An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 5 Color

5.1 Color Spaces

To describe an image, we need a way to represent the color information. To describe a gray-level image, we only need one number to indicate the brightness or luminance of each spatial sample. We need more numbers if we want to describe a color image. In reality, our perception of light depends on the light frequency and other properties. When we view a source of light, our eyes respond to three main sensations. The first one is the **color**, which is the main frequencies of the light. The second one is the **intensity** (or brightness), which represents the total energy of the light; we can quantify brightness as the luminance of the light. The third one is the **purity** (or saturation) of the light, which describes how close a light appears to be a pure spectral color, such as red, green or blue. Pale colors have low purity and they appear to be almost white, which consists of a mixture of the red, green and blue colors. We use the term **chromaticity** to collectively refer to the two characteristics of light, color purity and dominant frequency (hue).

Very often, we employ a **color model** to precisely describe the color components or intensities. In general, a color model is any method for explaining the properties or behavior of color within some particular context. No single model can explain all aspects of color, so people make use of different models to help describe different color characteristics. Here, we consider a **color model** as an abstract mathematical model that describes how colors are presented as tuples of numbers, typically as three or four values or color components; the resulting set of colors that define how the components are to be interpreted is called a **color space**. The commonly used **RGB** color model naturally fits the representation of colors by computers. However, it is not a good model for studying the characteristics of an image.

5.2 RGB Color Model

X-ray, light, infrared radiation, microwave and radio waves are all electromagnetic (EM) waves with different frequencies. Light waves lie in the visible spectrum with a narrow wavelength band from about 350 to 780 nm. The retina of a human eye can detect only EM waves lying within this visible spectrum but not anything outside. The eye contains two kinds of light-sensitive receptor cells, **cones** and **rods** that can detect light.

The **cones** are sensitive to colors and there are three types of cones, each responding to one of the three primary colors, red, green and blue. Scientists found that our perception of color is a result of our cones' relative response to the red, green and blue colors. We can form any color by mixing these three colors with certain intensity values. The human eye can distinguish about 200 intensities of each of the red, green and blue colors. Therefore, it is natural that we represent each of these colors by a byte which can hold 256 values. In other words, 24 bits are enough to represent the 'true' color. More bits will not increase the quality of an image as human eyes cannot resolve the extra colors. Each eye has 6 to 7 million cones located near the center of the eye, allowing us to see the tiny details of an object.

On the other hand, the **rods** cannot distinguish colors but are sensitive to dim light. Each eye has 75 million to 150 millions rods located near its corner, allowing us to detect peripheral objects in an environment of near darkness.

We can characterize a visible color by a function $C(\lambda)$ where λ is the wavelength of the color in the visible spectrum. The value for a given wavelength λ gives the relative intensity of that wavelength in the color. This description is accurate when we measure the color with certain physical instrument. However, the human visual system (HVS) does not perceive color in this way. Our brains do not receive the entire distribution $C(\lambda)$ of the visible spectrum but rather three values – the **tristimulus values** – that are the responses of the three types (red, green and blue) of cones to a color. This human characteristics leads to the formulation of the trichromatic theory: *If two colors produce the same tristimulus values, they are visually indistinguishable.* A consequence of of this theory is that it is possible to match all of the colors in the visible spectrum by appropriately mixing three primary colors. In other words, any color can be created by combining red, green, and blue in varying proportions. This leads to the development of the **RGB color model**.

The RGB (short for red, green, blue) color model decomposes a color into three components, Red (R), Green (G), and Blue (B); we can represent any color by three components R, G, B just like the case that a spatial vector is specified by three components x, y, z. If the color components R, G and B are confined to values between 0 and 1, all definable colors lie in a unit cube as shown in Figure 5-1. This color space is most natural for representing computer images, in which a color specification such as (0.1, 0.8, 0.23) can be directly translated into three positive integer values, each of which is represented by one byte.

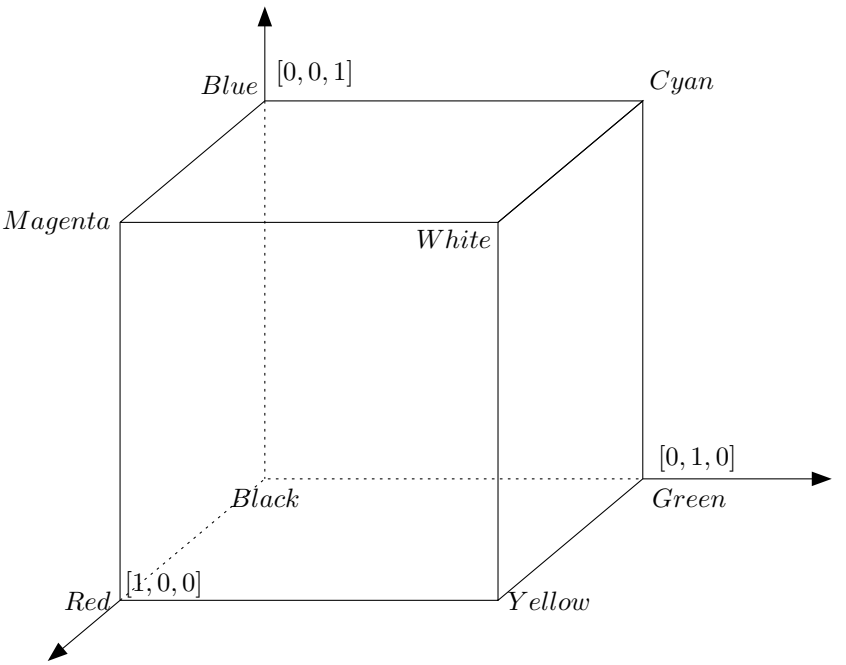

Figure 5-1 RGB Color Cube

In this model, we express a color C in the vector form,

$$C = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \qquad 0 \leq R, G, B \leq 1 \tag{5.1}$$

In some other notations, the authors may consider R, G, and B as three unit vectors like the three spatial unit vectors **i**, **j**, and **k**. Just as a spatial vector $\mathbf{V}$ can be expressed as $\mathbf{V} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$, any color is expressed as $C = (rR + gG + bB)$, and the red, green, blue intensities are specified by the values of r, g, and b respectively. In our notation here, R, G, and B may represent the intensity values of the color components. The next few sections discuss in more detail the color representation of various standards.

Suppose we have two colors C_1 and C_2 given by

$$C_1 = \begin{pmatrix} R_1 \\ G_1 \\ B_1 \end{pmatrix}, \qquad C_2 = \begin{pmatrix} R_2 \\ G_2 \\ B_2 \end{pmatrix}$$

Does it make sense to add these two colors to produce a new color C? For instance, consider

$$C = C_1 + C_2 = \begin{pmatrix} R_1 + R_2 \\ G_1 + G_2 \\ B_1 + B_2 \end{pmatrix}$$

You may immediately notice that the sum of two components may give a value larger than 1 which lies outside the color cube and thus does not represent any color. Just like adding two points in space is illegitimate, we cannot arbitrarily combine two colors. A linear combination of colors makes sense only if the sum of the coefficients is equal to 1. Therefore, we can have

$$C = \alpha_1 C_1 + \alpha_2 C_2 \tag{5.2}$$

when

$$0 \leq \alpha_1, \alpha_2 \quad \textit{and} \quad \alpha_1 + \alpha_2 = 1$$

In this way, we can guarantee that the resulted components will always lie within the color cube as each value will never exceed one. For example,

$$R = \alpha_1 R_1 + \alpha_2 R_2 \leq \alpha_1 \times 1 + \alpha_2 \times 1 = 1$$

which implies

$$R \leq 1$$

The linear combination of colors described by Equation (5.2) is called *color blending*.

5.3 Color Systems

In the RGB model described above, a given color is a point in a color cube as shown in Figure 5-1, and can be expressed as

$$C = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \qquad 0 \leq R, G, B \leq 1$$

However, RGB systems do not produce identical perceptions and they vary significantly from one to another. For example, suppose we have a yellow color described by the triplet (0.9, 0.8, 0.0). If we feed these values to a CRT and a film image recorder, we shall see different colors, even though in both cases the red is 90 percent of the maximum, the green is 80 percent of the maximum, and there is no blue. The reason is that the CRT phosphors and the film dyes have different color distribution responses. Consequently, the range of displayable colors (or the color **gamut**) is different in each case.

Different organizations have different interests and emphasis on color models. For example, the graphics community is interested in device-independent graphics; it will be a burden for them to develop graphics APIs that address the real differences among display properties. Fortunately, this bas been addressed in colorimetery literature, and standards exist for many common color systems. For example, the National Television System Committee (NTSC) defines an RGB system which forms the basis for many CRT systems. We can view the differences in color systems as the differences between various coordinate system for representing the tristimulus values. For example, if

$$C_1 = \begin{pmatrix} R_1 \\ G_1 \\ B_1 \end{pmatrix}, \quad \text{and} \quad C_2 = \begin{pmatrix} R_2 \\ G_2 \\ B_2 \end{pmatrix} \tag{5.3}$$

are the representations of the same color in two different systems, we can find a 3×3 color conversion matrix M such that

$$C_2 = MC_1 \tag{5.4}$$

Regardless of the way we find this matrix, it allows us to produce similar displays on different color systems.

However, this is not a good approach because the color gamuts of two systems may not be the same; even after the conversion of the color components from one system to another, the color may not be producible on the second system. Also, the printing and graphic arts industries use a four color subtractive system (CMYK) that includes black (K) as the fourth primary. Moreover, the linear color theory is only an approximation to human perception of colors. The distance between two points in the color cube does not necessarily measure how far apart the colors are perceptually. For example, humans are particularly sensitive to color shifts in blue.

The International Commission on Illumination, referred to as the CIE (Commission Internationale de l'Eclarage) defined in 1931 three standard primaries, which are actually imaginary colors. CIE defined the three standard primaries mathematically with positive color-matching functions shown in Figure 5-2. If the spectral power distribution (SPD) for a colored object is weighted by the curves of Figure 5-2, the CIE chromaticity coordinates can be calculated. This provides an international standard definition of all colors. The CIE primaries also eliminate negative-value color-matching and other problems related to the

selection of a set of real primaries.

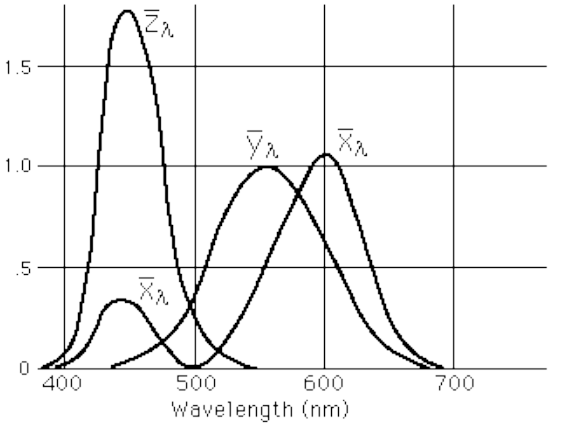

Figure 5-2 Matching functions of the three CIE primaries

5.3.1 The XYZ Color Model

The set of CIE primaries defines a color model that is in general referred to as the **XYZ color model**, where parameters X, Y, and Z represent the tristimulus values, the amount of each CIE primary required to produce a given color. The tristimulus values do not correspond to real colors, but they do have the property that any real color can be represented as a positive combination of them. Therefore, an RGB model describes a color in the same way as the XYZ model does. Actually, most color standards are based on this theoretical XYZ model. In this model, the Y primary is the luminance of the color and all colors can be represented by positive tristimulus values.

Due to the nature of the distribution of cones in the eye, the tristimulus values depend on the observer's field of view. To eliminate this variable, the CIE defined the standard (colorimetric) observer, which is characterized by three color matching functions. The color matching functions are the numerical description of the chromatic response of the observer. The three color-matching functions are referred to as $\overline{X}(\lambda), \overline{Y}(\lambda)$, and $\overline{Z}(\lambda)$, which can be thought of as the spectral sensitivity curves of three linear light detectors that yield the CIE XYZ tristimulus values X, Y, and Z. The tabulated numerical values of these functions are known collectively as the CIE standard observer.

The tristimulus values for a color with a spectral power distribution $I(\lambda)$ are given in terms of the standard observer by:

$$\begin{aligned} X &= \int_0^\infty \overline{X}_\lambda I(\lambda) d\lambda \\ Y &= \int_0^\infty \overline{Y}_\lambda I(\lambda) d\lambda \\ Z &= \int_0^\infty \overline{Z}_\lambda I(\lambda) d\lambda \end{aligned} \tag{5.5}$$

where λ is the wavelength of the equivalent monochromatic light.

A color can be specified by the tristimulus values, X, Y, and Z:

$$C = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{5.6}$$

We may also represent a color in the XYZ color space as an additive combination of the primaries using unit vectors $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$. Therefore, we can express Equation (5.6) as

$$C = X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z} \tag{5.7}$$

We can use 3×3 matrices to convert from XYZ color representation to representations in other standard systems. Also, it is convenient to normalize the X, Y, and Z values against the sum $X + Y + Z$, which is the total light energy. The normalized values are usually referred to as the **chromaticity coordinates**:

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z} \tag{5.8}$$

As $x + y + z = 1$, any color can be represented with just the x and y coordinates if the total energy is known. The parameters x and y depend only on hue and purity of the color and are called **chromaticity values**. Instead of using the total energy, people typically use the luminance Y and the chromaticity values x, and y to specify a color. The other two CIE values can be calculated as

$$X = \frac{x}{y}Y, \quad Z = \frac{z}{y}Y \tag{5.9}$$

where $z = 1 - x - y$. Using chromaticity coordinates (x, y), we can represent all colors on a two-dimensional diagram known as chromaticity diagram.

Color Gamuts

The gamut is the set of possible colors within a color system. No one system can reproduce all possible colors in the visible spectrum. It is not possible for a designer to create every color in the spectrum with either additive or subtractive colors. Both systems can reproduce a subset of all visible color, and while those subsets generally overlap, there are colors which can be reproduced with additive color and not with subtractive color and vice versa.

5.3.2 YUV Color Model

While the RGB color model is well-suited for displaying color images on a computer screen, it is not an effective model for image processing or video compression. This is because the human visual system (HVS) is more sensitive to luminance (brightness) than to colors. Therefore, it is more effective to represent a color image by separating the luminance from the color information and representing luma with a higher resolution than color.

The YUV color model, defined in the TV standards, is an efficient way of representing color images by separating brightness from color values. Historically, YUV color space was developed to provide compatibility between color and black /white analog television systems; it is not defined precisely in the technical and scientific literature. In this model, Y is the luminance (luma) component, which is the same as the Y component in the CIE

XYZ color space, and U and V are the color differences known as chrominance or chroma, which is defined as the difference between a color and a reference white at the same luminance. The conversion from RGB to YUV is given by the following formulas:

$$\begin{aligned} Y &= k_r R + k_g G + k_b B \\ U &= B - Y \\ V &= R - Y \end{aligned} \tag{5.10}$$

with

$$\begin{aligned} &0 \leq k_r, k_b, k_g \\ &k_r + k_b + k_g = 1 \end{aligned} \tag{5.11}$$

Note that equations (5.10) and (5.11) imply that $0 \leq Y \leq 1$ if the R, G, B components lie within the unit color cube. However, U and V can be negative. Typically,

$$k_r = 0.299, k_g = 0.587, k_b = 0.114 \tag{5.12}$$

which are values used in some TV standards.

The complete description of an image is specified by Y (the luminance component) and the two color differences (chrominance) U and V. If the image is black-and-white, $U = V = 0$. Note that we do not need another difference ($G-Y$) for the green component because that would be redundant. We can consider (5.10) as three equations with three unknowns, R, G, and B. We can always solve for the three unknowns and recover R, G, and B. A fourth equation is not necessary.

It seems that there is no advantage of using YUV over RGB to represent an image as both system requires three components to specify an image sample. However, as we mentioned earlier, human eyes are less sensitive to color than to luminance. Therefore, we can represent the U and V components with a lower resolution than Y and the reduction of the amount of data to represent chrominance components will not have an obvious effect on visual quality. Representing chroma with less number of bits than luma is a simple but effective way of compressing an image.

The conversion from RGB space to YUV space can be also expressed in matrix form:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{5.13}$$

The conversion from YUV space to RGB space using matrix is accomplished with the inverse transformation of (5.13):

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -0.509 & -0.194 \end{pmatrix} \begin{pmatrix} Y \\ U \\ V \end{pmatrix} \tag{5.14}$$

5.3.3 YCbCr Color Model

The YCbCr color model defined in the standards of ITU (International Telecommunication Union) is closely related to YUV but with the chrominace components scaled and shifted to ensure that they lie within the range 0 and 1. It is sometimes abbreviated to YCC. It is

also used in the JPEG and MPEG standards. In this model, an image sample is specified by a luminance (Y) component and two chrominance components (Cb, and Cr). The following equations convert an RGB image to one in YCbCr space.

$$\begin{aligned} Y &= k_r R + k_g G + k_b B \\ C_b &= \frac{B - Y}{2(1 - k_b)} + 0.5 \\ C_r &= \frac{(R - Y)}{2(1 - k_r)} + 0.5 \\ k_r &+ k_b + k_g = 1 \end{aligned} \tag{5.15}$$

An image may be captured in the RGB format and then converted to YCbCr to reduce storage or transmission requirements. Before displaying the image, it is usually necessary to convert the image back to RGB. The conversion from YCbCr to RGB can be done by solving for R, G, B in the equations of (5.15). The equations for converting from YCbCr to RGB are shown below:

$$\begin{aligned} R &= Y + (2C_r - 1)(1 - k_r) \\ B &= Y + (2C_b - 1)(1 - k_b) \\ G &= \frac{Y - k_r R - k_b B}{k_g} \\ &= Y - \frac{k_r(2C_r - 1)(1 - k_r) + k_b(2C_b - 1)(1 - k_b)}{k_g} \end{aligned} \tag{5.16}$$

If we use the ITU standard values $k_b = 0.114, k_r = 0.299, k_g = 1 - k_b - k_r = 0.587$ for (5.15) and (5.16), we will obtain the following commonly used conversion equations.

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ C_b &= 0.564(B - Y) + 0.5 \\ C_r &= 0.713(R - Y) + 0.5 \\ \\ R &= Y + 1.402C_r - 0.701 \\ G &= Y - 0.714C_r - 0.344C_b + 0.529 \\ B &= Y + 1.772C_b - 0.886 \end{aligned} \tag{5.17}$$

In equations (5.15), it is obvious that $0 \leq Y \leq 1$ as $0 \leq R, G, B \leq 1$. It turns out that the chrominance components C_b and C_r defined in (5.15) also always lie within the range [0, 1]. We prove this for the case of C_b. From (5.15), we have

$$\begin{aligned} C_b &= \frac{B-Y}{2(1-k_b)} + \frac{1}{2} \\ &= \frac{B - k_r R - k_g G - k_b B + 1 - k_b}{2(1-k_b)} \\ &= \frac{B}{2} + \frac{-k_r R - k_g G + 1 - k_b}{2(1-k_b)} \\ &\geq \frac{B}{2} + \frac{-k_r \times 1 - k_g \times 1 + 1 - k_b}{2(1-k_b)} \\ &= \frac{B}{2} \\ &\geq 0 \end{aligned}$$

Thus

$$C_b \geq 0 \tag{5.18}$$

Also,

$$\begin{aligned} C_b &= \frac{B-Y}{2(1-k_b)} + \frac{1}{2} \\ &= \frac{B - k_r R - k_g G - k_b B}{2(1-k_b)} + \frac{1}{2} \\ &\leq \frac{B - k_b B}{2(1-k_b)} + \frac{1}{2} \\ &= \frac{B}{2} + \frac{1}{2} \\ &\leq \frac{1}{2} + \frac{1}{2} \\ &= 1 \end{aligned}$$

Thus

$$C_b \leq 1 \tag{5.19}$$

Combining (5.18) and (5.19), we have

$$0 \leq C_b \leq 1 \tag{5.20}$$

Similarly

$$0 \leq C_r \leq 1 \tag{5.21}$$

In summary, we have the following situation.

$$\begin{aligned} &\text{If} \quad 0 \leq R, G, B \leq 1 \\ &\text{then} \quad 0 \leq Y, C_b, C_r \leq 1 \end{aligned} \tag{5.22}$$

Note that the converse is not true. That is, if $0 \leq Y, C_b, C_r \leq 1$, it does **not** imply $0 \leq R, G, B \leq 1$. A knowledge of this helps us in the implementations of the conversion from RGB to YCbCr and vice versa. We mentioned earlier that the eye can only resolve about 200 different intensity levels of each of the RGB components. Therefore, we can quantize all the RGB components in the interval [0,1] to 256 values, from 0 to 255, which can be represented by one byte of storage without any loss of visual quality. In other words, one byte (or an 8-bit unsigned integer) is enough to represent all the values of each RGB component. When we convert from RGB to YCbCr, it only requires one 8-bit unsigned integer to represent each YCbCr component. This implicitly implies that all conversions can be done efficiently in integer arithmetic.

5.4 RGBA Color Model

A computer monitor displays different amounts of red (R), green (G), and blue (B) light at each pixel position. The R, G, and B values form a certain color. These values are usually packed together and the packed value is referred to as the **RGB** value. Very often, an RGB value is packed with another value called **alpha** value to form the **RGBA** value. The alpha (α) value is used for color blending and denotes the degree of transparency of the associated pixel or object. If alpha is 1, the object is opaque and blocks objects behind it; if alpha is 0, the object is totally transparent and cannot be seen; if its value is between 0 and 1, the object is translucent. When initializing an OpenGL program, we can select the RGB mode with the command

glutInitDisplayMode (GLUT_RGB);

or the RGBA mode using

glutInitDisplayMode (GLUT_RGBA);

In OpenGL, the alpha value has meaning only if we enable color blending which is enabled by the command

glEnable(GL_BLEND);

The default is that all objects are opaque. We can specify an RGB value or an RGBA value using

glColor3f(r, g, b);

or

glColor4f(r, g, b, a);

All the values r, g, b, a lie within the range $[0, 1]$. That is, $0 \leq r, g, b, a \leq 1$. We may also use array specifications such as

```
float a[3] = { 1, 1, 0 };
glColor3fv( a );
```

or

```
float a4[4] = { 1, 1, 0, 1 };
glColor4fv( a4 );
```

We can assign an OpenGL color specification to individual vertices within the **glBegin/glEnd** pairs.

OpenGL represents color information in floating-point format internally. We can specify color values in integer format, but they will be converted automatically to floating-point values. For example, we can specify a color with an unsigned byte using the command,

```
glColor4ub( 0, 255, 0, 1);
```

which is equivalent to

```
glColor4f( 0.0, 1.0, 0.0, 1.0);
```

As we have discussed before, our eye can only resolve about 200 values for each of the primary colors. Therefore, most applications do not need to use other integer formats such as **unsigned int** (ui) which is a 32-bit integer format to specify a color component.

5.5 Color Blending

When we render only opaque polygons, the z-buffer for hidden surface removal is enough to render objects properly. However, when an OpenGL program runs in RGBA mode and **blending** is enabled, the alpha (A) value controls how RGB values are written into the frame buffer; fragments from different objects are combined to form the color of the same pixel, and we say that we **blend** or **composite** these objects together. (A fragment is all the data in a location of the frame buffer necessary to generate a pixel.) In the blending process, we combine the color value of the fragment being processed (source color) with that of the pixel already stored in the frame buffer (destination color). The combined color is put back to the same pixel location of the frame buffer. The new destination color is the combination of the old destination color and the source color. This process is shown in Figure 5-3. Blending occurs after our scene has been rasterized and converted to fragments.

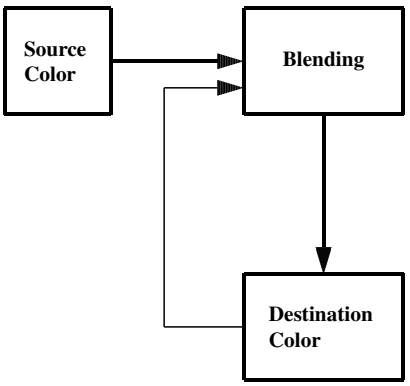

Figure 5-3 Color Blending

The alpha component (A) of an RGBA quantity is a measure of the *opacity* of a surface, which is the percentage of light that is blocked by the surface. An opacity of 1 (A = 1) indicates that the surface is totally opaque and blocks all light incident on it. An opacity of 0 (A = 0) corresponds to a totally transparent surface where all incident light passes

through it. The *transparency* or *translucency* of a surface is 1 - A. Therefore, an alpha value is meaningful only if it lies within the range $[0, 1]$.

In the rasterization process, polygons are usually rendered one at a time into the frame buffer. Without blending, each new fragment overwrites any existing color values in the frame buffer (color buffer), which holds the color for display. With blending, we need to apply opacity as part of fragment processing. Usually the polygon color that we are working on is considered as the **source color** and the color in the color buffer is the **destination color**. We combine the two colors using source and destination factors. Suppose we represent the source (polygon) colors, destination colors, source factors, and destination factors using 4-tuples (RGBA), C_s, C_d, S and D respectively :

$$C_s = \begin{pmatrix} R_s \\ G_s \\ B_s \\ A_s \end{pmatrix}, \quad C_d = \begin{pmatrix} R_d \\ G_d \\ B_d \\ A_d \end{pmatrix}, \quad S = \begin{pmatrix} S_r \\ S_g \\ S_b \\ S_a \end{pmatrix}, \quad D = \begin{pmatrix} D_r \\ D_g \\ D_b \\ D_a \end{pmatrix} \tag{5.23}$$

Then the blended color (new destination color) C'_d is given by

$$C'_d = \begin{pmatrix} R_s S_r + R_d D_r \\ G_s S_g + G_d D_g \\ B_s S_b + B_d D_b \\ A_s S_a + A_d D_a \end{pmatrix} \tag{5.24}$$

Each component is clamped to [0, 1].

The next question is how to generate the blending factors. OpenGL provides the function **glBlendFunc**() to do so. Before using this function, we need to enable blending using the command

```
glEnable(GL_BLEND);
```

(We can disable blending using "glDisable(GL_BLEND);".) After blending has been enabled, we can set up the source and destination factors by the command

```
glBlendFunc( sourceFactor, destinationFactor );
```

where $sourceFactor$ and $destinatiionFactor$ are type **GLenum**. OpenGL defines a number of blending factors, including the values 1 (GL_ONE) and 0 (GL_ZERO), the source alpha S_a and $1 - S_a$ (GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA), and the destination alpha D_a and $1 - D_a$ (GL_DST_ALPHA and GL_ONE_MINUS_DST_ALPHA). Table 5-1 shows the source and destination factors that the function **glBlendFunc**() can take as parameters.

We can also specify a (R_c, G_c, B_c, A_c) using the **glBlendColor**() function. For example, suppose we select GL_SRC_ALPHA for the source blending factor and GL_ONE_MINUS_SRC_ALPHA for the destination blending factor, then the new destination color is given by

$$C'_d = \begin{pmatrix} R'_d \\ G'_d \\ B'_d \\ A'_d \end{pmatrix} = \begin{pmatrix} R_s S_a + R_d(1 - S_a) \\ G_s S_a + G_d(1 - S_a) \\ B_s S_a + B_d(1 - S_a) \\ A_s S_a + A_d(1 - S_a) \end{pmatrix} \tag{5.25}$$

Actually, this is one of the most commonly used options in computing the composite color.

Note that unlike most OpenGL functions that users do not need to worry about the order in which polygons are rasterized, the effect of color blending depends on the order of rendering polygons. The destination and source factors could be interchanged if the order of rendering two polygons is interchanged.

Table 5-1 Blending Factors

GL Constant	**Computed Blend Factor**
GL_ZERO	(0, 0, 0, 0)
GL_ONE	(1, 1, 1, 1)
GL_DST_COLOR	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1, 1)-(R_s, G_s, B_s, A_s)
GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1)-(A_s, A_s, A_s, A_s)
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1, 1)-(A_d, A_d, A_d, A_d)
GL_SRC_ALPHA_SATURATE	(f, f, f, 1); f=min(A_s, 1-A_d)
GL_CONSTANT_COLOR	(R_c, G_c, B_c, A_c)
GL_CONSTANT_ALPHA	(A_c, A_c, A_c, A_c)

Example 5-1

This example draws two overlapping colored triangles, one red and one green, each with an alpha value of 0.8. The background is white. Blending is enabled and the parameters for source and destination blending factors for the function **glBlendFunc**() are set to GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA respectively. When the program starts up, a red triangle is drawn on the left and then a green triangle is drawn on the right with part of it overlapping with the red triangle as shown in case (a) of Figure 5-4. Case (b) of Figure 5-4 shows a similar drawing except that the green triangle is drawn on the left first.

For case (a), when the red triangle is first drawn, its color $C_r = (1, 0, 0, 0.8)$, which is the source, is blended with the background white color $C_w = (1, 1, 1, 1)$, which is the original destination color to give the new destination color C_d. In this case, the source blending factor is 0.8 (source alpha), and the destination blending factor is 0.2 (one minus source alpha). Thus

$$C_d = \begin{pmatrix} 1 \times 0.8 + 1 \times 0.2 \\ 0 \times 0.8 + 1 \times 0.2 \\ 0 \times 0.8 + 1 \times 0.2 \\ 0.8 \times 0.8 + 1 \times 0.2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.2 \\ 0.2 \\ 0.84 \end{pmatrix} \tag{5.26}$$

When the green triangle is drawn on the right side, besides the white background, we have three different regions, the nonoverlapping 'red' region on the left, the 'red-green' overlapping region at the center and the nonoverlapping 'green' region as shown in Figure 5-4(a). The color of the nonoverlapping 'red' region is given by (5.26). Similarly, the color of the nonoverlapping 'green' region is $(0.2, 1, 0.2, 0.84)$. To calculate the color of the overlapping region, we use $C_g = (0, 1, 0, 0.8)$ as the source color and $C_d = (1, 0.2, 0.2, 0.84)$ as the destination color; the new destination color is given

by

$$C_d^{(a)} = \begin{pmatrix} 0 \times 0.8 + 1 \times 0.2 \\ 1 \times 0.8 + 0.2 \times 0.2 \\ 0 \times 0.8 + 0.2 \times 0.2 \\ 0.8 \times 0.8 + 0.84 \times 0.2 \end{pmatrix} = \begin{pmatrix} 0.2 \\ 0.84 \\ 0.04 \\ 0.808 \end{pmatrix} \tag{5.27}$$

We see that green is the dominant component, followed by red in the overlapping region.

In case (b), the green triangle is drawn first. The color for the overlapping region can be similarly calcualted:

$$C_d^{(b)} = \begin{pmatrix} 0.84 \\ 0.2 \\ 0.04 \\ 0.808 \end{pmatrix} \tag{5.28}$$

In this case, red is the dominant color followed by green.

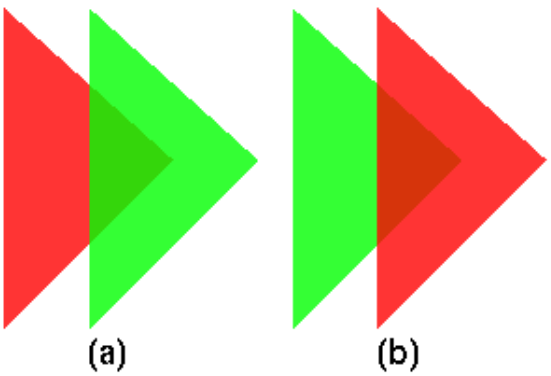

Figure 5-4 Color Blending of Example 5-1

The following is the code for this example.

```
static void init(void)
{
  glEnable (GL_BLEND);
  glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glShadeModel (GL_FLAT);
  glClearColor (1.0, 1.0, 1.0, 1.0);
}

static void triangle( float color[4] )
{
  glColor4fv( color );
  glBegin (GL_TRIANGLES);
    glVertex3f(-2, -1.9, 0.0);
    glVertex3f( -1, -0.9, 0.0);
    glVertex3f( -2,  0, 0.0);
  glEnd();
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glOrtho (-2.2, 2.2, -2.2, 2.2, -2.0, 10.0);
```

```
    //draw red triangle first, then green
    float red[4] = { 1, 0, 0, 0.8};
    triangle( red );
    glTranslatef( 0.5, 0, 0 );
    float green[4] = { 0, 1, 0, 0.8};
    triangle( green );

    //draw green triangle first, then red
    glTranslatef( 1.2, 0, 0 );
    triangle ( green );
    glTranslatef( 0.5, 0, 0 );
    triangle ( red );

    glFlush();
}
```

Note that when we enable blending, we usually do not enable hidden surface removal. This is because objects behind any object already rendered would not be rasterized and thus its color would not combine with the frame buffer color. If a scene consists of both opaque and transparent objects, any object behind and opaque one should not be rendered, but translucent objects in front of opaque objects should be composited. A simple solution to this problem is to enable hidden-surface removal as usual and make the z buffer read-only for any object that is translucent. This can be done using the command

glDepthMask(GL_FALSE);

When the z buffer is read-only, a translucent object lying behind any opaque object already rendered is discarded. On the other hand, a translucent object lying in front of a rendered opaque object will be blended; however, as the z buffer is read-only, the depth value in the buffer will not be changed. When we render opaque objects, we set the depth mask to true (so that the z buffer is writable) and render them as usual. In summary, when drawing translucent objects, we enable depth buffering but make the depth buffer read-only and draw the objects in the following order:

1. First draw all opaque objects, with depth buffer in normal operation.
2. Preserve the depth values by making depth buffer read-only.
3. Draw the translucent objects only if they are in front of the opaque ones, and blend them with the opaque ones.

Finally, one should note that in some systems, the hardware frame buffer may not fully support blending and the blending result may not be exactly the same as what we expect.

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273