delivers realistic lighting at moderate computational cost. OpenGL implements Phong's model. But, before we start coding up light let's first get an understanding of the model.

11.2.1 Phong Basics

In Phong's model the light reflected off an object O is the sum of three components – *ambient*, *diffuse* and *specular* – based on the *reflectance* properties of its surface. We'll describe each component next before explaining how to specify and calculate them.

Ambient: Ambient reflectance models O's reflection of background light that strikes it from multiple directions. Ambient light is scattered equally in all directions from the surface of O as well because of fine-scale graininess. See Figure 11.9.

Of the light sources in the environment – e.g., lamps and the sun – a part of the light from each is presumed ambient in that it's scattered by minute particles such as dust in the environment, effectively becoming part of background light before striking O. The direction of the light's source, therefore, is lost in that part of it which is ambient. Neither does it matter where the viewer is located because of the scattered reflection from the surface of O, presumed equal in all directions. In practical terms, the ambient component models that part of light which supplies constant illumination throughout a scene. An example of a familiar light source which is mostly ambient is a tube lamp recessed behind a frosted panel.

In addition to the ambient parts of each light source, there is presumed to be a *global ambient light* as well, from no identifiable source (i.e., "true" background light). For example, when modeling a scene inside a building, we can adjust the global ambient to account for light coming in from outside through doors and windows, without trying to model every possible light source such as the sun and lights on the street, which would be very complex indeed.

The total ambient component of the light reflected from an object O is the sum of what it reflects of the ambient parts from each source, plus what it reflects of the global ambient. Informally:

$$\text{ambient reflectance from } O = \sum(\text{reflectance of ambient part from each light source}) + \text{reflectance of global ambient}$$

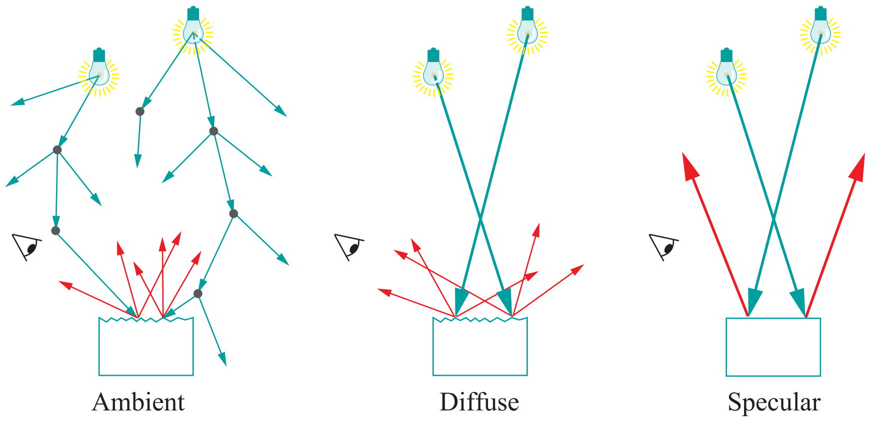


Figure 11.9: Ambient, diffuse and specular reflectance: incident light drawn blue, reflected red.

Diffuse: Diffuse reflectance specifically models the fine-scale graininess of the surface: the diffuse part of light from a particular source travels in a coherent beam toward O and then is scattered equally in all directions by diffuse reflectance from the surface of O. Therefore, the direction of the light source does matter in case of diffuse reflectance, but not that of the viewer. Practically then, the diffuse component models the "soft" part of the light with little focus, e.g., that reflected off polished wood or silky fabric.

The total diffuse component of light reflected from O is the sum of the reflectances of the diffuse parts from each source:

$$\text{diffuse reflectance from } O = \sum(\text{reflectance of diffuse part from each light source})$$

Specular: Specular reflectance models the shininess of the surface: the specular part of light from a particular source travels in a coherent beam toward O and then is reflected in mirror-like manner, again in a coherent beam, by specular reflectance from the surface of O. Both the direction of the light source and the viewer matter in the case of specular reflectance. Specular light is "hard" light with a focus, e.g., that from a beam bouncing off a polished metal surface.

The total specular component of the light reflected from O is the sum of the reflectances of the specular parts from each source:

$$\text{specular reflectance from } O = \sum(\text{reflectance of specular part from each light source})$$

Remark 11.1. Because specular reflection is mirror-like, while the ambient and diffuse reflections are due to scattering from the surface of the object, the color of specularly reflected light depends primarily on that of the source itself, while those of the ambient and diffusely reflected on the native color of the object, as well as the light source.

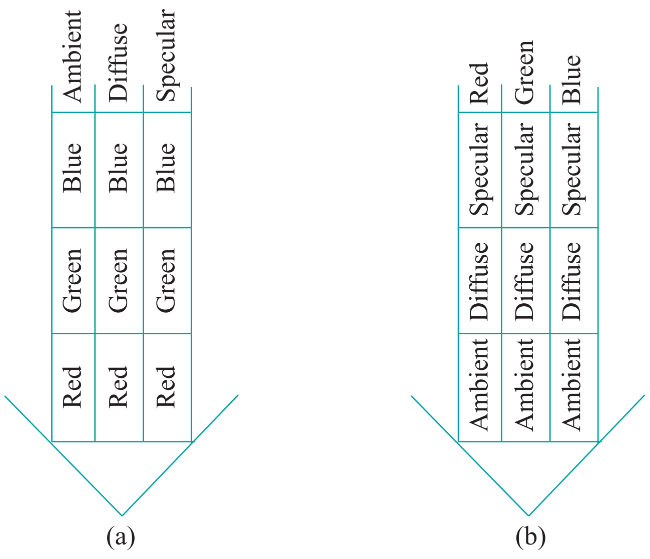


Figure 11.10: Orthogonal splitting of light: (a) Reflectance followed by color (b) Color followed by reflectance.

Important: The split of light into the three components of ambient, diffuse and specular according to reflectance is *independent* of the split into the primary color components of red, green and blue, in that each of the ambient, diffuse and specular components has RGB subcomponents and all nine subcomponents can be independently set. Or, one can equivalently say that each of RGB has ambient, diffuse and specular subcomponents which can all be independently set. In other words, you can think of light as being split as in Figure 11.10(a) or Figure 11.10(b) – it does not matter. Yet another way this is often phrased is by saying that color and reflectance splits are *orthogonal*.

A final component of light emerging from O is not reflected.

Emissive: The emissive component of light from an object O is that which is "manufactured" at O and unrelated to external light sources or the global ambient light. An example of an emissive object would be a lamp or the headlight of an automobile.

It is extremely important to keep in mind that, in OpenGL implementations, emissive light is perceived *only* by the viewer and does *not* illuminate other objects – it does *not* make O a light source for the rest of the environment.

11.2.2 Specifying Light and Material Values

OpenGL allows several light sources to be specified – the exact number depending on the implementation. For each of the N light sources L^i, $0 \le i \le N-1$, the RGB intensities of each of its ambient, diffuse and specular components can be set to between 0 and 1, for nine values altogether per light source. These are written, typically, in a 3×3 *light properties matrix*:

$$\begin{bmatrix} L^i_{amb,\,R} & L^i_{amb,\,G} & L^i_{amb,\,B} \\ L^i_{dif,\,R} & L^i_{dif,\,G} & L^i_{dif,\,B} \\ L^i_{spec,\,R} & L^i_{spec,\,G} & L^i_{spec,\,B} \end{bmatrix} \tag{11.2}$$

Similarly, for each object O or, more precisely, each vertex V of O, one can set *scaling factors* between 0 and 1 to determine how much of each component of the incident light is reflected, for again nine values, contained in a 3×3 *material properties matrix*:

$$\begin{bmatrix} V_{amb,\,R} & V_{amb,\,G} & V_{amb,\,B} \\ V_{dif,\,R} & V_{dif,\,G} & V_{dif,\,B} \\ V_{spec,\,R} & V_{spec,\,G} & V_{spec,\,B} \end{bmatrix} \tag{11.3}$$

These so-called reflectance values represent the object's color: the higher one is, the more of the corresponding incoming light is reflected, and the more of that color the object appears to be.

The RGB values of the global ambient light are contained in a 3-vector called the *global ambient light vector*:

$$[globAmb_R \quad globAmb_G \quad globAmb_B] \tag{11.4}$$

The RGB values of the emissive light from a vertex V is a 3-vector called the *emissive light vector*:

$$[V_{emit,\,R} \quad V_{emit,\,G} \quad V_{emit,\,B}] \tag{11.5}$$

11.2.3 Calculating the Reflected Light

We come now to calculating each component of the reflected light.

Ambient

Calculating the ambient component emerging from a vertex V owing to a particular light source consists simply of scaling the light's ambient intensity by V's ambient reflectance. If the original intensity of the ambient light of some primary color from a source L (or the global ambient) is I, then that of its reflection from the surface at V is

$$I * \textit{material ambient scaling factor} \tag{11.6}$$

The *material ambient scaling factor* is the fraction of the incident ambient light that the material reflects. It is nothing but the ambient reflectance value $V_{amb,\ X}$, where X may be R, G or B, in the first row of the material properties matrix. An example will clarify use of the equation.

Example 11.1. Say the intensities of the ambient light from source L are given by

$$L_{amb,\ R} = 0.4, \quad L_{amb,\ G} = 0.9, \quad L_{amb,\ B} = 0.2$$

and the ambient reflectances of V by

$$V_{amb,\ R} = 0.9, \quad V_{amb,\ G} = 0.9, \quad V_{amb,\ B} = 0.1$$

Then the part of the red light emanating from V owing to the L ambient is

$$L_{amb,\ R} * V_{amb,\ R} = 0.36$$

and the part of the green light emanating from V owing to the L ambient is

$$L_{amb,\ G} * V_{amb,\ G} = 0.81$$

and the part of the blue light emanating from V owing to the L ambient is

$$L_{amb,\ B} * V_{amb,\ B} = 0.02$$

Exercise 11.1. If the global ambient light vector is

$$[0.2\ 0.2\ 0.2]$$

and all the ambient reflectances of a vertex V are as in the preceding example, calculate the parts of the RGB light emanating from V owing to the global ambient.

Diffuse

Calculation of the diffuse component of the light reflected from V is more complex than that of the ambient as, not only must the incident light be scaled by the reflectance at V, but its direction taken into account as well. The latter is done by measuring the angle between the direction of the light source and the normal to the surface at V.

Remark 11.2. A line l is *normal* to a surface s at the point P if it is perpendicular to the tangent plane p of s at P. Any non-zero vector n parallel to l is a *normal vector* to s at P. See Figure 11.11. (Think intuitively of the tangent plane as a hard board pressed to touch s at P.)

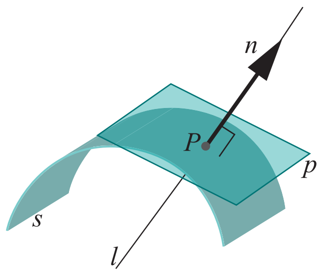


Figure 11.11: A normal vector n to a surface s at P lies along the normal line l there and is perpendicular to the tangent plane p at P.

The light source L is modeled as a point. Further, the surface of the object O around the illuminated vertex V is assumed flat; in fact, it's taken to coincide with its own tangent plane at V. See Figure 11.12(a). Diffuse light is reflected in all directions from V.

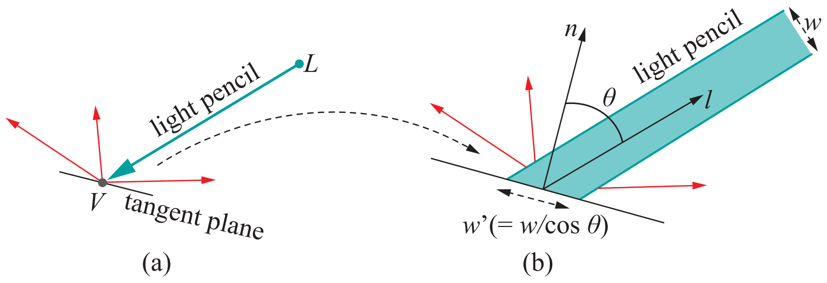


Figure 11.12: Calculating the diffuse component: (a) A light pencil from a point source L hits the surface, represented by it tangent plane at V(b) A blow-up of the pencil showing the normal vector n and the light direction vector l.

One observes from the blow-up in Figure 11.12(b) that a pencil of light of cross-sectional width w from L illuminates an area of width w', which is, typically, greater than w.

In this figure, θ is the angle between a direction vector l from V to the light source L, called the *light direction vector* and an outward normal vector n at V. The angle θ is called the *angle of incidence* of the light. We ask the reader to show next, by elementary trigonometry in Figure 11.12(b), that the width $w' = w/\cos\theta$.

Exercise 11.2. Verify the claim just made about the width of the area illuminated by a light of width w being $w' = w/\cos\theta$.

Since the area illuminated is greater by a factor of $1/\cos\theta$ than the cross-sectional area of the light pencil, the intensity of the light is diminished by a factor of $1/\cos\theta$ from I to $1/(1/\cos\theta) * I = \cos\theta * I$. Accordingly, if the original intensity of the diffuse light of some primary color emanating from the light source L is I, then that of its reflection from the surface of O at V is

$$\cos\theta * I * \textit{material diffuse scaling factor} \tag{11.7}$$

The *material diffuse scaling factor*, given by the values $V_{dif,\ X}$, where X is R, G or B, in the second row of material properties matrix, determines the fraction of the incident diffuse light the material reflects.

Example 11.2. Say the intensities of the diffuse light from source L are given by

$$L_{dif,\ R} = 0.3, \quad L_{dif,\ G} = 1.0, \quad L_{dif,\ B} = 1.0$$

and the diffuse reflectances of a vertex V by

$$V_{dif,\ R} = 0.8, \quad V_{dif,\ G} = 1.0, \quad V_{dif,\ B} = 0.8$$

and that the angle θ of incidence at V is $60°$.

Then the part of the red light emanating from V owing to the L diffuse is

$$\cos\theta * L_{dif,\ R} * V_{dif,\ R} = 0.5 * 0.3 * 0.8 = 0.12$$

Likewise, the part of the green light emanating from V owing to the L diffuse is

$$\cos\theta * L_{dif,\ G} * V_{dif,\ G} = 0.5$$

and the part of the blue light emanating from V owing to the L diffuse is

$$\cos\theta * L_{dif,\ B} * V_{dif,\ B} = 0.4$$

Remark 11.3. The relationship that the intensity of the reflected lighted varies as the cosine of the angle of incidence is known as *Lambert's law*. It is Lambert's law which explains why early mornings and late evenings, when the sun is lower in the sky, are cooler and darker than mid-day.

Specular

For specular light, as in the case of diffuse light, the light source L is modeled as a point, and so too the eye E. And, again, the surface of O is identified with its tangent plane at the illuminated vertex V. An outward normal vector to the surface of O at V is n. Let s be a vector, call it a *halfway vector*, which bisects the angle between a light direction vector l from V toward the light source L and an eye direction vector e from V toward the eye E. See Figure 11.13(a) (ignore r for now).

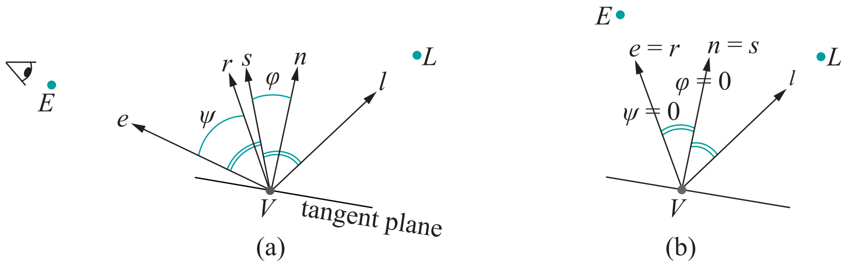


Figure 11.13: Calculating the specular component: (a) The light direction vector l, eye direction vector e, halfway vector s, normal vector n and reflection vector r (b) The special case when reflection is in the direction of the eye. (Double arcs indicate equal angles.)

We'll next state a relationship between the intensity of the reflected specular light and that of the incident which may seem unintuitive at first, but motivation will soon be apparent:

If the intensity of the specular light of some primary color emanating from the light source L is I, then that of its reflection from the surface of O at V is

$$\cos^f \phi * I * \textit{material specular scaling factor} \tag{11.8}$$

where ϕ is the angle between halfway vector s and the normal vector n, $f \geq 0$ is a scalar, called the *shininess exponent* and the *material specular scaling factor*, a value read from the material properties matrix, determines the fraction of the incident specular light the material reflects.

Here's what's happening. If the surface of O is *perfectly* mirror-like, then a ray of light from L to V reflects according to the laws of reflection, which say that the normal to O at V, the incident ray and the reflected ray all lie on the same plane and, moreover, that the incident ray and the reflected ray make the same angle with the normal. In this particular case, if the eye E is located in the direction of reflection, given by the *reflection vector* r, then it perceives all the incident light, if not no light at all.

Say ψ is the angle the reflection vector makes with the eye direction vector e, as in Figure 11.13(a). Figure 11.13(b) shows a particular case of the general Figure 11.13(a), where the eye is actually situated in the direction of reflection, so that $\psi = 0$. Observe, in this case, that the halfway vector is aligned with the normal, in other words, $\phi = 0$ as well.

Most real surfaces, however, are not perfectly mirror-like and do not reflect light only along the direction of reflection, but, rather, spread it *about* that direction with an intensity which diminishes with increasing angle. In other word, maximum intensity is perceived by the viewer in a configuration as in Figure 11.13(b); nevertheless, even in a general configuration as in Figure 11.13(a), the eye receives light, though, with intensity inversely related to ψ.

This suggests that the intensity of specular reflection be modeled by the formula

$$\textit{angular attenuation factor} * I * \textit{material specular scaling factor} \qquad (11.9)$$

where the *angular attenuation factor* is, in fact, the factor in inverse relationship with ψ.

Phong suggested the angular attenuation factor $\cos^f \psi$, where the exponent f is larger the more mirror-like (i.e., shiny) the surface is. His considerations were empirical rather than based on actual physics. In particular, $\cos \psi$ is a function of ψ which is at its maximum of 1 when $\psi = 0$ and drops off as ψ increases, behavior expected of the angular attenuation factor. The function $\cos^f \psi$ also shows the same behavior, but more markedly, as f increases. In particular, the larger the value of f the more rapid the drop from the value of 1 as ψ increases from 0. See Figure 11.14. Intuitively, the shinier the surface the more rapidly the light diminishes away from the direction of reflection.

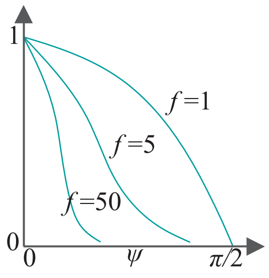


Figure 11.14: Graphs of $\cos^f \psi$ for different values of f (not exact plots).

The angle ψ is often replaced by ϕ, the angle between the halfway vector s and the normal vector n, because ϕ is easier to compute, and because it is legitimate to do so given the following linear relation between the two.

Example 11.3. Show that $\psi = 2\phi$.

Answer: Label the angles from the tangent plane to the light direction, the reflection and eye direction vectors θ_1, θ_2 and θ_3, respectively, as in Figure 11.15.

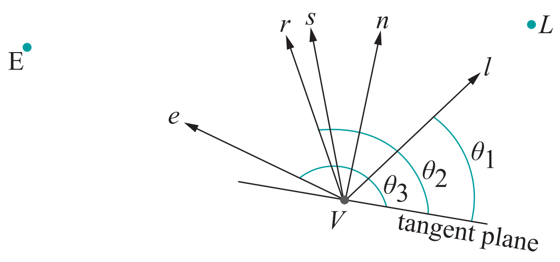


Figure 11.15: Proving that $\psi = 2\phi$.

The angle to the halfway vector, then, is $(\theta_1+\theta_3)/2$, implying that the angle between the halfway vector and the normal is $\phi = (\theta_1+\theta_3)/2 - \pi/2$.

The angle between the light vector and the normal, which is $\pi/2 - \theta_1$, is the same as the angle between the normal and the reflection vector, by laws of reflection. Therefore, $\theta_2 = \pi/2 + (\pi/2 - \theta_1) = \pi - \theta_1$. This, then, implies that the angle between the eye direction vector and the reflection vector is $\psi = \theta_3 - \theta_2 = \theta_3 - (\pi - \theta_1) = \theta_1 + \theta_3 - \pi$. That $\psi = 2\phi$ now follows.

Given the relationship between ϕ and ψ contained in the preceding example, substituting $\cos^f \phi$ for $\cos^f \psi$ as the angular attenuation factor in Equation (11.9) makes no qualitative difference. The result of the substitution, in fact, is Equation (11.8), which is now fully justified.

Example 11.4. Give a formula for the halfway vector s in terms of the light direction vector l and the eye direction vector e from V, which are, of course, the two vectors that s bisects. Assume that both l and e are of unit length. Give s as a unit vector as well.

Answer: See Figure 11.16, where $l = \overrightarrow{OA}$, $e = \overrightarrow{OB}$, and where $l+e$ is drawn with the help of the parallelogram law of addition of vectors. Since $|l| = |e| = 1$, all four sides of the parallelogram $OACB$ are of unit length as well. A consequence is that corresponding sides of the triangles OAC and OBC are of equal lengths. The two triangles are, therefore, congruent, so $\angle AOC = \angle BOC$. One concludes that the vector $l+e$ bisects l and e.

Accordingly, the unit halfway vector

$$s = \frac{l+e}{|l+e|} \quad \text{(provided that } l+e \text{ is not the zero vector)}$$

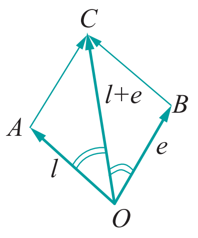


Figure 11.16: The vector $l = e$ bisects l and e.

Remark 11.4. When a vector u is used to represent a direction, so that its magnitude is not of importance, it is often convenient to scale it to unit length, a step called *normalizing* u. Normalization of a non-zero vector u (note that a vector representing a direction cannot be zero) consists simply of dividing it by its length, in other words, replacing u by $u/|u|$.

Exercise 11.3. Give a simple formula, in an OpenGL setting, for the eye direction vector from a vertex V whose position vector is v. Accordingly, rewrite the formula for the halfway vector of the preceding example in terms of l and v.

Example 11.5. Say the intensities of the specular light from source L are given by

$$L_{spec,\,R} = 1.0, \quad L_{spec,\,G} = 1.0, \quad L_{spec,\,B} = 1.0$$

and the specular reflectances of a vertex V by

$$V_{spec,\,R} = 0.0, \quad V_{spec,\,G} = 1.0, \quad V_{spec,\,B} = 0.6$$

and that the angle ϕ between the halfway vector and the outward normal vector at V is $60°$ and that the shininess exponent is 2.0.

Then the part of the red light emanating from V owing to the L specular is

$$\cos^f \phi * L_{spec,\ R} * V_{spec,\ R} = 0.25 * 1.0 * 0.0 = 0.0$$

Likewise, the part of the green light emanating from V owing to the L specular is

$$\cos^f \phi * L_{spec,\ G} * V_{spec,\ G} = 0.25$$

and the part of the blue light emanating from V owing to the L specular is

$$\cos^f \phi * L_{spec,\ B} * V_{spec,\ B} = 0.15$$

It's interesting that calculation of the reflected light never actually required determination of the reflection vector r itself. It's not hard though to find r, as we see next.

Exercise 11.4. Suppose that n is the unit (outward) normal vector and l the unit light direction vector at a vertex V. Prove that the unit vector r in the direction of reflection is given by the equation

$$r = 2(n \cdot l)n - l$$

Part answer: According to the laws of reflection we have to verify that r lies on the plane of l and n and makes the same angle with n as l. We must also prove that r is of unit length.

That r lies on the plane of l and n follows from its formula above, because of the linear dependence of r on l and n. Now

$$|r|^2 = r \cdot r = (2(n \cdot l)n - l) \cdot (2(n \cdot l)n - l) = 4(n \cdot l)^2 - 4(n \cdot l)^2 + l \cdot l = |l|^2 = 1$$

proving that r indeed is a unit vector.

We'll leave the reader to prove that r makes the same angle with n as l by computing its dot product with n.

11.2.4 First Lighting Equation

Our formulae from the last section straightforwardly combine into a single so-called *lighting equation*. Assume that we are given the values of the lighting properties matrix (11.2) for each light source L^i, $0 \leq i \leq N-1$, the material properties matrix (11.3) for the vertex V, the global ambient light vector (11.4), as well as the emissive light vector (11.5) at V. Further, denote the normalized light direction and halfway vectors corresponding to light source L^i at vertex V by l^i and s^i, respectively. Denote the normalized outward surface normal vector at V by n and its shininess exponent by f.

Here then is the lighting equation giving the color intensity V_X at V, where X may be any of RGB:

$$\begin{aligned} V_X \;=\; & V_{emit,\,X} \;+ \\ & globAmb_X * V_{amb,\,X} \;+ \\ & \sum_{i=0}^{N-1} \Big(L^i_{amb,\,X} * V_{amb,\,X} \;+ \\ & \qquad \max\{l^i \cdot n\,,\,0\} * L^i_{dif,\,X} * V_{dif,\,X} \;+ \\ & \qquad (\max\{s^i \cdot n\,,\,0\})^f * L^i_{spec,\,X} * V_{spec,\,X} \Big) \end{aligned} \tag{11.10}$$

Note: The dot product of two unit vectors gives the cosine of the angle between them.

Note: If the RHS sums to more than 1, for any of X equal to R, G or B, then it is clamped to 1.

The lighting equation simply collects the components we have already discussed separately. The first summand on the RHS is the emissive component, while the second the global ambient scaled by the ambient reflectance. The third summand is a summation over the n light sources of

(a) The incident ambient component scaled by the ambient reflectance (Equation (11.6)).

(b) The incident diffuse component scaled by the diffuse reflectance and the cosine of the incident angle (Equation (11.7)).

(c) The incident specular component scaled by the specular reflectance and the angular attenuation factor (Equation (11.8)).

The reason for the $\max\{*\,,0\}$ terms is to not allow a negative multiplier, which would imply the physically impossible phenomenon of light being subtracted. For example, $l^i \cdot n$ is negative when the angle between l^i and n is greater than $\pi/2$, which means that the light source L^i is behind the surface on which V is located, contributing zero light, rather than negative light.

Equation (11.10) is actually a first draft. The final lighting equation of OpenGL, which we'll see soon, enhances it by taking into account the attenuation of light over distance, as well as the spotlight effect, where light from a source emerges as a cone, rather than in all directions.

Exercise 11.5. There are two light sources L^0 and L^1, the respective values of whose lighting properties matrices are

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.7 & 0.1 & 0.1 \\ 0.7 & 0.1 & 0.1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.1 & 0.7 & 0.1 \\ 0.1 & 0.7 & 0.1 \end{bmatrix}$$

The material properties matrix at a vertex V is

$$\begin{bmatrix} 0.1 & 0.8 & 0.9 \\ 0.1 & 0.8 & 0.9 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

Furthermore, the shininess exponent of the surface at V is 2.0, there is no emission from V, and the unit outward normal vector at V is

$$[0.0\ 1.0\ 0.0]^T$$

The position vectors of L^0, L^1, V and the eye are, respectively,

$$[0.0\ 5.0\ 5.0]^T, \quad [5.0\ 5.0\ 5.0]^T, \quad [0.0\ 0.0\ 5.0]^T \text{ and } [0.0\ 0.0\ 0.0]^T$$

The global ambient light vector is

$$[0.1\ 0.1\ 0.1]^T$$

Compute the color vector at V using the lighting equation (11.10).

Remark 11.5. It is important to realize that Phong's lighting model is *local*: the color at each vertex V depends only on the interaction between the external light properties and the material properties at V *itself*. No account is taken of whether V is obscured from a light source by another object (shadows), or of light that strikes V not directly from a light source but having bounced off other objects (reflection and secondary lighting). Colloquially, object-object light interaction is not considered, only light-object. We will discuss two global lighting models, ray tracing and radiosity, where shadows, reflections and other secondary effects are captured, in a later chapter.

11.3 OpenGL Light and Material Properties

The mapping from Phong's lighting model to OpenGL syntax is pretty much one-to-one. For each light source the user defines the values in the lighting properties matrix (11.2), as also the values in the material properties matrix (11.3) for each vertex. The global ambient vector (11.4) is user-defined as well. The user, too, defines the shininess exponent f, the emission color vector (11.5) and, very importantly, the normal vector at each vertex.

If you are beginning to worry that that's a lot of values to specify to light a scene, don't! Remember that OpenGL is a state machine, so material properties – which are state variables – persist in their current setting until explicitly changed, making it convenient for the programmer to apply the same properties to all vertices of a single object. Moreover, OpenGL has sensible defaults for values the programmer doesn't care to define.

Time to look at code.

Experiment 11.1. Run again `sphereInBox1.cpp`, which we ran the first time in Section 9.4. Press the up-down arrow keys to open or close the box. Figure 11.17 is a screenshot of the box partly open. We'll use this program as a running example to explain much of the OpenGL lighting and material color syntax. **End**

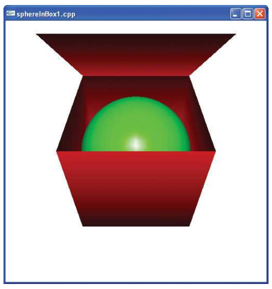


Figure 11.17: Screenshot of `sphereInBox1.cpp`.

11.3.1 Light Properties

Properties of light sources are set by statements of the form:

`glLight*(`*light*`, `*parameter*`, `*value*`)`

where *light* is the label of the light (viz., `GL_LIGHT0`, `GL_LIGHT1`, ...) and its particular *parameter* set to *value*.

The properties of the single light source of `sphereInBox1.cpp` are specified by the following statements in the `setup()` routine:

```
glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmb);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDifAndSpec);
glLightfv(GL_LIGHT0, GL_SPECULAR, lightDifAndSpec);
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
```

Typically, the diffuse and specular color vectors, i.e., the values of the `GL_DIFFUSE` and `GL_SPECULAR` parameters, respectively, are set identically to values perceived as the actual color of the light source. So, that of `sphereInBox1.cpp` is a bright white.

It's simplifying, as well, to consolidate all light source ambients – their `GL_AMBIENT` values – into the global ambient; in other words, set light source ambient colors all to 0.0 and adjust the one global ambient light vector. We follow this approach in `sphereInBox1.cpp`, as in all our lit programs. The fourth component, the alpha value, of each of the three color vectors – ambient, diffuse and specular – should always be 1.0 for a light source.

The value $\{x, y, z, w\}$ of `GL_POSITION` specifies the location $[x\ y\ z\ w]^T$ of the light source in homogeneous coordinates. If $w \neq 0$ then the light source is said to be positional and is located at world coordinates

$$[x/w \quad y/w \quad z/w]^T$$

The single positional light source of `sphereInBox1.cpp` is at $[0.0\ 1.5\ 3.0]^T$, which is just above and some ways in front of the box. We'll discuss what happens if $w = 0$ in Section 11.5 when we discuss directional light sources.

Note that *no visible object* is created by OpenGL at the location of a light source! This location is simply a point used for the purpose of lighting calculation. If you want the light to appear to be from a lamp or car headlight or such object you'll have to model the object and position it yourself.

Global ambient light in `sphereInBox1.cpp` is set with the statement

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, globAmb);
```

in the `setup()` routine, where the second parameter points to the global ambient vector. Don't forget that lighting calculation is enabled with the call `glEnable(GL_LIGHTING)` and individual lights with calls to `glEnable(GL_LIGHT`*i*`)`.

Exercise 11.6. Show that nothing, in fact, is lost according to the first lighting equation (11.10) by setting all light source ambient colors to 0.0. In particular, prove that, however the light source ambients are initially set, they can all be reset to 0.0 and the global ambient adjusted accordingly so that the color computed at each vertex by (11.10) remains unchanged.

11.3.2 Material Properties

Material properties at a vertex are set by statements of the form:

```
glMaterial*(face, parameter, value)
```

where the *parameter* of *face* is set to *value*. The value of face can be `GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK` for both faces.

Material properties of (each vertex of) the box of `sphereInBox1.cpp` are specified by the following statements in the `drawScene()` routine:

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, matAmbAndDif1);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, matSpec);
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, matShine);
```

Typically, the ambient and diffuse color vectors are set identically to values perceived as an object's native color. OpenGL makes it convenient to do so via the `GL_AMBIENT_AND_DIFFUSE` parameter; however, they can be set separately as well using `GL_AMBIENT` and `GL_DIFFUSE`. As specular light is obtained from reflection from the light source, it's reasonable to set an object's `GL_SPECULAR` value either to white $\{1.0, 1.0, 1.0, 1.0\}$, fully reflecting the incident specular light, or a shade of gray $\{\gamma, \gamma, \gamma, 1.0\}$, *equally* scaling each color component. Ignore, for now, the fourth, or alpha, component of the material color vectors, all currently set to 1 – the alpha value pertains to blending, which is discussed in a later chapter.

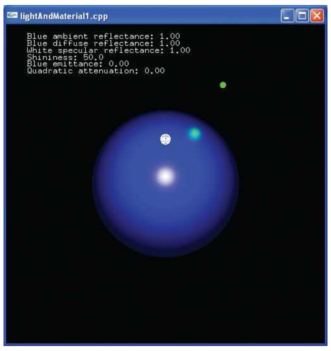
The value of `GL_SHININESS` is, of course, the shininess exponent f of the first lighting equation (11.10). Its value must be in the range $[0.0, 128.0]$. The default is 0.0, which causes no angular attenuation of specular reflectance.

The emissive color at a vertex can be set using the `GL_EMISSION` parameter, but we choose to go with the default of $\{0.0, 0.0, 0.0, 1.0\}$, in other words, no emission, for each vertex in `sphereInBox1.cpp`.

Exercise 11.7. (Programming) What are the material specular values and the shininess exponent of the sphere of `sphereInBox1.cpp`?

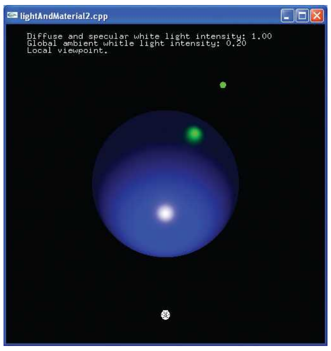
11.3.3 Experimenting with Properties

The two programs `lightAndMaterial1.cpp` and `lightAndMaterial2.cpp` allow the user to experiment with various material and light properties. Both show a blue ball lit by two lights, one white and one green, whose positions are indicated by small wire spheres. Figure 11.18 shows screenshots of both the programs.



```
Blue ambient reflectance: 1.00
Blue diffuse reflectance: 1.00
White specular reflectance: 1.00
Shininess: 50.0
Blue emittance: 0.00
Quadratic attenuation: 0.00
```

(a)



```
Diffuse and specular white light intensity: 1.00
Global ambient whitle light intensity: 0.20
Local viewpoint.
```

(b)

Figure 11.18: Screenshots of (a) `lightAndMaterial1.cpp` (b) `lightAndMaterial2.cpp`.

Using the first program one can change material properties of the blue ball, as well as move it. The second program, on the other hand, allows properties of the white light to be controlled, as also of the global ambient, and enables the user to rotate the white light. Text messages show property values. Let's take a quick tour of the two before experimenting with properties.

Experiment 11.2. Run `lightAndMaterial1.cpp`.

The ball's current ambient and diffuse reflectances are identically set to a maximum blue of $\{0.0, 0.0, 1.0, 1.0\}$, its specular reflectance to the highest gray level $\{1.0, 1.0, 1.0, 1.0\}$ (i.e., white), shininess to 50.0 and emission to zero $\{0.0, 0.0, 0.0, 1.0\}$.

Press 'a/A' to decrease/increase the ball's blue **A**mbient and diffuse reflectance. Pressing 's/S' decreases/increases the gray level of its **S**pecular reflectance. Pressing 'h/H' decreases/increases its s**H**ininess, while pressing 'e/E' decreases/increases the blue component of the ball's **E**mission.

The program has further functionalities which we'll explain as they become relevant. End

Experiment 11.3. Run `lightAndMaterial2.cpp`.

The white light's current diffuse and specular are identically set to a maximum of $\{1.0, 1.0, 1.0, 1.0\}$ and it gives off zero ambient light. The green light's attributes are fixed at a maximum diffuse and specular of $\{0.0, 1.0, 0.0, 1.0\}$, again with zero ambient. The global ambient is a low intensity gray at $\{0.2, 0.2, 0.2, 1.0\}$.

Press 'w' or 'W' to toggle the **W**hite light off and on. Pressing 'g' or 'G' toggles the **G**reen light off and on. Press 'd/D' to decrease/increase the gray level of the white light's **D**iffuse and specular intensity (the ambient intensity never changes from zero). Pressing 'm/M' decreases/increases the gray intensity of the global a**M**bient. Rotate the white light about the ball by pressing the arrow keys.

The program has more functionality too which we'll need later. **End**

Experiment 11.4. Run `lightAndMaterial1.cpp`.

Reduce the specular reflectance of the ball. Both the white and green highlights begin to disappear, as it's the specular components of the reflected lights which appear as specular highlights. **End**

Exercise 11.8. (Programming) The specular highlight is sharpened or blunted, respectively, by increasing or decreasing the shininess exponent. Why?

Hint: The higher the shininess exponent the more rapidly the specular light diminishes as the vertex normals turn away from the eye direction (recall the definition of the angular attenuation factor in Section 11.2.3).

Experiment 11.5. Restore the original values of `lightAndMaterial1.cpp`.

Reduce the diffuse reflectance gradually to zero. The ball starts to lose its roundness until it looks flat as a disc. The reason for this is that the ambient intensity, which does not depend on eye or light direction, is uniform across vertices of the ball and cannot, therefore, provide the sense of depth that obtains from a contrast in color values across the surface. Diffuse light, on the other hand, which varies across the surface depending on light direction, can provide an illusion of depth.

Even though there is a specular highlight, sensitive to both eye and light direction, it's too localized to provide much contrast. Reducing the shininess spreads the highlight but the effect is not a realistic perception of depth.
Moral: Diffusive reflectance lends three-dimensionality. **End**

Experiment 11.6. Restore the original values of `lightAndMaterial1.cpp`.

Now reduce the ambient reflectance gradually to zero. The ball seems to shrink! This is because the vertex normals turn away from the viewer at the now hidden ends of the ball, scaling down the diffuse reflectance there (recall the $\cos\theta$ term in the diffusive reflectance equation (11.7)). The result is that, with no ambient reflectance to offset the reduction in diffuse, the ends of the ball are dark.
Moral: Ambient reflectance provides a level of uniform lighting over a surface.
End

Experiment 11.7. Restore the original values of `lightAndMaterial1.cpp`.

Reduce both the ambient and diffuse reflectances to nearly zero. It's like the cat disappearing, leaving only its grin! Specular light is clearly for highlights and not much else. **End**

Exercise 11.9. (Programming) Restore the original values of `lightAndMaterial1.cpp`.

Reduce all three of the ball's diffuse, ambient and specular reflectances and raise its emissive light intensity. It does appear to glow but also appears flat. Why?

Experiment 11.8. Run `lightAndMaterial1.cpp` with its original values.

With it's current high ambient, diffuse and specular reflectances the ball looks a shiny plastic. Reducing the ambient and diffuse reflectances makes for a heavier and less plastic appearance. Restoring the ambient and diffuse to higher values, but reducing the specular reflectance makes it a less shiny plastic. Low values for all three of ambient, diffuse and specular reflectances give the ball a somewhat wooden appearance. **End**

Experiment 11.9. Run `lightAndMaterial2.cpp`.

Reduce the white light's diffuse and specular intensity to 0. The ball becomes a flat dull blue disc with a green highlight. This is because the ball's ambient (and diffuse) is blue and cannot reflect the green light's diffuse component, losing thereby three-dimensionality.

Raising the white global ambient brightens the ball, but it still looks flat in the absence of diffusive light. **End**

Exercise 11.10. (Programming) When the white light is switched off in `lightAndMaterial2.cpp`, the only evidence of green on the ball is the specular highlight; moreover, if the ambient is tamped down as well then the ball begins to disappear altogether.

However, this is not so in the opposite situation, when the white light is switched on and the green off – a sector of the ball is clearly visible no matter how low the ambient. Why?

Experiment 11.10. Nate Robins has a bunch of great tutorial programs at the site [96]. This is a good time to run his lightmaterial tutorial, which allows the user to control a set of parameters as well. **End**

11.3.4 Color Material Mode

Remember `glColor*()` which we used to set color in the dark days before there were light sources? Now that we have light and `glMaterial*()` allows us to set all sorts of material properties, it seems there's no use any more for `glColor*()`. Well, it turns out that the good folk who designed OpenGL found a way to keep it on the payroll.

Here's how. Suppose you're in the not uncommon situation coloring a scene where only a particular color attribute, say the ambient and diffuse reflectances of the front faces, changes from one object to the next, other attributes remaining constant. What you can do in this case, instead of repeatedly calling `glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE,` *value*`)`, is to:

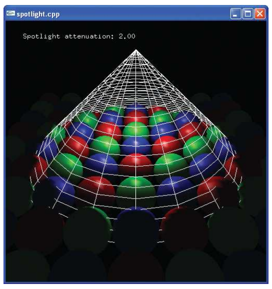
1. Enable the so-called *color material mode* with a call to `glEnable(GL_COLOR_MATERIAL)`.

2. Call `glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE)`, which tells OpenGL to use the current color, set by `glColor*()`, to determine the front-face ambient and diffuse color values.

   Generally, the `glColorMaterial()` call can be of the form `glColorMaterial(`*face*`,` *parameter*`)` where *face* can be `GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK`, and *parameter* one of `GL_AMBIENT`, `GL_DIFFUSE`, `GL_AMBIENT_AND_DIFFUSE`, `GL_SPECULAR` or `GL_EMISSION`.

3. Make a call to `glColor*()` to set the front-face ambient and diffuse color from one object to the next.

This method may, in fact, be more efficient with certain implementations of OpenGL, not to mention the convenience of not having to change a programming habit if one is used to coloring with `glColor*()`.


spotlight.cpp

```
Spotlight attenuation: 2.00
```

Figure 11.19: Screenshot of `spotLight.cpp`.

Experiment 11.11. Run `spotlight.cpp`. The program is primarily to demonstrate spotlighting, the topic of a forthcoming section. Nevertheless, press the page-up key to see a multi-colored array of spheres. Figure 11.19 is a screenshot.

Currently, the point of interest in the program is the invocation of color material mode for the front-face ambient and diffuse reflectances by means of the last two statements in the initialization routine, viz.,

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

and subsequent coloring of the spheres in the drawing routine by `glColor4f()` statements. **End**

11.4 OpenGL Lighting Model

The so-called OpenGL *lighting model* sets certain environmental parameters. The terminology, even though used in the red book, is somewhat unfortunate as it may suggest laws of interaction between light and objects, or a relation with Phong's model – neither of which is true. The four parameters the OpenGL lighting model sets are the following:

1. The global ambient light with the statement

   `glLightModel*(GL_LIGHT_MODEL_AMBIENT,` *globAmb*`)`

   where *globAmb* is the global ambient light vector. This we've seen already.

2. Whether to use a local or infinite viewpoint for lighting calculation.

   See again the lighting equation (11.10). The halfway vector s^i at a vertex, one for each light source, is the unit vector bisecting the angle between the direction vector l^i to the light source L^i and the direction vector e to the eye.

   The OpenGL eye being fixed at the origin $[0\ 0\ 0]^T$, evidently $e = -V$, where V is the vertex's position vector, which changes from one to another. However, it simplifies lighting computation to keep e constant, particularly $e = [0\ 0\ 1]^T$, equivalent to *assuming* an eye that is infinitely far up the z-axis and so, effectively, in the same direction from every vertex. See Figure 11.20. This simplification, often, still gives adequately authentic lighting.

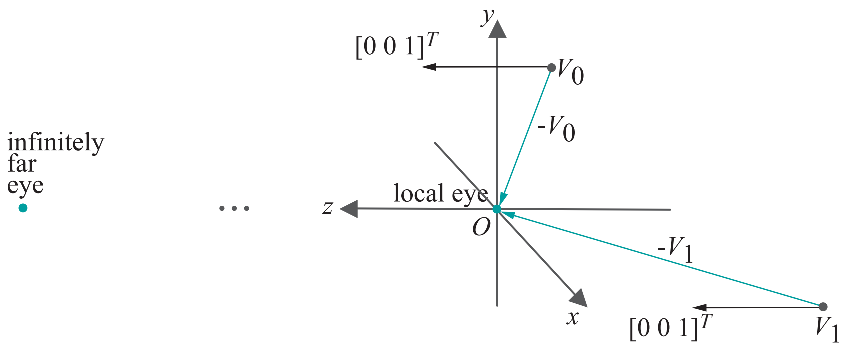


Figure 11.20: Local versus infinite viewpoint: the direction vector from each vertex toward the infinite viewpoint is black, while that toward the local viewpoint – i.e., the eye vector – is blue.

Remark 11.6. The direction vector l^i to the light source, too, changes from vertex to vertex if the source is a positional one, i.e., if $w \neq 0$ in the value $[x\ y\ z\ w]^T$ of the source's `GL_POSITION` parameter. Moreover, a simplification exactly similar to that of assuming an infinite viewpoint can be achieved, not by tweaking the OpenGL lighting model, but by making the light directional by setting $w = 0$. We'll discuss this in the next section.

The OpenGL default viewpoint, in fact, is infinite. For lighting calculation to be done using a local viewpoint instead – with the eye at the origin, that is – call

```
glLightModel*(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE)
```

which is what we do in the `setup()` routines of both `sphereIn-Box1.cpp` and `lightAndMaterial1.cpp`, while `lightAndMaterial2.cpp` provides an option. The local viewpoint is more realistic at the expense of greater computation.

Remark 11.7. The chosen light model viewpoint is used *only for lighting calculations*. The viewing frustum or box stays unchanged – therefore, in the case of a frustum, for example, we still *see the scene* from the eye at the origin.

Exercise 11.11. (Programming) Press 'l' or 'L' to toggle between the **L**ocal and the infinite viewpoint in `lightAndMaterial2.cpp`. The change seems to be only in the highlights, in other words, only the specular reflectances. Why?

3. Whether to enable two-sided lighting.

The OpenGL default is to perform lighting calculations for each polygon based on its specified `GL_FRONT` face parameter values and its specified vertex normals, regardless of if it is front or back facing. As the user likely sets material properties and normal values with the front faces of polygons in mind, results tend to be unrealistic for those whose back faces happen to be visible. So, when back faces might be visible, the command to use is

```
glLightModel*(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE)
```

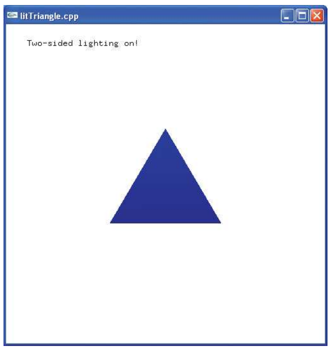
which causes OpenGL to

(a) use the `GL_BACK` (or `GL_FRONT_AND_BACK`) parameter values to color back-facing polygons, and

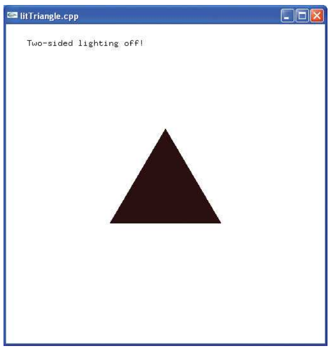
(b) *reverse* the specified vertex normal for back-facing polygons.

Experiment 11.12. Run `litTriangle.cpp`, which draws a single triangle, whose front is coded red and back blue, initially front-facing and lit two-sided. Press the left and right arrow keys to turn the triangle and space to toggle two-sided lighting on and off. See Figure 11.21 for screenshots.

Notice how the back face is dark when two-sided lighting is disabled – this is because the normals are pointing oppositely of the way they should be. **End**



(a)



(b)

Figure 11.21: Screenshots of `litTriangle.cpp` showing the back face with (a) two-sided lighting on (b) two-sided lighting off.

4. Whether to apply specular light before or after texturing.

   The following remarks will be more meaningful after the discussion of textures in the next chapter.

   The OpenGL default is to apply textures after all lighting calculations, which can cause specular highlights to be smothered. However, the command

   ```
   glLightModel*(GL_LIGHT_MODEL_COLOR_CONTROL,
                 GL_SEPARATE_SPECULAR_COLOR)
   ```

   makes OpenGL

   (a) separately produce two colors at each vertex: a primary color calculated from all incoming non-specular components and a secondary color from all incoming specular components,

   (b) combine only the primary color with texture color at the time of texture mapping and, finally,

   (c) add in the secondary color to the result of the previous step, which assures the specular highlights.

11.5 Directional Lights, Positional Lights and Attenuation of Intensity

Directional and Positional Light Sources

We know that the value of the `GL_POSITION` parameter of a light source L specifies its location $[x\ y\ z\ w]^T$ in homogeneous coordinates.

If $w \neq 0$, then the light source is called *positional*, or *local*, and located at world coordinates $[x/w \;\; y/w \;\; z/w]^T$. This is the kind of source we have used so far. If $w = 0$, then the light source is *directional* and assumed located at an infinite distance in the direction of $[x \; y \; z]^T$ from the origin. See Figure 11.22.

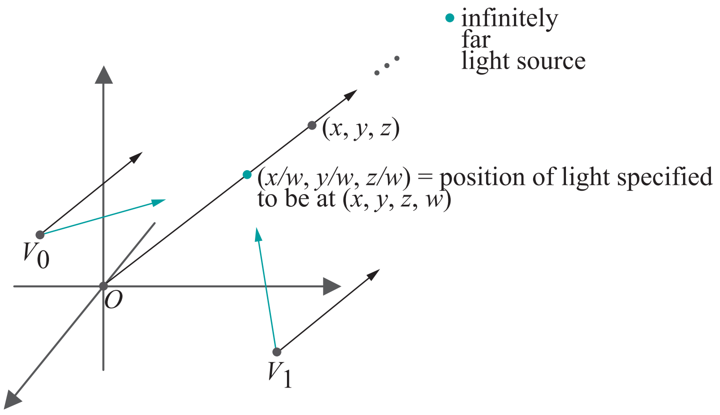

infinitely far light source
(x, y, z)
(x/w, y/w, z/w) = position of light specified to be at (x, y, z, w)
V0
O
V1

Figure 11.22: Directional versus positional light: the direction vector from each vertex toward the directional light is black and parallel to the direction of the directional light, while that toward the positional light is blue.

A positional light is located within the environment, e.g., a car headlight, while a directional light is far removed, e.g., the sun. From the point of view of lighting calculation, the difference is that the light direction vector l from a vertex V to the light source L depends on the coordinates of V if L is positional, while it is constant for all vertices if L is directional. Evidently, lighting calculation is cheaper for directional sources.

The default value for `GL_POSITION` is $[0\;0\;1\;0]^T$, which defines a directional light shining down from high up the z-axis.

Experiment 11.13. Press 'p' or 'P' to toggle between **P**ositional and directional light in `lightAndMaterial2.cpp`.

The white wire sphere indicates the positional light, while the white arrow the incoming directional light. **End**

Attenuation of Light

In the real world, the intensity of light from a source diminishes with distance from the source following an inverse square law. This phenomenon, called *distance attenuation*, can be modeled in OpenGL as well by a multiplicative *distance attenuation factor*

$$\frac{1}{k_c + k_l d + k_q d^2}$$

where d is the distance from the light source and k_c, k_l and k_d are the values of the light parameters `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` and `GL_QUADRATIC_ATTENUATION`, respectively. These values are set by statements of the form

```
glLightf(GL_LIGHTi, GL_CONSTANT_ATTENUATION, k_c);
glLightf(GL_LIGHTi, GL_LINEAR_ATTENUATION, k_l);
glLightf(GL_LIGHTi, GL_QUADRATIC_ATTENUATION, k_q);
```

The default values are $k_c = 1$ and $k_l = k_q = 0$, which imply no attenuation over distance at all. Attenuating the intensity of a directional light over distance is not meaningful as it's already infinitely far from every vertex; therefore, default values for the attenuation parameters cannot be changed for such a source.

Experiment 11.14. Run `lightAndMaterial1.cpp`. The current values of the constant, linear and quadratic attenuation parameters are 1, 0 and 0, respectively, so there's no attenuation. Press 't/T' to decrease/increase the quadratic a**T**tenuation parameter. Move the ball by pressing the up/down arrow keys to observe the effect of attenuation. **End**

11.6 Spotlights

The default for a light source is that it's *regular*, emitting light in all directions. This can be altered by turning it into a *spotlight*, in which case the emitted light is in the shape of a cone. Figures 11.23(a) and (b) show, respectively, plane sections of the light from both a regular and a spotlight.

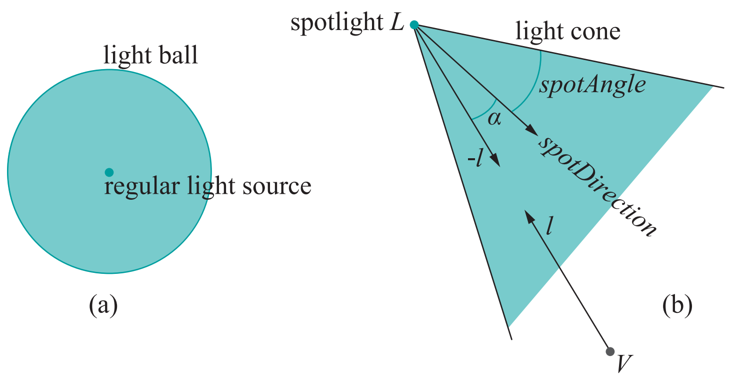


Figure 11.23: Sections of (a) Regular light (b) Spotlight. Shown for the spotlight are the light direction vector l from vertex V toward light source L, the inverse vector $-l$ from the light source toward the vertex, the direction vector *spotDirection* of the cone's axis and the half-angle *spotAngle* at its apex.

Experiment 11.15. Run `spotlight.cpp`, which shows a bright white spotlight illuminating a multi-colored array of spheres. A screenshot was shown earlier in Figure 11.19.

Press the page up/down arrows to increase/decrease the angle of the light cone. Press the arrow keys to move the spotlight. A white wire mesh is drawn along the light cone boundary. **End**

All statements pertaining to the spotlight properties of the single light source of `spotlight.cpp` are located in the `drawScene()` routine. The first step to turning a light source L into a spotlight is to specify the half-angle at the apex of the light cone, called the *cone angle*, with the command

`glLightf(GL_LIGHT0, GL_SPOT_CUTOFF,` *spotAngle*`)`

which, in fact, sets the cone angle to the value *spotAngle*. This should be between 0.0 and 90.0. The default is the special value of 180.0, meaning that L is not a spotlight, but a regular source emitting in all directions.

The next step is to specify the direction of the spotlight or, more specifically, that of the axis of its cone with a command

`glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION,` *spotDirection*`)`

which sets the axis in a direction parallel to the vector

$$spotDirection = [x\ y\ z]^T$$

The default value of `GL_SPOT_DIRECTION` is $[0\ 0\ -1]^T$, aiming the spotlight down the negative z-axis.

A final spotlight parameter is `GL_SPOT_EXPONENT`, whose value is called the *spotlight attenuation factor* and which controls the distribution of intensity through the light cone. If the value of `GL_SPOT_EXPONENT` is h and the angle between the axis of the light cone and the direction from source L toward vertex V is α, then the intensity of light at V is attenuated by the multiplicative factor $\cos^h \alpha$. This, of course, presumes that V lies within the light cone in the first place; if not, no light reaches V from L at all. Note that, as depicted in Figure 11.23, a vector from L toward V is, simply, $-l$, the negative of a light direction vector.

The motivation behind the spotlight attenuation factor is similar to that for the angular attenuation factor in the calculation of specular reflection in Equation (11.8) – so that the greater the value of h, the more rapidly the intensity of the spotlight attenuates away from the cone's axis. Equivalently, the greater h the more "concentrated" the spotlight. The default value of `GL_SPOT_EXPONENT` is 0, implying no attenuation at all.

Experiment 11.16. Run again `spotlight.cpp`. The current value of the spotlight's a**T**tenuation is 2.0, which can be decreased/increased by pressing 't/T'. Note the change in visibility of the balls near the cone boundary as the attenuation changes. **End**

Exercise 11.12. A spotlight should always be positional. Why?

For use in the upcoming final OpenGL light equation, let's write a single complete formula for a *spotlight attenuation factor* or, briefly, *saf*, at a vertex V, for a given light source L. Denote the unit vector along the spotlight axis – the normalized value of `GL_SPOT_DIRECTION` – by d and assume that l, the light direction vector from V, is normalized as well (see Figure 11.23). Then:

$$saf = \begin{cases} 1 & , \text{ if } \ spotAngle = 180^\circ \\ 0 & , \text{ if } \ -l \cdot d < \cos(spotAngle) \\ (-l \cdot d)^h & , \text{ otherwise} \end{cases} \qquad (11.11)$$

Here's how to parse the formula.

The first line is the case when L is not a spotlight, so there's no attenuation.

For the second line, recall that $-l$ is the unit vector from L toward V. Therefore, $-l \cdot d = \cos\alpha$, where α is the angle between the axis of the light cone and the direction of V from L. Now, if $\cos\alpha < \cos(spotAngle)$, then $\alpha > spotAngle$, which means that V lies outside the light cone and gets zero light. This explains the second line.

The third line, of course, gives the angular attenuation factor.

Exercise 11.13. Why isn't it necessary to write $(\max\{-l \cdot d, 0\})^h$, instead of $(-l \cdot d)^h$, in Equation (11.11) in a manner similar to the first lighting equation (11.10)?

Exercise 11.14. (Programming) In addition to the spotlight attenuation, light from a spotlight source can be distance attenuated as well. Additionally, allow distance attenuation to be controlled in `spotlight.cpp`. Add vertical motion capability to the light source to in order to accentuate the effect of distance attenuation. And while you're at it, why not make the light emerge from a well at the bottom of a flying saucer?!

11.7 OpenGL Lighting Equation

We now have the two additional pieces needed to enhance the first lighting equation (11.10) to the form that is, in fact, used by OpenGL to calculate RGB color intensities at a vertex V, namely, distance attenuation and spotlight attenuation. The enhancement is straightforward.

All symbols from the first lighting equation retain the same meaning. Additionally, d^i denotes the distance of V from the ith light source; k_c^i, k_l^i and k_q^i denote, respectively, the constant, linear and quadratic attenuation parameters for the ith light source; and saf^i is the spotlight attenuation factor for the ith light source at the vertex V, as given by Equation (11.11).

So finally, here it is, the grand ole lighting equation of OpenGL:

$$\begin{aligned} V_X \;=\;& V_{emit,\,X} \;+\\ & globAmb_X * V_{amb,\,X} \;+\\ & \sum_{i=0}^{n-1} \frac{1}{k_c^i + k_l^i d^i + k_q^i (d^i)^2} \;*\; saf^i \;*\\ & \quad \Big(L^i_{amb,\,X} * V_{amb,\,X} \;+\\ & \quad \max\{l^i \cdot n\,,\,0\} \,*\, L^i_{dif,\,X} * V_{dif,\,X} \;+\\ & \quad (\max\{s^i \cdot n\,,\,0\})^f * L^i_{spec,\,X} * V_{spec,\,X} \Big) \end{aligned} \tag{11.12}$$

where V_X is the color intensity at V, X being any of RGB.

The additions to the first lighting equation (11.10) are exactly the two multiplicative terms on the third line of the current equation, representing distance attenuation and spotlight attenuation, respectively.

Remark 11.8. It's really Phong's lighting equation, but, given the context, we'll more often than not call it the OpenGL lighting equation.

Remark 11.9. We must revisit Exercise 11.6 at this time. Its implication that all individual light source ambients can be consolidated into the global ambient is not true any more if one uses Equation (11.12) instead of Equation (11.10), because the same light source can contribute different amounts of ambient light to different vertices owing to distance and spotlight attenuation.

Nevertheless, the simplification of setting all individual light source ambients to zero, and adjusting only the global, is probably still authentic enough for most applications.

Exercise 11.15. If there is a single directional light source in an OpenGL program, which is not distance attenuated, which of the three – ambient, diffuse and specular – reflectance components at its vertices is changed by *translating* an object?

Exercise 11.16. If there is a single positional light source in an OpenGL program, which is not a spotlight and not distance attenuated, which of the three – ambient, diffuse and specular – reflectance components at an object's vertex can change by moving the light source? By translating the object?

11.8 OpenGL Shading Models

A *shading model* is a method to shade, or color, the interiors of primitives. Keep in mind that Phong's lighting model, as implemented through the OpenGL lighting equation, determines colors *only* at the vertices of primitives, but says nothing about how to spread them inside. OpenGL's default shading model, called *smooth shading* or *Gouraud* shading, is to interpolate color values computed at its vertices through a primitive's interior. We discussed in Section 7.2 the mechanics of interpolation by computing barycentric coordinates of interior points.

An alternate shading model, called *flat shading*, is available, as well, in OpenGL. It is specified by a call to

```
glShadeModel(GL_FLAT)
```

The default of smooth shading is restored by calling

```
glShadeModel(GL_SMOOTH)
```

When flat shading, even if the color values differ across the vertices of a primitive, OpenGL chooses *one* of them and applies its color to the entire primitive. For example, the first vertex (according to the order in the code) of a polygon is used. In a triangle strip, the i th triangle is painted with the color of the $i + 2$ th vertex. The reader is referred to the red book for a full listing of which vertex it is whose color is used for a given primitive.

Flat shading can be a reasonable alternative in the absence of lighting. Computationally it's, of course, far less expensive than smooth shading. One interesting application of flat shading is in applying "discrete" color schemes, which, often, is difficult with smooth shading. The following experiment is an illustration.

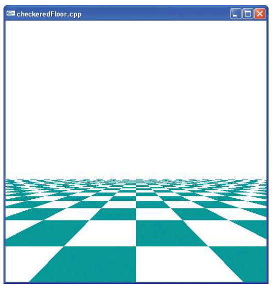


Figure 11.24: Screenshot of `checkeredFloor.cpp`.

Experiment 11.17. Run `checkeredFloor.cpp`, which creates a checkered floor drawn as an array of flat shaded triangle strips. See Figure 11.24. Flat shading causes each triangle in the strip to be painted with the color of the last of its three vertices, according to the order of the strip's vertex list.
End

Exercise 11.17. **(Programming)** Try and replicate the checkered floor of the preceding experiment using smooth shading instead of flat.

In Section 11.12 we'll see yet another shading model – Phong's shading model – which is more sophisticated than smooth shading and computationally more expensive as well. Phong's shading model is not available in first-generation OpenGL, i.e., OpenGL 1.x, but can be user-coded in the second generation of the API.

11.9 Animating Light

There are three ways that a light source can be animated by changing spatial properties:

1. By moving its position.
2. By changing its direction if it's a spotlight.

3. By changing the light cone angle if it's a spotlight.

We've already seen light animation in two programs in this chapter: `lightAndMaterial2.cpp` and `spotlight.cpp`.

The things to keep in mind are:

(a) A light source's position vector, specified by a `glLightfv(`*light*`, GL_POSITION, `*lightPos*`)` statement, is transformed by the value of the current modelview matrix by multiplication from the left. (See Section 4.2 if you need to review modelview matrices.)

Effectively, modelview transformations in the code prior to the `glLightfv(`*light*`, GL_POSITION, `*lightPos*`)` statement apply to a light's position, exactly as those prior to a `glVertex3f()` statement, defining a vertex, apply to that vertex.

(b) Likewise, a spotlight source's direction vector, specified by the `glLightfv(`*light*`, GL_SPOT_DIRECTION, `*spotDirection*`)` statement is transformed by the value of the current modelview matrix by multiplication from the left.

For example, as the light source of `sphereInBox1.cpp` is positioned by the `glLightfv(GL_LIGHT0, GL_POSITION, lightPos)` statement in the initialization routine `setup()`, it is unaffected by any modelview transformations in `drawScene()`.

However, both lights of `lightAndMaterial1.cpp` are positioned in the display routine following the viewing command `gluLookAt()`, so their positions are, in fact, transformed by `gluLookAt()`, which effectively means that the lights stay static relative to the scene, no matter if the viewpoint is changed. The position of the lights in `lightAndMaterial2.cpp` is similarly transformed by its own `gluLookAt()`.

Note: The push-pop pairs surrounding the code to position the lights in both programs are to isolate the transformations applied to the spheres that depict the light sources.

The spotlight of `spotlight.cpp` is positioned in the display routine after the viewing transformation *and* a user-specified translation; moreover, its cone angle can be changed by the user too. We ask you next to look into changing its direction.

Exercise 11.18. (Programming) Consider this an extension of Exercise 11.14 – add capability to aim the spotlight of `spotlight.cpp`.

Remark 11.10. We've discussed only animating the spatial attributes of a light source. Color values can easily be animated as well.

Exercise 11.19. (Programming) Cause the color of the balls of `spotlight.cpp` to brighten, fade and change.

Exercise 11.20. (Programming) Make the ball of `ballAndTorus.cpp` carry a spotlight, which is aimed always at the torus, and whose cone angle and color change as the ball travels. You may want to copy some light and material properties from `ballAndTorusShadowed.cpp`, but ignore the shadows.

11.10 Partial Derivatives, Tangent Planes and Normal Vectors 101

This section is an introduction to the calculus sometimes required to calculate normals to surfaces. It is not mandatory reading. We suggest you skip this section initially and consult it later if need be.

Actually, if you know how to compute derivatives of a function of a single variable, e.g., $f(x) = x^2$ or $f(x) = \sin x$, as we'll assume you do, you already know how to compute partial derivatives. Because. . .

Definition 11.1. Suppose that f is a function of more than one variable $x, y, \ldots$ The *partial derivative* of f with respect to one of these variables, say x, is the derivative of f as a function *only* of x, assuming the other variables all fixed. The partial derivative of f with respect to x is denoted $\frac{\partial f}{\partial x}$.

Example 11.6. Evaluate the partial derivatives of

$$f(x, y) = x^2 + y^2$$

at the point $(1, 2)$.

Answer: We have

$$\frac{\partial f}{\partial x}(x, y) = 2x, \qquad \frac{\partial f}{\partial y}(x, y) = 2y$$

Therefore,

$$\frac{\partial f}{\partial x}(1, 2) = 2, \qquad \frac{\partial f}{\partial y}(1, 2) = 4$$

Remark 11.11. Often $\frac{\partial f}{\partial x}(x, y)$ is simply written $\frac{\partial f}{\partial x}$, e.g., the first two equations of the preceding answer could be written

$$\frac{\partial f}{\partial x} = 2x, \qquad \frac{\partial f}{\partial y} = 2y$$

Example 11.7. Evaluate the partial derivatives of

$$f(x, y) = x^2 \sin y$$

at the point $(1, \pi/2)$.

Answer: We have

$$\frac{\partial f}{\partial x} = 2x \sin y, \qquad \frac{\partial f}{\partial y} = x^2 \cos y$$

Therefore,

$$\frac{\partial f}{\partial x}(1,\, \pi/2) = 2, \qquad \frac{\partial f}{\partial y}(1,\, \pi/2) = 0$$

Example 11.8. Evaluate the partial derivatives of

$$f(x, y, z) = xz + \sin x \cos y \cos z + y$$

at the point $(\pi/2, \pi, 0)$.

Answer: We have

$$\frac{\partial f}{\partial x} = z + \cos x \cos y \cos z, \quad \frac{\partial f}{\partial y} = 1 - \sin x \sin y \cos z, \quad \frac{\partial f}{\partial z} = x - \sin x \cos y \sin z$$

Therefore,

$$\frac{\partial f}{\partial x}(\pi/2,\, \pi,\, 0) = 0, \quad \frac{\partial f}{\partial y}(\pi/2,\, \pi,\, 0) = 1, \quad \frac{\partial f}{\partial y}(\pi/2,\, \pi,\, 0) = \pi/2$$

Exercise 11.21. Evaluate the partial derivatives of

$$f(x, y) = xy$$

at the point $(2, 3)$.

Exercise 11.22. Evaluate the partial derivatives of

$$f(x, y, z) = x \cos y + y \cos z + z \cos x$$

at the point $(\pi/2, 0, \pi/2)$.

The reader may wonder that if the partial derivative $\frac{\partial f}{\partial x}$, for example, is obtained by differentiating f with respect to the single variable x, assuming the others fixed, then why those other variables occasionally pop up again in the expression for $\frac{\partial f}{\partial x}$? Here's the reason.

Consider the function $f(x, y) = x^2 \sin y$ of Example 11.7 above. Fixing y at, say, the value $\pi/6$ gives the function $f(x, \pi/6) = x^2/2$, while fixing y at $\pi/2$ gives the function $f(x, \pi/2) = x^2$. Both $f(x, \pi/6)$ and $f(x, \pi/2)$ are functions of the one variable x, but they are *different* functions because y's been fixed at two *different* values.

Moreover,

$$\frac{\partial f}{\partial x}(x, \pi/6) = \frac{\mathrm{d}}{\mathrm{d}x}(x^2/2) = x \quad \text{and}$$

$$\frac{\partial f}{\partial x}(x, \pi/2) = \frac{\mathrm{d}}{\mathrm{d}x}(x^2) = 2x$$

are different as well, as they are derivatives of different functions. This is why $\frac{\partial f}{\partial x}$ depends on y, as well as on x.

So far so good. At least calculating partial derivatives is no different from calculating ordinary derivatives. But what do partial derivatives mean? For example, we understand the *geometric meaning* of ordinary derivatives along curves specified both implicitly and parametrically:

(a) *Implicit*: Suppose a curve is given by the equation

$$y = f(x)$$

Then the value of

$$\frac{\mathrm{d}f}{\mathrm{d}x}$$

at $x = a$ is the gradient of the tangent line to the curve at the point $(a, f(a))$.

For example, the gradient of the tangent line l at the point $(1, 1)$ of the parabola

$$y = x^2$$

is 2 as

$$\frac{\mathrm{d}}{\mathrm{d}x}(x^2) = 2x$$

which equals 2 when x is 1. See Figure 11.25(a).

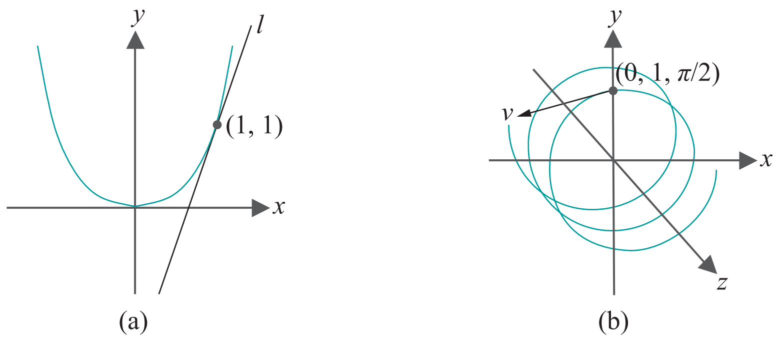


Figure 11.25: Tangents: (a) Tangent line l to the parabola $y = x^2$ at $(1, 1)$ (b) Tangent vector v to the helix $c(t) = (\cos t,\ \sin t,\ t)$ at $(0,\ 1,\ \pi/2)$.

(b) *Parametric*: Suppose a curve is given by

$$c(t) = (f(t),\ g(t),\ h(t))$$

Then the value of the vector

$$c'(t) = \begin{bmatrix} \dfrac{\mathrm{d}f}{\mathrm{d}t} & \dfrac{\mathrm{d}g}{\mathrm{d}t} & \dfrac{\mathrm{d}h}{\mathrm{d}t} \end{bmatrix}^T$$

at $t = a$ is a tangent vector (provided it's non-zero) to the curve at the point $(f(a),\ g(a),\ h(a))$.

For example, a tangent vector v to the helix

$$c(t) = (\cos t,\ \ \sin t,\ \ t)$$

at the point $(0,\ 1,\ \pi/2)$, corresponding to $t = \pi/2$, is $[-1\ 0\ 1]^T$, as

$$\begin{bmatrix} \dfrac{\mathrm{d}}{\mathrm{d}t}(\cos t) & \dfrac{\mathrm{d}}{\mathrm{d}t}(\sin t) & \dfrac{\mathrm{d}}{\mathrm{d}t}(t) \end{bmatrix}^T = [-\sin t \quad \cos t \quad 1]^T$$

which equals $[-1\ 0\ 1]^T$ when $t = \pi/2$. See Figure 11.25(b).

It turns out that, just as the computation of partial derivatives is based on computing ordinary derivatives, their geometric significance obtains from that of ordinary derivatives too. Here's how:

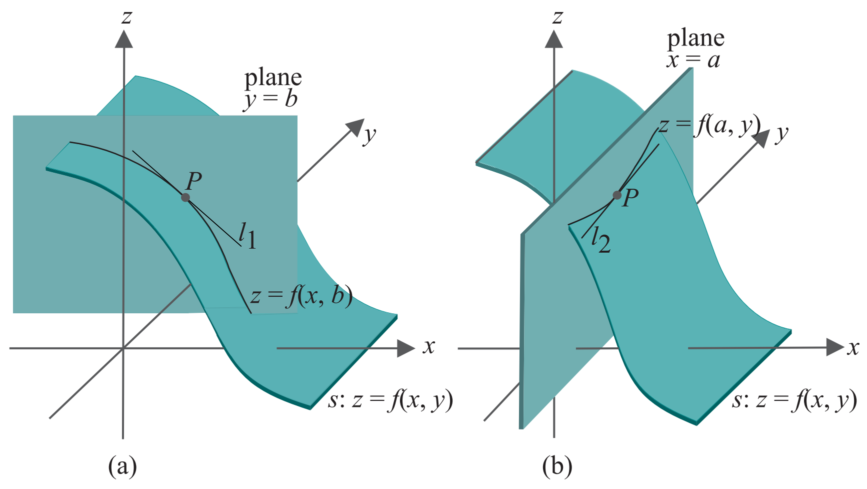


Figure 11.26: Section of the graph s of $z = f(x, y)$ by the (a) plane $y = b$, giving the tangent line l_1 at $P = (a,\ b,\ f(a, b))$, (b) plane $x = a$, giving the tangent line l_2 at P.

(a) *Implicit*: Consider $z = f(x, y)$, a function of two variables. It defines a surface s, called the graph of f. See Figure 11.26(a).

Now, if we fix y at, say, the value b, then $z = f(x, b)$ gives a curve s. In fact, this curve is the section of s by the plane $y = b$.

We know that the value of $\frac{\partial f}{\partial x}$ at (a, b) is the value at a of the ordinary derivative $\frac{\mathrm{d}}{\mathrm{d}x} f(x, b)$. This helps find geometric meaning for the partial derivative as follows.

The value of

$$\frac{\partial f}{\partial x}$$

at (a, b) is the gradient of the tangent line l_1 to the sectional curve $z = f(x, b)$ at the point $P = (a,\ b,\ f(a, b))$.

Likewise, the value of

$$\frac{\partial f}{\partial y}$$

at (a, b) is the gradient of the tangent line l_2, at the point $P = (a,\ b,\ f(a, b))$, to the curve $z = f(a, y)$, which is the section of s by the plane $x = a$ (Figure 11.26(b)).

(b) *Parametric*:

Consider next the surface s specified by the parametric equations

$$x = f(u, v),\ y = g(u, v),\ z = h(u, v),\ \ (u, v) \in W$$

where $W = [u_1, u_2] \times [v_1, v_2]$ is a rectangle in uv parameter space. The function $(u, v) \mapsto s(u, v) = (f(u, v),\ g(u, v),\ h(u, v))$ maps W to the surface s in 3-space. See Figure 11.27.

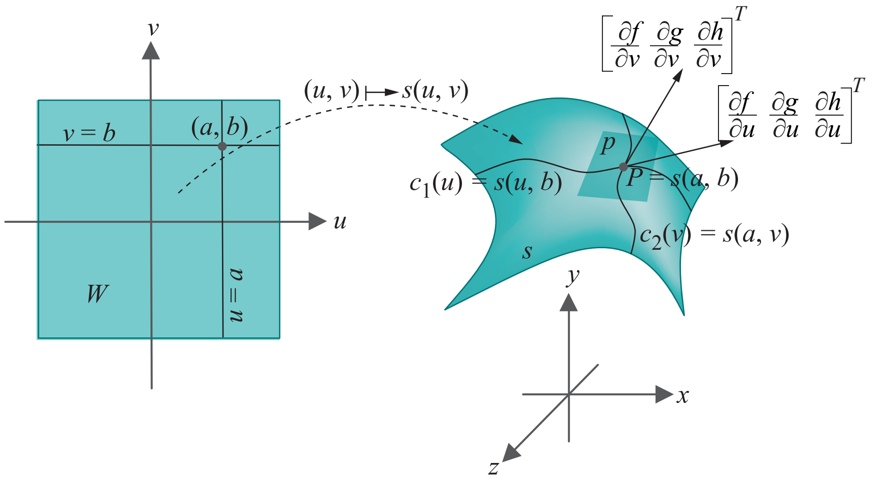

v
$v = b$
(a, b)
u
W
$u = a$
$(u, v) \mapsto s(u, v)$
$\left[\frac{\partial f}{\partial v}\ \frac{\partial g}{\partial v}\ \frac{\partial h}{\partial v}\right]^T$
$\left[\frac{\partial f}{\partial u}\ \frac{\partial g}{\partial u}\ \frac{\partial h}{\partial u}\right]^T$
p
$c_1(u) = s(u, b)$
$P = s(a, b)$
$c_2(v) = s(a, v)$
s
y
x
z

Figure 11.27: The surface s is the image of a parameter rectangle W by the map $(u, v) \mapsto s(u, v) = (f(u, v),\ g(u, v),\ h(u, v))$. Tangents to the parameter curves on s at the point $P = s(a, b)$ span the tangent plane p at P.

Fix a point $(a, b) \in W$. The image of the line $v = b$ by s is the u-parameter curve c_1 with equation

$$c_1(u) = (f(u,b),\ g(u,b),\ h(u,b)),\quad u \in [u_1, u_2]$$

The tangent vector to this curve is

$$\begin{aligned} c_1'(u) &= \begin{bmatrix} \frac{\mathrm{d}}{\mathrm{d}u} f(u,b) & \frac{\mathrm{d}}{\mathrm{d}u} g(u,b) & \frac{\mathrm{d}}{\mathrm{d}u} h(u,b) \end{bmatrix}^T \\ &= \begin{bmatrix} \frac{\partial f}{\partial u}(u,b) & \frac{\partial g}{\partial u}(u,b) & \frac{\partial h}{\partial u}(u,b) \end{bmatrix}^T \end{aligned}$$

at $u \in [u_1, u_2]$. Therefore, the value of the vector

$$\begin{bmatrix} \frac{\partial f}{\partial u} & \frac{\partial g}{\partial u} & \frac{\partial h}{\partial u} \end{bmatrix}^T$$

at the point (a, b) is a tangent vector (provided it's non-zero) to the u-parameter curve

$$c_1(u) = s(u, b)$$

at the point $s(a, b)$.

Likewise, the value of the vector

$$\begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial g}{\partial v} & \frac{\partial h}{\partial v} \end{bmatrix}^T$$

at the point (a, b) is a tangent vector (provided it's non-zero) to the v-parameter curve

$$c_2(v) = s(a, v)$$

at the point $s(a, b)$.

Definition 11.2. If the tangent vectors

$$\begin{bmatrix} \frac{\partial f}{\partial u} & \frac{\partial g}{\partial u} & \frac{\partial h}{\partial u} \end{bmatrix}^T$$

and

$$\begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial g}{\partial v} & \frac{\partial h}{\partial v} \end{bmatrix}^T$$

to the two parameter curves through the point $P = s(a, b)$ are linearly independent – in other words, they are not collinear – then they span a plane p, called the *tangent plane* to the surface s at P. This is the case in Figure 11.27.

Any line l on p through P is said to be a *tangent line* to s at P and any non-zero vector v lying on p is said to be a *tangent vector* to s at P (v is usually drawn emanating from P). See Figure 11.28. The line perpendicular to p through P is said to be the *normal line* to s at P and any non-zero vector lying on this line a *normal vector* to s at P.

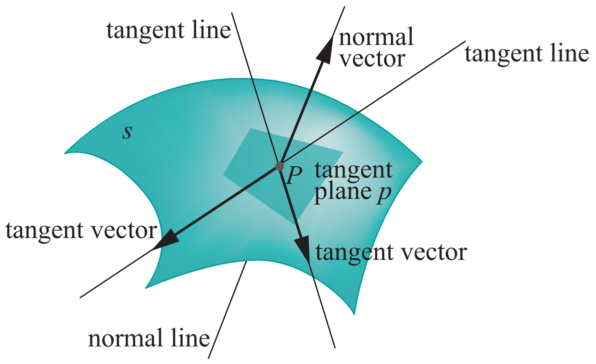


Figure 11.28: Two tangent lines and vectors on them, normal line and a normal vector to the surface s at P.

A tangent plane to a surface is precisely the geometric analogue of a tangent line to a curve. A thin straight stick pressed to a plane wire curve aligns itself along the tangent line at the point of contact; similarly, a thin flat board pressed to a surface in 3-space aligns itself along the tangent plane at the point of contact.

Example 11.9. Determine the tangent plane and a normal vector to the paraboloid

$$z = x^2 + y^2$$

at the point $(1, 2, 5)$.

Answer: It's easy first to write the given implicit equation in the parametric form

$$x = u, \quad y = v, \quad z = u^2 + v^2 \tag{11.13}$$

Differentiating,

$$\begin{aligned} \left[\frac{\partial x}{\partial u} \; \frac{\partial y}{\partial u} \; \frac{\partial z}{\partial u}\right]^T &= [1\ 0\ 2u]^T \\ \left[\frac{\partial x}{\partial v} \; \frac{\partial y}{\partial v} \; \frac{\partial z}{\partial v}\right]^T &= [0\ 1\ 2v]^T \end{aligned} \tag{11.14}$$

The point $(1, 2, 5)$ corresponds to the parameter values $u = 1$ and $v = 2$ in (11.13). Therefore, two tangent vectors to the paraboloid at $(1, 2, 5)$ are obtained by substituting these particular parameter values into the general expressions (11.14) above for tangent vectors at arbitrary points. Particularly, these two vectors are $[1\ 0\ 2]^T$ and $[0\ 1\ 4]^T$, which are evidently linearly independent. Therefore, the tangent plane to the paraboloid at $(1, 2, 5)$ is spanned by $[1\ 0\ 2]^T$ and $[0\ 1\ 4]^T$. See Figure 11.29.

A normal vector to the paraboloid at the point $(1, 2, 5)$ is perpendicular to its tangent plane there and, therefore, to both spanning vectors $[1\ 0\ 2]^T$

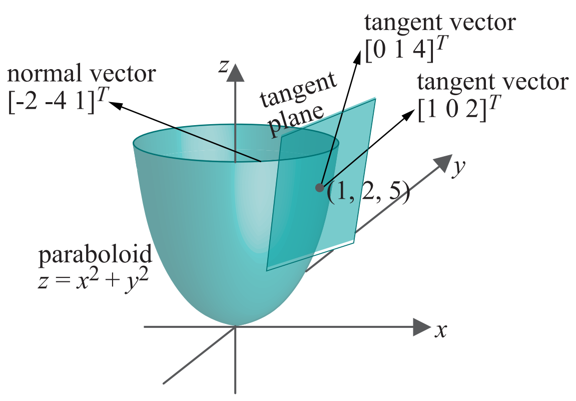


Figure 11.29: Tangent vectors, tangent plane and normal vector at the point $(1, 2, 5)$ to the paraboloid $z = x^2 + y^2$.

and $[0\ 1\ 4]^T$. It is obtained, then, as the cross-product of the latter (cross-products of vectors were reviewed in Section 5.4.3), viz.,

$$[1\ 0\ 2]^T \times [0\ 1\ 4]^T = [-2\ \ -4\ 1]^T$$

Remark 11.12. Computing the tangent plane at a point of a surface and computing a normal vector there are equivalent.

Exercise 11.23. Determine the tangent plane and a normal vector to the circular cylinder

$$x = \cos u,\ \ y = \sin u,\ \ z = v$$

at the point corresponding to the parameter values $(u, v) = (\pi/4,\ 3)$.

Exercise 11.24. Determine the tangent plane and a normal vector to the saddle-shaped surface (hyperbolic paraboloid is the mathematical name)

$$z = xy$$

at the point $(2, 3, 6)$.

Exercise 11.25. (Programming) Draw the paraboloid of Example 11.9 and its tangent plane at some point. The paraboloid should be wireframe and the tangent plane a finely meshed rectangle. Allow the user to press the arrow keys to slide the tangent plane over the paraboloid.

Normals from Function Gradients

Definition 11.2 of a tangent plane assumes a parametric representation of the surface. There's, however, a neat way to compute directly a normal vector at a point of a surface given implicitly.

If a surface s is specified implicitly by an equation of the form

$$F(x, y, z) = 0$$

then a normal vector to the surface at the point (a, b, c) is given by the value of the so-called *gradient* of F, denoted $grad(F)$, at that point, provided this value is not the zero vector. The gradient is defined by

$$grad(F) = \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial y} & \frac{\partial F}{\partial z} \end{bmatrix}^T$$

We'll not try to prove that $grad(F)$ is indeed normal to the surface $F(x, y, z) = 0$, but simply assume so for the purpose of computation. For the actual proof and more about the gradient, as well as its related functions *divergence* and *curl*, the reader is referred to books on vector calculus, e.g., Schey [118] and Spiegel [130].

Example 11.10. Determine a normal vector to the paraboloid

$$z = x^2 + y^2$$

at the point $(1, 2, 5)$.

Answer: Write the implicit equation in the form

$$F(x, y, z) = z - x^2 + y^2 = 0$$

Then

$$grad(F) = \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial y} & \frac{\partial F}{\partial z} \end{bmatrix}^T = [-2x \quad -2y \quad 1]^T$$

Therefore, a normal vector at the point $(1, 2, 5)$ is $[-2 \quad -4 \quad 1]^T$, which is obtained from putting $x = 1$ and $y = 2$ in the preceding equation. This result checks with Example 11.9.

Exercise 11.26. Verify your answer to Exercise 11.23 by finding a normal vector to the cylinder using the *grad* function. You must write an implicit equation for the cylinder first.

11.11 Computing Normals and Lighting Surfaces

Look carefully at the OpenGL lighting equation (11.12) once more. Outside of a bunch of user-specified color properties, the only data needed to compute the color intensities at a vertex V of an object O consists of the position of V, the positions of the light sources *and* the normal vector n at V.

The position of V is, of course, part of O's design. As for the light sources, they are usually few, and the user is free to locate them as he pleases. Remaining is the normal vector n, which the user is free to set as well. However, for authentic lighting it should actually be perpendicular to the surface of O at V or at least nearly so. For example, the choice of the normal vector n at the vertex V of the sphere in Figure 11.30 seems good, though either of the other two vectors drawn there could conceivably have been picked as well.

We'll discuss computing surface normals following the informal taxonomy of 2D objects in Section 10.2 before moving on to Bézier and quadric surfaces for which OpenGL provides automatic normals.

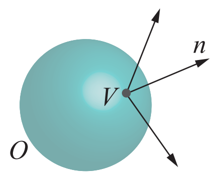


Figure 11.30: Three vectors at a vertex on a sphere, one of which has been chosen as the normal.

11.11.1 Polygons and Planar Surfaces

Polygons in particular, and planar surfaces in general, are the simplest. The normal at each vertex is simply normal to the plane itself containing the surface. In particular, unit vertex normals are all identical across a given side of the surface.

So how does one determine the normal direction to a plane p? If two non-collinear vectors u and v are known to lie on p, then the cross-product $u \times v$ is normal to p (cross-products were reviewed in Section 5.4.3). For example, any two adjacent edges of a polygon determine non-collinear vectors u and v spanning the plane p containing the polygon; therefore, $u \times v$ is normal to p. In Figure 11.31, $n = (P_1 - P_0) \times (P_4 - P_0)$ is normal to p.

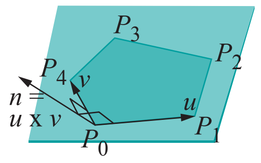


Figure 11.31: Vector n is normal to the plane p.

Exercise 11.27. Determine a normal to the plane p of the triangle with vertices at

$$P_0 = [0\ 3\ 5]^T, \quad P_1 = [1\ \ -2\ 0]^T, \quad P_2 = [3\ 3\ 3]^T$$

11.11.2 Meshes

Polygonal meshes are of interest next. Let's work with real examples.

Experiment 11.18. Run again `sphereInBox1.cpp`. The normal vector values at the eight box vertices of `sphereInBox1.cpp`, placed in the array `normals[]`, are

$$[\pm 1/\sqrt{3} \quad \pm 1/\sqrt{3} \quad \pm 1/\sqrt{3}]^T$$

each corresponding to one of the eight possible combinations of signs. **End**

The choice of the normals in `sphereInBox1.cpp` is easily motivated. The box being situated symmetrically about the origin, the normal values are chosen as unit vectors along the lines from the origin to each of the eight vertices, which indeed give the values above. The box is depicted in Figure 11.32(a), where only the normal vector at the lower-right vertex V

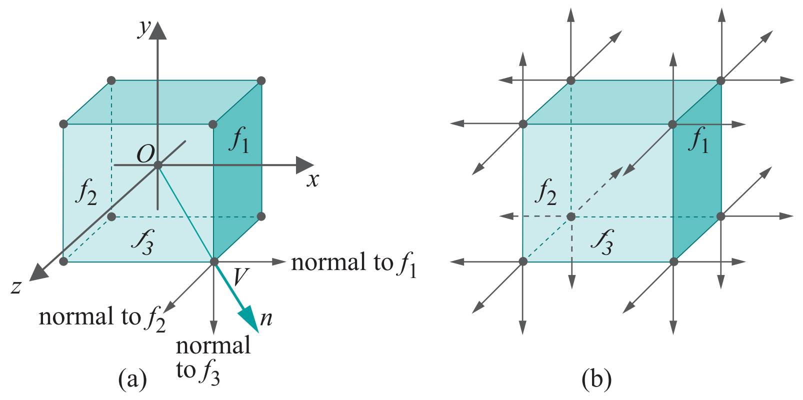


Figure 11.32: (a) The box of `sphereInBox1.cpp` with the averaged normal vector n at vertex V, together with the normals to the three faces that meet at V (f_1 right face, f_2 front face, f_3 bottom face) (b) The unaveraged normals of `sphereInBox2.cpp`.

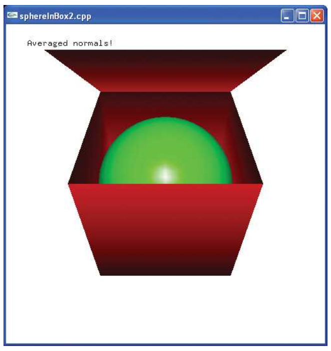
of the front face is shown: it is the arrow n drawn by extending OV a unit distance from V.

In fact, probably a better rationale for this particular choice of normals – which would still hold if the same box happened to be drawn not centered at the origin, but elsewhere – is that the one at each vertex is the normalized *average* of the unit outward normals to the three faces meeting at that vertex. For example, in Figure 11.32(a) the unit outward normals to f_1, f_2 and f_3 are $[1\ 0\ 0]^T$, $[0\ 0\ 1]^T$ and $[0\ \ -1\ 0]^T$, respectively, whose average is $[1/3\ \ -1/3\ \ 1/3]^T$, which normalizes to $[1/\sqrt{3}\ \ -1/\sqrt{3}\ \ 1/\sqrt{3}]^T$, which one can verify from the code is indeed the value of the normal at V in `sphereInBox1.cpp`.

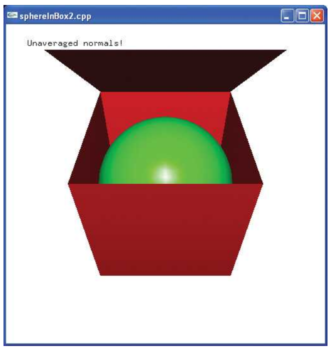
Although they possess the virtue of symmetry, it's clear, nevertheless, the box normals of `sphereInBox1.cpp` are not nearly actually perpendicular to the surface of the box, in particular, not to any of its faces. This consideration leads to another approach – to set the normal at each vertex of a face as a normal to that face itself. This is implemented as an option in `sphereInBox2.cpp`.

Experiment 11.19. Run `sphereInBox2.cpp`, which modifies `sphereIn-Box1.cpp`. Press the arrow keys to open or close the box and space to toggle between two methods of drawing normals.

The first method is that of `sphereInBox1.cpp`, specifying the normal at each vertex as an average of incident face normals. The second creates the box by first drawing one side as a square with the normal at each of its four vertices specified to be the unit vector perpendicular to the square, then placing that square in a display list and, finally, drawing it six times appropriately rotated. Figure 11.32(b) shows the vertex normals to three faces. Figure 11.33 shows screenshots of the box created with and without averaged normals. End



(a)



(b)

Figure 11.33: Screenshot of `sphereInBox2.cpp`: (a) Averaged box normals (b) Unaveraged box normals.

The contrast in output between the two ways of defining box normals in `sphereInBox2.cpp` is clear and the reason not hard to understand. The first method softens the edges because the averaged normal at each vertex is shared by all its three adjacent faces. Consequently, the interpolation of color values in each face's interior continues smoothly across its boundary.

The second method is significantly different. As each face is drawn separately with the normals at all its four vertices equal and perpendicular to the face itself, interpolation in the interior results in the entire face being colorized as if with that one normal value throughout. Moreover, this normal value turns abruptly by 90° from one face to the next. The upshot is that there is a significant difference in color intensities, as well, from one face to the next, throwing the edges between them into sharp relief. Which approach to choose depends on the effect desired.

Remark 11.13. Using the second method, colors at pixels along an edge are defined differently by its two adjacent faces, while pixel colors at a vertex are defined, in fact, by its three adjacent faces. At these pixels, therefore, code order determines which color prevails. This is not desirable, but it is not a serious issue because such "ambiguous" pixels lie only along edges and not in the interior of faces which constitute the bulk of the figure.

Versions of the averaging approach implemented sometimes to achieve greater realism use a *weighted* average rather than a straight one. Two possibilities are:

(a) Weight each adjacent face normal with the angle of that face at the vertex. In Figure 11.34, five faces meet at the vertex V subtending angles $\theta_1, \theta_2, \dots, \theta_5$, respectively. The angle-weighted average value of the normal at V is:

$$n = \frac{\theta_1 n_1 + \theta_2 n_2 + \theta_3 n_3 + \theta_4 n_4 + \theta_5 n_5}{\theta_1 + \theta_2 + \theta_3 + \theta_4 + \theta_5}$$

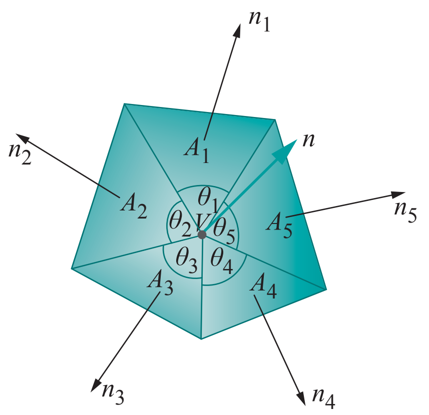


Figure 11.34: Weighted average of normals: θ_i are angles, A_i area, n_i face normals and n a weighted average normal at V.

(b) Weight each adjacent face normal with the area of that face. The areas of the five faces in Figure 11.34 meeting at V are $A_1, A_2, \dots, A_5$, respectively. The area-weighted average value of the normal at V is then:

$$n = \frac{A_1 n_1 + A_2 n_2 + A_3 n_3 + A_4 n_4 + A_5 n_5}{A_1 + A_2 + A_3 + A_4 + A_5}$$

Important: Whatever approach you adopt to compute normals, make sure, as a last step, to normalize each to unit length (easy enough – just divide each by its length). The reason is that OpenGL uses the dot product to compute the cosine of the angle between two vectors (see Equation (11.12)), which is correct *only if* they are of unit length.

Example 11.11. For the trash can mesh whose vertices are given in Figure 11.35, compute the unit normals to the three faces adjacent to the vertex V. Then compute the (unweighted) average of these three normals and normalize to unit length.

Answer: The three edge vectors emanating from V are:

$$\begin{aligned}
u_1 &= [1\ \ -1\ \ -1]^T - [1\ \ -1\ \ 1]^T = -2\mathbf{k} \\
u_2 &= [1.2\ \ 1\ \ 1.2]^T - [1\ \ -1\ \ 1]^T = 0.2\mathbf{i} + 2\mathbf{j} + 0.2\mathbf{k} \\
u_3 &= [-1\ \ -1\ \ 1]^T - [1\ \ -1\ \ 1]^T = -2\mathbf{i}
\end{aligned}$$

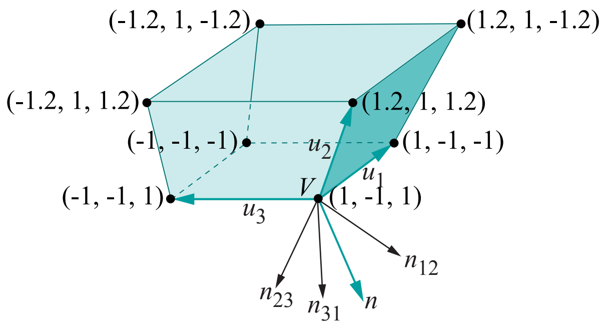


Figure 11.35: Trash can of five quadrilateral sides. The vectors n_{12}, n_{23} and n_{31} from V are normals to V's adjacent faces, while n is the averaged normal.

Therefore, the outward unit normal to the face with edges u_1 and u_2 is

$$n_{12} = (u_1 \times u_2) \; / \; |u_1 \times u_2| = (4\mathbf{i} - 0.4\mathbf{j}) \; / \; \sqrt{4^2 + 0.4^2} \simeq 0.995\mathbf{i} - 0.0995\mathbf{j}$$

and that to the face with edges u_2 and u_3 is

$$n_{23} = (u_2 \times u_3) \; / \; |u_2 \times u_2| = (-0.4\mathbf{j} + 4\mathbf{k}) \; / \; \sqrt{4^2 + 0.4^2} \simeq -0.0995\mathbf{j} + 0.995\mathbf{k}$$

while the outward unit normal to the face with edges u_3 and u_1, the bottom face, is easily seen to be

$$n_{31} = -\mathbf{j}$$

The normalize average of these normals is

$$\begin{aligned} n &= (n_{12} + n_{23} + n_{31}) \; / \; |n_{12} + n_{23} + n_{31}| \\ &\simeq (0.995\mathbf{i} - 1.199\mathbf{j} + 0.995\mathbf{k}) \; / \; \sqrt{0.995^2 + 1.199^2 + 0.995^2} \\ &\simeq 0.538\mathbf{i} - 0.649\mathbf{j} + 0.538\mathbf{k} \end{aligned}$$

Exercise 11.28. (Programming) Use data from the preceding example to replace the box of `sphereInBox2.cpp` with a trash can. Omit the sphere. Let the user choose between averaged and unaveraged normals. Allow the can to be rotated keeping the light source fixed.

11.11.3 General Surfaces

As a general surface is drawn by approximating it with a polygonal mesh, the thought comes to mind to simply use the methods of the preceding section to find normals. Precisely, (a) formulate a mesh approximation of the surface and (b) specify the normal at each vertex as an average of those of its adjacent faces (we really want to use an average here, especially if the original surface is smooth, to avoid color discontinuities between adjacent mesh faces).

This approach is perfectly reasonable if the surface is known to the user only by its mesh approximation. However, if one knows, say, a parametric representation of the surface, why not get the normals from the "horse's mouth" – that being the parametrization itself? In other words, use the parametrization to *analytically* compute the normals at the mesh vertices. This makes for stable normals independent of the vagaries of the particular mesh approximation, not to mention those of the averaging process (possibly, angle-weighted or area-weighted). For example, working from the mesh approximation of the surface s in Figure 11.36, normals to the six faces adjacent to vertex V must be averaged to determine the normal n at V. However, knowledge of s itself could enable a direct computation.

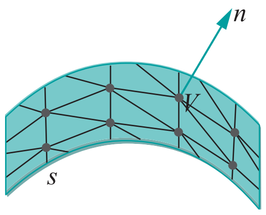


Figure 11.36: Normal vector n to the surface s at a vertex V of its mesh approximation.

So let's see how to compute normals analytically. We're going to assume in the following that you know that a tangent plane at the point $s(u, v)$ to a surface s given parametrically by the equations

$$x = f(u, v), \ y = g(u, v), \ z = h(u, v)$$

is spanned by the two vectors

$$\begin{bmatrix} \frac{\partial f}{\partial u} & \frac{\partial g}{\partial u} & \frac{\partial h}{\partial u} \end{bmatrix}^T \quad \text{and} \quad \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial g}{\partial v} & \frac{\partial h}{\partial v} \end{bmatrix}^T$$

evaluated at (u, v) (provided they are not collinear). Moreover, a normal vector to s at $s(u, v)$ is the cross-product

$$\begin{bmatrix} \frac{\partial f}{\partial u} & \frac{\partial g}{\partial u} & \frac{\partial h}{\partial u} \end{bmatrix}^T \times \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial g}{\partial v} & \frac{\partial h}{\partial v} \end{bmatrix}^T \tag{11.15}$$

evaluated at (u, v). If you need to brush up, Section 11.10 has a review of the needed calculus.

Denote the normalized value of the vector (11.15) – obtained by dividing it by its magnitude – by

$$[f_n(u, v) \quad g_n(u, v) \quad h_n(u, v)]^T \tag{11.16}$$

which, therefore, is a unit normal to s at $s(u, v)$.

Finally, we'll specify either $[f_n(u, v) \quad g_n(u, v) \quad h_n(u, v)]^T$ or its reverse, $[-f_n(u, v) \quad -g_n(u, v) \quad -h_n(u, v)]^T$, as the unit normal at $s(u, v)$ depending on which direction is appropriate for front-facing triangles. There's not much to worry about making a wrong choice, as it'll be plenty clear from the viewable output! Let's get to work on a benign surface first.

Cylinder

Example 11.12. Consider the circular cylinder $s(u, v)$ with parametric equations

$$x = \cos u, \ y = \sin u, \ z = v, \quad \text{where} \quad (u, v) \in [-\pi, \pi] \times [-1, 1]$$

We drew it using these equations in `cylinder.cpp` of Experiment 10.3. To color and light, let's do normal calculations. The vectors spanning the tangent plane at $s(u, v)$ are

$$\left[\frac{\partial(\cos u)}{\partial u} \quad \frac{\partial(\sin u)}{\partial u} \quad \frac{\partial v}{\partial u} \right]^T = [-\sin u \quad \cos u \quad 0]^T$$

and

$$\left[\frac{\partial(\cos u)}{\partial v} \quad \frac{\partial(\sin u)}{\partial v} \quad \frac{\partial v}{\partial v} \right]^T = [0\ 0\ 1]^T$$

so a normal vector is

$$[-\sin u \quad \cos u \quad 0\,]^T \times [\,0\ 0\ 1\,]^T = [\,\cos u \quad \sin u \quad 0]^T$$

which happens to be normalized already. So, in the terminology of (11.16), for the cylinder,

$$f_n(u, v) = \cos u, \quad g_n(u, v) = \sin u, \quad h_n(u, v) = 0$$

We'll add this normal data to `cylinder.cpp` next.

Experiment 11.20. Run `litCylinder.cpp`, which builds upon `cylinder.cpp` using the normal data calculated above, together with color and a single directional light source. Press 'x/X', 'y/Y' and 'z/Z' to turn the cylinder. The functionality of being able to change the fineness of the mesh approximation has been dropped. Figure 11.37 is a screenshot. **End**

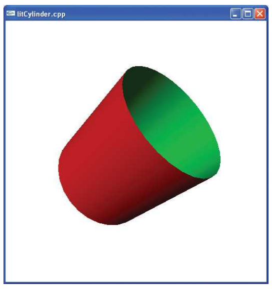


Figure 11.37: Screenshot of `litCylinder.cpp`.

Compare the two programs `cylinder.cpp` and `litCylinder.cpp` – it's not really a lot of code from the first to the second. Essentially, the additions are (a) the `fn()`, `gn()` and `hn()` normal component functions as calculated above, (b) the `fillNormalArray()` function to fill the array `normals[]`, and (c) a bunch of routine code specifying light and material properties, which can be kept similar across most programs with lighting.

So the extra code arising from analytic normal computation is really in (a) and (b), about 20 lines all told. Not too bad, huh? And it gets better. As we used the template of `cylinder.cpp` to draw various surfaces, simply swapping in new `f()`, `g()` and `h()` functions according to the given parametrization, so we can use `litCylinder.cpp` for lit applications, additionally swapping in new `fn()`, `gn()` and `hn()` functions.

Exercise 11.29. (Programming) Reverse the normals of `litCylinder.cpp` by changing their specification in the `fillNormalArray()` routine as follows:

```
normals[k++] = -fn(i,j);
normals[k++] = -gn(i,j);
normals[k++] = -hn(i,j);
```

Not good! As we remarked earlier, wrongly-oriented normals are easy to spot. Can you fix the problem caused by the normal values above by a minimal amount of code change *only* in the drawing routine?
Hint: Think orientation, in particular, reversing the orientation of the strip triangles.

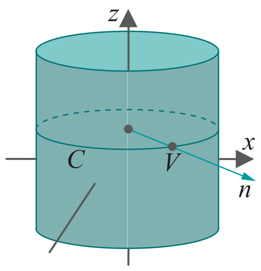


Figure 11.38: Normal n to a cylinder.

Exercise 11.30. It's a bit late now, but do we really need partial derivatives, as in Example 11.12, to determine the normal to the cylinder at the point $V = (\cos u, \sin u, v)$?

The outward normal to the cylinder at V evidently lies along a radius of the circle C which is the section of the cylinder through V by a plane perpendicular to its axis. See Figure 11.38. Use this to compute the parametric equation for a unit normal vector to the cylinder without any calculus.

Often, as in the preceding exercise, normals to a surface can be determined by elementary geometric considerations. Unfortunately, this does not seem to be the case with the doubly-curled cone of Experiment 10.8.

Doubly-curled Cone

Next, we light the doubly-curled cone of `doublyCurledCone.cpp`. Its parametric equations are

$$x = t\cos(A + a\theta)\cos\theta,\ y = t\cos(A + a\theta)\sin\theta,\ z = t\sin(A + a\theta),$$

where $0 \le t \le 1$ and $0 \le \theta \le 4\pi$. A somewhat tedious calculation gives a normal to the cone as

$$\begin{aligned}\begin{bmatrix}\frac{\partial x}{\partial\theta} & \frac{\partial y}{\partial\theta} & \frac{\partial z}{\partial\theta}\end{bmatrix}^T \times \begin{bmatrix}\frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} & \frac{\partial z}{\partial t}\end{bmatrix}^T & \\ = \ & [-at\sin\theta + t\sin(A + a\theta)\cos(A + a\theta)\cos\theta, \\ & \ at\cos\theta + t\sin(A + a\theta)\cos(A + a\theta)\sin\theta, \\ & \ -t\cos^2(A + a\theta)]^T \end{aligned} \tag{11.17}$$

Moreover, the length of this normal is

$$t\sqrt{a^2 + \cos^2(A + a\theta)} \tag{11.18}$$

Dividing the normal (11.17) by its length (11.18) gives a unit normal to the cone.

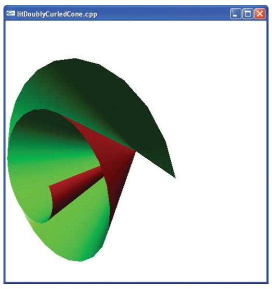


Figure 11.39: Screenshot of `litDoublyCurled-Cone.cpp`.

Experiment 11.21. The program `litDoublyCurledCone.cpp`, in fact, applies the preceding equations for the normal and its length. Press 'x/X', 'y/Y', 'z/Z' to turn the cone. See Figure 11.39 for a screenshot.

As promised, `litDoublyCurledCone.cpp` is pretty much a copy of `litCylinder.cpp`, except for the different `f()`, `g()`, `h()`, `fn()`, `gn()` and `hn()` functions, as also the new `normn()` to compute the normal's length.
End

Exercise 11.31. Verify Equations (11.17) and (11.18) for the normal and its magnitude of the doubly-curled cone.

Exercise 11.32. (Programming) The doubly-curled cone would probably benefit from at least one more light source, particularly to brighten the inside. Code this in.

Exercise 11.33. (Programming) Color and light the table of Experiment 10.7. You don't need any calculus in order to compute the normals to the various component surfaces – which happen each to be either cylindrical or flat. Make sure to choose normals so that edges appear sharp.

Exercise 11.34. (Programming) Color and light the helical pipe of Experiment 10.4.

Exercise 11.35. (Programming) Color and light the pipe of Exercise 10.46, which coils around a torus.

Exercise 11.36. (Programming) Color and light the single-sheeted hyperboloid of Experiment 10.11.

Exercise 11.37. Which of the three components – ambient, diffuse and specular – of light reflected from a vertex V are affected if the normal at V is altered?

11.11.4 Bézier and Quadric Surfaces

Good news! All one has to do is type in the command `glEnable(GL_AUTO_NORMAL)` for OpenGL to automatically calculate unit normals at the vertices of a Bézier surface which has been created using `glMap2f(GL_MAP2_VERTEX_3, ... )` and `glEnable(GL_MAP2_VERTEX_3)`.

Canoe

Experiment 11.22. Run `litBezierCanoe.cpp`. Press 'x/X', 'y/Y', 'z/Z' to turn the canoe. You can see a screenshot in Figure 11.40.

This program illuminates the final shape of `bezierCanoe.cpp` of Experiment 10.20 with a single directional light source. Other than the expected command `glEnable(GL_AUTO_NORMAL)` in the initialization routine, an important point to notice about `litBezierCanoe.cpp` is the reversal of the sample grid along the u-direction. In particular, compare the statement

```
glMapGrid2f(20, 1.0, 0.0, 20, 0.0, 1.0)
```

of `litBezierCanoe.cpp` with

```
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0)
```

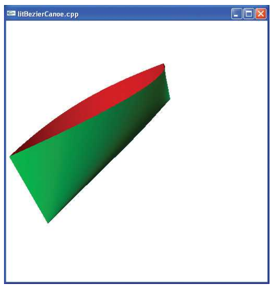


Figure 11.40: Screenshot of `litBezierCanoe.cpp`.

of bezierCanoe.cpp. This change reverses the directions of one of the tangent vectors evaluated at each vertex by OpenGL and, correspondingly, that of the normal (which is the cross-product of the two tangent vectors).

Modify litBezierCanoe.cpp by changing

```
glMapGrid2f(20, 1.0, 0.0, 20, 0.0, 1.0);
```

back to bezierCanoe.cpp's

```
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
```

Wrong normal directions! The change from bezierCanoe.cpp is necessary. Another solution is to leave glMapGrid2f() as it is in bezierCanoe.cpp, instead making a call to glFrontFace(GL_CW). **End**

The lesson to take from this is that if you obtain normals automatically from OpenGL, then you might have to subsequently alter their orientation for authenticity, which is not unreasonable because OpenGL cannot know which you intend to be the front face of a primitive.

Remark 11.14. If the user wishes to define her own normals for a Bézier surface, she can do so with a glMap2f(GL_MAP2_NORMAL, ...) call. We'll not have occasion to use this call ourselves.

Quadrics are even simpler. The call

```
gluQuadricNormals(qobj, GLU_SMOOTH)
```

automatically generates a normal at each vertex of the quadric pointed by qobj.

The next program we'll look at is a fairly substantial animation which invokes both glEnable(GL_AUTO_NORMAL) for Bézier surface normals and gluQuadricNormals(qobj, GLU_SMOOTH) for quadric surfaces.

Movie with a Ship and Torpedo

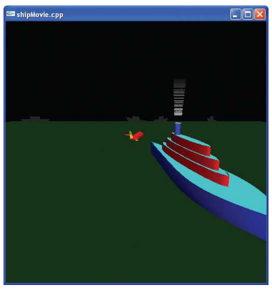


Figure 11.41: Screenshot of shipMovie.cpp.

Experiment 11.23. Run shipMovie.cpp. Pressing space start an animation sequence which begins with a torpedo traveling toward a moving ship and which ends on its own after a few seconds. Figure 11.41 is a screenshot as the torpedo nears the ship.

There are a few different objects. The hull of the ship is obviously inspired by the Bézier canoe of the previous experiment. The deck is a flat Bézier surface – all its control point y-values are identical – which is designed to fit the hull. Each of the ship's three storeys is a cylindrical quadric, as is its chimney.

The torpedo should be familiar from the program torpedo.cpp of Experiment 10.21. Each of the four grayish boats in the background is a couple of quads, while the sea itself is a solid blue cube.

The smoke from the chimney is a simple-minded *particle system*. In particular, we render a sequence of quadric discs in point mode and hack for it a coloring and animation scheme. **End**

Exercise 11.38. (Programming) The program `shipMovie.cpp` bears a lot of improvement. Try at least the following:

(a) Add detail to the ship.

(b) Make the water more realistic, possibly by adding movement, variation in color, etc.

(c) Put stars and a moon in the sky.

(d) Improve the smoke particle system.

(e) Make a particle system to simulate water spray from the torpedo's propeller.

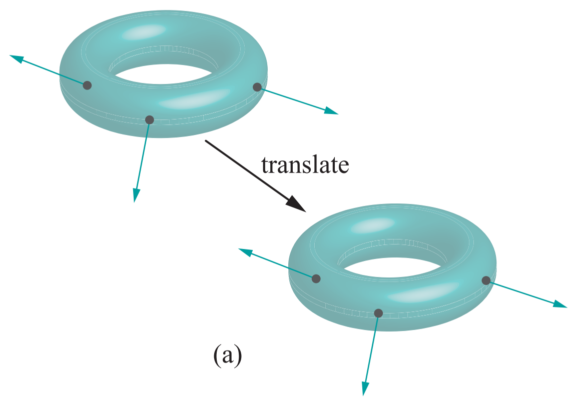
Exercise 11.39. (Programming) Fill, paint and light the character of `animateMan*.cpp` in surroundings less bland than a plane with a ball. Make an animation sequence.

11.11.5 Transforming Normals

Normals are transformed by modelview transformations, but not as straightforwardly as vertices are by multiplication from the left by the transformation matrix. Let's see how they are transformed by each of the fundamental transformations – translation, rotation and scaling.

1. Translation:

   A translation leaves a normal vector at a vertex unchanged because the normal simply translates parallelly (see Figure 11.42(a)).



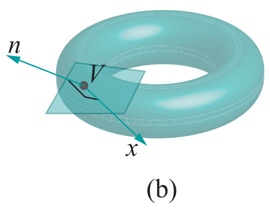


Figure 11.42: (a) Vertex normals translate parallelly as the torus is translated (b) The normal n at V is perpendicular to any vector x which lies on the tangent plane at V.

2. Rotation and Scaling:

   These cases are not as simple and require a bit of calculation.

   A given rotation or non-degenerate scaling, say t, is a non-singular linear transformation, with, therefore, a non-singular defining matrix, say N. Suppose, as well, that n is a normal vector at a vertex V of an object O. Therefore, n is perpendicular to an arbitrary vector, say x, tangent to the surface of O at V (see Figure 11.42(b)).

   Now, if we apply t, it will transform all the vertices of O, as well as vectors tangent to O's surface, by multiplication on the left by N.

   Note: To convince yourself that tangent vectors are transformed identically with vertices, think of a tangent vector as connecting two vertices infinitesimally close together on the surface of O. Therefore, these two vertices "carry" the tangent vector with them.

   So x is transformed to Nx. We would, therefore, like to transform n to a vector perpendicular to Nx. Since n is perpendicular to x, we already have $n \cdot x = 0$, which is equivalent to $n^T x = 0$, the latter being a matrix equation. It follows that

$$n^T(N^{-1}N)x = n^T x = 0$$

   Therefore,

$$0 = n^T(N^{-1}N)x = (n^T N^{-1})(Nx) = ((N^{-1})^T n)^T(Nx)$$

   (invoking rules of matrix algebra).

   One sees that $((N^{-1})^T n) \cdot Nx = 0$, so $(N^{-1})^T n$ is indeed perpendicular to Nx. The conclusion, then, is that the appropriate transformation to apply to the normal vector n, under a rotation or non-degenerate scaling corresponding to the matrix N, is left multiplication by $(N^{-1})^T$, i.e., $n \mapsto (N^{-1})^T n$.

OpenGL actually transforms normals as just described. If the current modelview matrix is

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{24} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

then "erasing" the translational part, which, as we know, has no impact on the normal, leaves the 3×3 matrix

$$N = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

and, in fact, the matrix $(N^{-1})^T$ to use to transform normals, called the *normal matrix*, is stored in a state variable (which can be accessed by the user via the Shading Language in OpenGL 2.0 and higher). It should be noted that, correspondingly, the OpenGL normal is a 3-vector (recall that it was only to accommodate translations into the matrix multiplication scheme that real world 3-vectors were homogenized to length 4).

Exercise 11.40. We gave above a general formula for how a normal vector is transformed by a rotation or non-degenerate scaling in terms of its defining matrix. Ignoring the formula for a moment, can you deduce from elementary considerations what should happen in the particular case of a rotation? Then relate your answer to the formula.

11.11.6 Normalizing Normals

Normalizing a (non-zero) vector means dividing it by its magnitude to obtain a vector with the same direction, but of unit length. We've already seen that it's important to specify normalized normals because OpenGL uses the dot product to compute the cosine of the angle between two vectors, which is correct only if they are both of unit length.

Here's a simple modification of `litTriangle.cpp` to show what can happen if one is careless.

Experiment 11.24. Run `sizeNormal.cpp` based on `litTriangle.cpp`.

The ambient and diffuse colors of the three triangle vertices are set to red, green and blue, respectively. The normals are specified separately as well, initially each of unit length perpendicular to the plane of the triangle.

However, pressing the up/down arrow keys changes (as you can see) the size, but not the direction, of the normal at the red vertex. Observe the corresponding change in color of the triangle. Figure 11.43 is a screenshot.

End

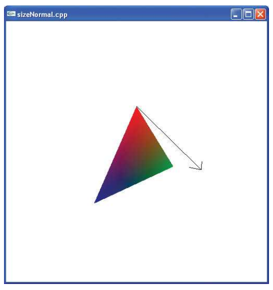


Figure 11.43: Screenshot of `sizeNormal.cpp`.

There are, typically, two reasons why normals turn out not normalized:

(a) The user does not specify them of unit length in the first place.

(b) Even if they are specified of unit length, a subsequent application of a scaling transformation changes the length.

If the user is not inclined to write code to ensure normals of unit length, there's a way to ask OpenGL's help. Calling `glEnable(GL_NORMALIZE)` causes OpenGL to normalize all normal vectors before lighting calculation. Beware, though, it's not a particularly efficient call and should be avoided if possible.

Experiment 11.25. Run `sizeNormal.cpp` after placing the statement `glEnable(GL_NORMALIZE)` at the end of the initialization routine. Press the up/down arrow keys. The triangle no longer changes color (though the white arrow still changes in length, of course, because its size is that of the program-specified normal). End

There's a cheaper renormalization call, `glEnable(GL_RESCALE_NORMAL)`, which can be used if you originally did provide unit normals that were subsequently all changed by the *same* scaling transformation.

11.12 Phong's Shading Model

An alternate shading model first proposed by Phong, though computationally far more intensive, significantly improves the realism of a rendered image.

Note: Phong's shading model should not be confused with his lighting model, which we know already that OpenGL implements.

Instead of computing light values only at primitives' vertices and then interpolating through its interior as in Gouraud shading – or smooth shading as it's also called, the OpenGL default – Phong suggested to (a) interpolate the vertex normal values through the primitive, and then (b) compute light values at each pixel using the interpolated normals.

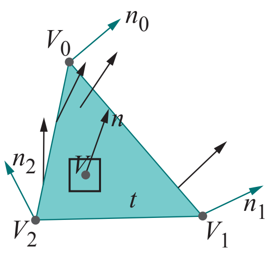


Figure 11.44: Normals n_0, n_1 and n_2 at the vertices of the triangle are programmer-specified. Shown also are (black) normalized interpolated normals at a few points and a pixel centered at V.

Figure 11.44 illustrates the idea. Unit normals n_0, n_1 and n_2 are specified by the programmer at the vertices V_0, V_1 and V_2, respectively, of triangle t. These normals are then interpolated, and normalized, throughout t. For example, if the barycentric coordinates of the point V are given by

$$V = c_0V_0 + c_1V_1 + c_2V_2$$

then the normal value n at V is computed to be

$$n = (c_0n_0 + c_1n_1 + c_2n_2) \,/\, |c_0n_0 + c_1n_1 + c_2n_2| \tag{11.19}$$

(provided the denominator is not zero).

The color values of a pixel which happens to be centered at V are then computed in Phong's model using the lighting equation (11.12), where, now, the normal value n applied is from (11.19) above, the color values $V_{*,X}$ are interpolated from the vertices as well, while the light direction and halfway vectors l^i and s^i are determined from the coordinates of V itself.

OpenGL, as we know, offers only flat and Gouraud shading as options. However, the OpenGL Shading Language, part of the version 2.0 specification, allows individual pixels to be programmed, which means the programmer herself can code in Phong shading. We'll be doing precisely this as an application, after learning the Shading Language ourselves in Chapter 20.

Remark 11.15. Phong lighting calculation at each vertex followed by Gouraud shading, OpenGL's default process, is often called *per-vertex* lighting to contrast it with the *per-pixel* lighting of Phong's shading model. (More appropriate might have been per-vertex and per-pixel *shading*, but the given usage is common.)

11.13 Summary, Notes and More Reading

In Chapters 4, 5 and 6 we learned to animate objects, in Chapter 10 to draw them, and now we have begun to “dress them up” with color and light. In this chapter we learned the underlying color and lighting models which OpenGL implements, the related syntax, and how to use them to specify light sources and material properties, as well as related environmental parameters. The technical issue of normal computation was an important part of our program too. We'll continue this theme in the next chapter when we learn of yet another technique to decorate an object, texturing.

For a further reference on coloring models, the somewhat encyclopedic Wyszecki and Stiles [147] is frequently called the bible of color science. The books by Berns [11] and Jackson et al. [71] are probably easier to read though.

Since the publication of Phong's model in 1975 [105] several other lighting models, both local and global, have been proposed. Local models like Phong's do not consider object-object light interaction, while global ones do, thereby displaying secondary effects such as shadows and reflections. Lighting models are often used in an application-specific manner, certain models being more realistic in rendering particular material properties and finishes.

A few of the local models which appeared after Phong's are Blinn [14], Cook-Torrance [28], He et al. [63, 64], Nayar-Oren [97], Poulin-Fournier [108] and Schlick [120]. However, the only local model that we discuss or use in this book is Phong's.

The two most commonly implemented global models are ray tracing [4, 143] and radiosity [54], which as a matter of fact complement each other. Global models, though much more realistic than local ones, are notoriously computation-intensive, so rarely apt for interactive applications. However, they are often used when frames can be created off-line, as in movies. We discuss both ray tracing and radiosity in Chapter 19.

The theory of lighting models necessarily involves a fair amount of physics and mathematics. The reader interested in learning more is best advised to start with advanced books such as those by Akenine-Möller, Haines & Hoffman [1], Buss [21] and Watt [142] and then proceed to original research papers, as the area is particularly active. The canonical source for the latest in CG research in general is the annual ACM SIGGRAPH conference [125].