Color and Blending

Prerequisites

None; this chapter discusses color issues from first principles.

Introduction

Color is a fundamental concept for computer graphics. We need to be able to define colors for our graphics that represent good approximations of real-world colors, and we need to be able to manipulate colors as we develop our applications.

There are many ways to specify colors, but all depend principally on the fact that the human visual system generally responds to colors through the use of three kinds of cells in the retina of the eye. This response is complex and includes both physical and psychological processes, but the fundamental fact of three kinds of stimulus is maintained by all the color models in computer graphics. For most work, the usual model is the RGB (Red, Green, Blue) color model that matches in software the physical design of computer monitors, which are made with a pattern of three kinds of phosphor which emit red, green, and blue light when they are excited by an electron beam. This RGB model is used for color specification in almost all computer graphics APIs, and it is the basis for the discussion here. There are a number of other models of color, and we refer you to textbooks and other sources, especially Foley et al. [FvD], for additional discussions on color models and for more complete information on converting color representations from one model to another.

Because the computer monitor uses three kinds of phosphor, and each phosphor emits light levels based on the energy of the electron beam that is directed at it, a common approach is to specify a color by the level of each of the three primaries. These levels are a proportion of the maximum light energy possible for that primary, so an RGB color is specified by a triple \((r, g, b)\) where each of the three components represents the amount of that particular component in the color. This proportion can be represented by a real number between 0.0 and 1.0, inclusive. Alternately, each color component can be represented by an integer that depends on the color depth available for the system; if you have eight bits of color for each component, the integers would be in the range 0 to 255. The real-number approach is used more commonly in graphics APIs because it is more device-independent. In either case, the number represents the proportion of the available color of that primary hue that is desired for the pixel. Thus the higher the number for a component, the brighter the light in that color, so with the real-number representation, black is represented by \((0.0, 0.0, 0.0)\) and white by \((1.0, 1.0, 1.0)\). The RGB primaries are represented respectively by \((1.0, 0.0, 0.0)\) — red, \((0.0, 1.0, 0.0)\) — green, and \((0.0, 0.0, 1.0)\) — blue; that is, colors that are fully bright in a single primary component and totally dark in the other primaries. Other colors are a mix of the three primaries as needed.

Internally, most graphics hardware deals with colors using integers. Floating-point values are converted to integers to save space and to speed operations, with the exact representation and storage of the integers depending on the number of bits per color per pixel and on other hardware design issues. This distinction sometimes comes up in considering details of color operations in your API, but is generally something that can be ignored. The color-generation process itself is surprisingly complex because the viewing device must generate perceptually-linear values, but most hardware generates color with exponential, not linear, properties. All these color issues are hidden from the API programmer, however, and are managed after being translated from the API representations of the colors; allowing API-based programs to work relatively the same across a wide range of platforms.
In addition to dealing with the color of light, modern graphics systems add a fourth component to the question of color. This fourth component is called “the alpha channel” because that was its original notation [POR], and it represents the opacity of the material that is being modeled. As is the case with color, this is represented by a real number between 0.0 (no opacity — completely transparent) and 1.0 (completely opaque — no transparency). This is used to allow you to create objects that you can see through at some level, and can be a very valuable tool when you want to be able to see more than just the things at the front of a scene. However, transparency is not determined globally by the graphics API; it is determined by compositing the new object with whatever is already present in the Z-buffer. Thus if you want to create an image that contains many levels of transparency, you will need to pay careful attention to the sequence in which you draw your objects, drawing the furthest first in order to get correct attenuation of the colors of background objects.

Definitions

The RGB cube

The RGB color model is associated with a geometric presentation of a color space. That space is a cube consisting of all points \((r, g, b)\) with each of \(r\), \(g\), and \(b\) having a value that is a real number between 0 and 1. Because of the easy analogy between color triples and space triples, every point in the unit cube can be easily identified with a RGB triple representation of a color. This gives rise to the notion of the RGB color cube that is seen in every graphics text (and thus that we won't repeat here).

To illustrate the numeric properties of the RGB color system, we will create the edges of the color cube as shown in Figure 3.1 below, which has been rotated to illustrate the colors more fully. To do this, we create a small cube with a single color, and then draw a number of these cubes around the edge of the geometric unit cube, with each small cube having a color that matches its location. We see the origin \((0,0,0)\) corner, farthest from the viewer, mostly by its absence because of the black background, and the \((1,1,1)\) corner nearest the viewer as white. The three axis directions are the pure red, green, and blue corners. Creating this figure is discussed below in the section on creating a model with a full spectrum of colors, and it would be useful to add an interior cube within the figure shown that could be moved around the space interactively and would change color to illustrate the color at its current position in the cube.

![Figure 3.1: tracing the colors of the edges of the RGB cube](image)

This figure suggests the nature of the RGB cube, but a view of the entire RGB cube is provided by the Figure 3.2, showing the cube from two points of view (from the white vertex and from the black vertex) so you can see the entire gamut of colors on the surface of the cube. Note that the
three vertices closest to the white vertex are the cyan, magenta, and yellow vertices, while the three vertices closest to the black vertex are the red, green, and blue vertices. This illustrates the additive nature of the RGB color model, with the colors getting lighter as the amounts of the primary colors increases. This will be explored later and will be contrasted to the subtractive nature of other color models. Not shown is the center diagonal of the RGB cube from \((0, 0, 0)\) to \((1, 1, 1)\) that corresponds to the colors with equal amounts of each primary; these are the gray colors that provide the neutral backgrounds that are very useful in presenting colorful images.

Figure 3.2: two views of the RGB cube — from white (left) and black (right)

Color is ubiquitous in computer graphics, and we can specify color in two ways: directly for the objects we are drawing, or by defining properties of object surfaces and lights. In this module we only think about the colors of objects, and save the color of lights and the way light interacts with surfaces for a later module on lighting. In general, the behavior of a scene will reflect both these attributes — if you have a red object and illuminate it with a blue light, your object will seem to be black, because a red object reflects no blue light and the light contains no other color than blue.

**Luminance**

Luminance of a color is the color’s brightness, or the intensity of the light it represents, without regard for its actual color. This concept is particularly meaningful for emissive colors on the screen, because these actually correspond to the amount of light that is emitted from the screen. The concept of luminance is important for several reasons. One is that a number of members of any population have deficiencies in the ability to distinguish different colors, the family of so-called color blindness problems, but are able to distinguish differences in luminance. You need to take luminance into account when you design your displays so that these persons can make sense of them. Luminance is also important because part of the interpretation of an image deals with the brightness of its parts, and you need to understand how to be sure that you use colors with the right kind of luminance for your communication. For example, in the chapter on visual communication we will see how we can use luminance information to get color scales that are approximately uniform in terms of having the luminance of the color represent the numerical value that the color is to represent.

For RGB images, luminance is quite easy to compute. Of the three primaries, green is the brightest and so contributes most to the luminance of a color. Red is the next brightest, and blue is the least bright. The actual luminance will vary from system to system and even from display device to display device because of differences in the way color numbers are translated into voltages and because of the way the phosphors respond. In general, though, we are relatively accurate if we assume that luminance is calculated by the formula

\[
0.59 \times \text{green} + 0.30 \times \text{red} + 0.11 \times \text{blue}
\]

so the overall brightness ratios are approximately 6:3:1 for green:red:blue.
Other color models

There are times when the RGB model is not easy to use. Few of us think of a particular color in terms of the proportions of red, green, and blue that are needed to create it, so there are other ways to think about color that make this more intuitive. And there are some processes for which the RGB approach does not model the reality of color production. So we need to have a wider range of ways to model color to accommodate these realities.

A more intuitive approach to color is found with either of the HSV (Hue-Saturation-Value) or HLS (Hue-Lightness-Saturation) models. These models represent color as a hue (intuitively, a descriptive variation on a standard color such as red, or magenta, or blue, or cyan, or green, or yellow) that is modified by setting its value (a property of darkness or lightness) and its saturation (a property of brightness). This lets us find numerical ways to say “the color should be a dark, vivid reddish-orange” by using a hue that is to the red side of yellow, has a relatively low value, and has a high saturation.

Just as there is a geometric model for RGB color space, there is one for HSV color space: a cone with a flat top, as shown in Figure 3.3 below. The distance around the circle in degrees represents the hue, starting with red at 0, moving to green at 120, and blue at 240. The distance from the vertical axis to the outside edge represents the saturation, or the amount of the primary colors in the particular color. This varies from 0 at the center (no saturation, which makes no real coloring) to 1 at the edge (fully saturated colors). The vertical axis represents the value, from 0 at the bottom (no color, or black) to 1 at the top. So a HSV color is a triple representing a point in or on the cone, and the “dark, vivid reddish-orange” color would be something like (40.0, 1.0, 0.7). Code to display this geometry interactively is discussed at the end of this chapter, and writing an interactive display program gives a much better view of the space.

The shape of the HSV model space can be a bit confusing. The top surface represents all the lighter colors based on the primaries, because colors getting lighter have the same behavior as colors getting less saturated. The reason the geometric model tapers to a point at the bottom is that there is no real color variation near black. In this model, the gray colors are the colors with a saturation of 0, which form the vertical center line of the cone. For such colors, the hue is meaningless, but it still must be included.

In the HLS color model, shown in Figure 3.4, the geometry is much the same as the HSV model but the top surface is stretched into a second cone. Hue and saturation have the same meaning as HSV but lightness replaces value, and lightness corresponds to the brightest colors at a value of 0.5. The rationale for the dual cone that tapers to a point at the top as well as the bottom is that as colors get lighter, they lose their distinctions of hue and saturation in a way that is very analogous
with the way colors behave as they get darker. In some ways, the HLS model seems to come
closer to the way people talk about “tints” and “tones” when they talk about paints, with the
strongest colors at lightness 0.5 and becoming lighter (tints) as the lightness is increased towards
1.0, and becoming darker (tones) as the lightness is decreased towards 0.0. Just as in the HSV
case above, the grays form the center line of the cone with saturation 0, and the hue is
meaningless.

![HLS Double Cone](image)

Figure 3.4: the HLS double cone from the red (left), green (middle), and blue(right) directions.

The top and bottom views of the HLS double cone look just like those of the HSV single cone, but
the side views of the HLS double cone are quite different. Figure 3.4 shows the HLS double cone
from the three primary-color sides: red, green, and blue respectively. The views from the top or
bottom are exactly those of the HSV cone and so are now shown here. The images in the figure do
not show the geometric shape very well; the discussion of this model in the code section below will
show you how this can be presented, and an interactive program to display this space will allow
you to interact with the model and see it more effectively in 3-space.

There are relatively simple functions that convert a color defined in one space into the same color as
defined in another space. We do not include all these functions in these notes, but they are covered
in [FvD], and the functions to convert HSV to RGB and to convert HLS to RGB are included in
the code discussions below about producing these figures.

All the color models above are based on colors presented on a computer monitor or other device
where light is emitted to the eye. Such colors are called emissive colors, and operate by adding
light at different wavelengths as different screen cells emit light. The fact that most color presented
by programming comes from a screen makes this the primary way we think about color in
computer graphics systems. This is not the only way that color is presented to us, however.
When you read these pages in print, and not on a screen, the colors you see are generated by light
that is reflected from the paper through the inks on the page. Such colors can be called
transmissive colors and operate by subtracting colors from the light being reflected from the page.
This is a totally different process and needs separate treatment. Figure 3.5 below illustrates this
principle. The way the RGB add to produce CMY and eventually white shows why emissive
colors are sometimes called additive colors, while the way CMY produce RGB and eventually
black shows why transmissive colors are sometimes called subtractive colors.
Transmissive color processes use inks or films that transmit only certain colors while filtering out all others. Two examples are the primary inks for printing and the films for theater lights; the primary values for transmissive color are cyan (which transmits both blue and green), magenta (which transmits both blue and red), and yellow (which transmits both red and green). In principle, if you use all three inks or filters (cyan, magenta, and yellow), you should have no light transmitted and so you should see only black. In practice, actual materials are not perfect and allow a little off-color light to pass, so this would produce a dark and muddy gray (the thing that printers call “process black”) so you need to add an extra “real” black to the parts that are intended to be really black. This cyan-magenta-yellow-black model is called CMYK color and is the basis for printing and other transmissive processes. It is used to create color separations that combine to form full-color images as shown in Figure 3.6, which shows a full-color image (left) and the sets of yellow, cyan, black, and magenta separations (right-hand side, clockwise from top left) that are used to create plates to print the color image. We will not consider the CMYK model further in this discussion because its use is in printing and similar technologies, but not in graphics programming. We will meet this question again when we discuss graphics hardcopy, however.

Color depth

The numerical color models we have discussed above are device-independent; they assume that colors are represented by real numbers and thus that there are an infinite number of colors available.
to be displayed. This is, of course, an incorrect assumption, because computers lack the capability of any kind of infinite storage. Instead, computers use color capabilities based on the amount of memory allocated to holding color information.

The basic model we usually adopt for computer graphics is based on screen displays and can be called direct color. For each pixel on the screen we store the color information directly in the screen buffer. The number of bits of storage we use for each color primary is called the color depth for our graphics. At the time of this writing, it is probably most common to use eight bits of color for each of the R, G, and B primaries, so we often talk about 24-bit color. In fact, it is not uncommon to include the Z-buffer depth in the color discussion, with the model of RGBA color, and if the system uses an 8-bit Z-buffer we might hear that it has 32-bit color. This is not universal, however; some systems use fewer bits to store colors, and not all systems use an equal number of bits for each color, while some systems use more bits per color. The very highest-end professional graphics systems, for example, often use 36-bit or 48-bit color.

One important effect of color depth is that the exact color determined by your color computations will not be displayed on the screen. Instead, the color is aliased by rounding it to a value that can be represented with the color depth of your system. This can lead to serious effects called Mach bands, in which the differences between adjacent color representations can be perceived visually. Because the human visual system is extremely capable of detecting edges, when these differences can be seen they are interpreted as strong edges and are very disruptive of the image presentation. You should be careful to look for Mach banding in your work, and when you see it, you should try to modify your image to make it less visible. Figure 3.7 shows a small image that shows Mach bands.

![Image showing Mach banding](image.png)

Figure 3.7: an image showing Mach banding

**Color gamut**

Color is not only limited by the number of colors that can be displayed, but also by limitations in the technology of display devices. No matter what display technology you use — phosphates on a video-based screen, ink on paper, or LCD cells on a flat panel — there is a limited range of color that the technology can present. This is also true of more traditional color technologies, such as color film. The range of a device is called its color gamut, and we must realize that the gamut of our devices limits our ability to represent certain kinds of images. A significant discussion of the color gamut of different devices is beyond the scope of the content we want to include for the first course in computer graphics, but it is important to realize that there are serious limitations on the colors you can produce on most devices.
Color blending with the alpha channel

In most graphics APIs, color can be represented as more than just a RGB triple; it can also include a blending level (sometimes thought of as a transparency level) so that anything with this color will have a color blending property. Thus color is represented by a quadruple (r,g,b,a) and the color model that includes blending is called the RGBA model. The transparency level \( a \) for a color is called the *alpha value*, and its value is a number between 0.0 and 1.0 that is actually a measure of opacity instead of transparency. That is, if you use standard kinds of blending functions and if the alpha value is 1.0, the color is completely opaque, but in the same situation if the alpha value is 0.0, the color is completely transparent. However, we are using the term “transparent” loosely here, because the real property represented by the alpha channel is *blending*, not transparency. The alpha channel was invented to permit image image compositing [POR] in which an image could be laid over another image and have part of the underlying image show through. So while we may say “transparent” we really mean blended.

This difference between blended and transparent colors can be very significant. If we think of transparent colors, we are modeling the logical equivalent of colored glass. This kind of material embodies transmissive, not emissive, colors — only certain wavelengths are passed through, while the rest are absorbed. But this is not the model that is used for the alpha value; blended colors operate by averaging emissive RGB colors, which is the opposite of the transmissive model implied by transparency. The difference can be important in creating the effects you need in an image. There is an additional issue to blending because averaging colors in RGB space may not result in the intermediate colors you would expect; the RGB color model is one of the worse color models for perceptual blending but we have no real choice in most graphics APIs.

**Challenges in blending**

The concept of blending is based on the premise that the color of a pixel in a scene can reasonably be represented by a linear combination of the colors of partially-transparent objects that lie under that pixel. For many kinds of objects this is not unreasonable, but the premise is vulnerable to the effects of our RGB color model.

If the RGB color model were uniform, then we would perceive uniform changes in color as we move from one color to another in the visual space. In Figure 3.9 below, we create some color strips with well-known endpoints and ask ourselves if we believe that the color in the middle of the strips really is the average of the colors at the end.

![Figure 3.9: the color strips between red and cyan (top) and green and magenta (bottom)](image)

There is no real solution for this problem in the RGB color space or in any system based on it, but the problem can be addressed by doing color interpolation in another color space: converting the two RGB colors to their representations in the other space, interpolating in the other space, and converting the result back to RGB space. Sometimes the HSV or HLS spaces are used for this
kind of interpolation, but what is really needed is a perceptually-uniform color space, and these are
difficult to describe simply.

When we look at each of these color strips, we should consider the luminance of the colors along
the strip. In the top strip, the red end has luminance 0.3 while the cyan end has luminance 0.7, but
the middle gray point has luminance 0.5. In the bottom strip, the green end has luminance 0.59
while the magenta end has luminance 0.41, and again the middle gray point has luminance 0.5.
Thus these strips seem to have a fairly uniform change in luminance. A similar analysis could be
made for other kinds of color interpolation.

Another way to consider these color strips, however, is in terms of the amount of color (the
chrominance or chroma) they represent. The chroma can be seen as the saturation in the HLS or
HSV color space, and in these terms both the endpoints of the color strips have chroma 1 while the
middle points have chroma 0.5, so the strips do not interpolate chroma well. Interpolating colors
while preserving chroma would require interpolations in another color space, such as HSV or
HLS.

Color in OpenGL

OpenGL uses the RGB and RGBA color models with real-valued components. These colors
follow the RGB discussion above very closely, so there is little need for any special comments on
color itself in OpenGL. Instead, we will discuss blending in OpenGL and then will give some
examples of code that uses color for its effects.

Enabling blending

In order to use colors from the RGBA model, you must specify that you want the blending enabled
and you must identify the way the color of the object you are drawing will be blended with the
color that has already been defined. This is done with two simple function calls:

```c
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The first is a case of the general enabling concept for OpenGL; the system has many possible
capabilities and you can select those you want by enabling them. This allows your program to be
more efficient by keeping it from having to carry out all the possible operations in the rendering
pipeline. The second allows you to specify how you want the color of the object you are drawing
to be blended with the color that has already been specified for each pixel. If you use this blending
function and your object has an alpha value of 0.7, for example, then the color of a pixel after it
has been drawn for your object would be 70% the color of the new object and 30% the color of
whatever had been drawn up to that point.

There are many options for the OpenGL blending function. The one above is the most commonly
used and simply computes a weighted average of the foreground and background colors, where the
weight is the alpha value of the foreground color. In general, the format for the blending
specification is

```c
    glBlendFunc(src, dest)
```

and there are many symbolic options for the source (src) and destination (dest) blending values; the
OpenGL manual covers them all.

Modeling transparency with blending

Blending creates some significant challenges if we want to create the impression of transparency.
To begin, we make the simple observation that is something is intended to seem transparent to
some degree, you must be able to see things behind it. This suggests a simple first step: if you are
working with objects having their alpha color component less than 1.0, it is useful and probably
important to allow the drawing of things that might be behind these objects. To do that, you should draw all solid objects (objects with alpha component equal to 1.0) before drawing the things you want to seem transparent, and then turn off the depth buffer by disabling the depth test with `glDisable(GL_DEPTH_TEST)` before drawing items with blended colors, and enabling the depth test again after drawing them. This at least allows the possibility that some concept of transparency is allowed.

But it may not be enough to do this, and in fact this attempt at transparency may lead to more confusing images than leaving the depth test intact. Let us consider the case that you have three objects to draw, and that you will be drawing them in a particular order. For the discussion, let’s assume that the objects are numbered 1, 2, and 3, that they have colors C1, C2, and C3, that you draw them in the sequence 1, 2, and 3, that they line up from the eye but have a totally white background behind them, and that each color has alpha = 0.5. Let’s assume further that we are not using the depth buffer so that the physical ordering of the objects is not important. The layout is shown in Figure 3.8. And finally, let’s further assume that we’ve specified the blend functions as suggested above, and consider the color that will be drawn to the screen where these objects lie.

When we draw the first object, the frame buffer will have color C1; no other coloring is involved. When we draw the second object on top of the first, the frame buffer will have color 0.5*C1+0.5*C2, because the foreground (C2) has alpha 0.5 and the background (C1) is included with weight 0.5 = 1 - 0.5. Finally, when the third object is drawn on top of the others, the color will be 0.5*C3 + 0.5*(0.5*C1+0.5*C2), or 0.5*C3 + 0.25*C2 + 0.25*C1. That is, the color of the most recent object drawn is emphasized much more than the color of the other objects. This shows up clearly in the right-hand part of Figure 3.8 below, where the red square is drawn after the other two squares. On the other hand, if you had drawn object three before object 2, and object 2 before object 1, the color would have been 0.5*C1 + 0.25*C2 + 0.25*C3, so the order in which you draw things, not the order in which they are placed in space, determines the color.

But this again emphasizes a difference between blending and transparency. If we were genuinely modeling transparency, it would not make any difference which object were placed first and which last; each would subtract light in a way that is independent of its order. So this represents another challenge if you would want to create an illusion of transparency with more than one non-solid object.

*Some examples*

**Example: An object with partially transparent faces**

If you were to draw a piece of the standard coordinate planes and to use colors with alpha less than 1.0 for the planes, you would be able to see through each coordinate plane to the other planes as though the planes were made of some partially-transparent plastic. We have modeled a set of three squares, each lying in a coordinate plane and centered at the origin, and each defined as having a rather low alpha value of 0.5 so that the other squares are supposed to show through. In this section we consider the effects of a few different drawing options on this view.
The left-hand side of Figure 3.10 below shows the image we get with these colors when we leave the depth test active. Here what you can actually “see through” depends on the order in which you draw the objects. With the depth test enabled, the presence of a transparent object close to your eye prevents the writing of its blend with an object farther away. Because of this, the first coordinate plane you draw is completely opaque to the other planes, even though we specified it as being partly transparent, and a second coordinate plane allows you to see through it to the first plane but is fully opaque to the second plane. We drew the blue plane first, and it is transparent only to the background (that is, it is darker than it would be because the black background shows through). The green plane was drawn second, and that only allows the blue plane to show through. The red plane was drawn third, and it appears to be fully transparent and show through to both other planes. In the actual working example, you can use keypresses to rotate the planes in space; note that as you do, the squares you see have the same transparency properties in any position.

However, in the image in the center of Figure 3.10, we have disabled the depth test, and this presents a more problematic situation. In this case, the result is something much more like transparent planes, but the transparency is very confusing because the last plane drawn, the red plane, always seems to be on top because its color is the brightest. This figure shows that the OpenGL attempt at transparency is not necessarily a desirable property; it is quite difficult to get information about the relationship of the planes from this image. Thus one would want to be careful with the images one would create whenever one chose to work with transparent or blended images. Because this figure is created by exactly the same code as the one above with GL_BLEND disabled instead of enabled, the code for the example is not included here.

Finally, we change the alpha values of the three squares to account for the difference between the weights in the final three-color section. Here we use 1.0 for the first color (blue), 0.5 for the second color (green) but only 0.33 for red, and we see that this final image, the right-hand image in Figure 3.8, has the following color weights in its various regions:

- 0.33 for each of the colors in the shared region,
- 0.5 for each of blue and green in the region they share,
- 0.33 each for red and green in the region they share,
- 0.33 for red and 0.67 for blue in the region they share,
- the original alpha values for the regions where there is only one color.

Note that the “original alpha values” gives us a solid blue, a fairly strong green, and a weak red as stand-alone colors. This gives us a closer approximation to the appearance actual transparency for these three colors, with a particular attention to the clear gray in the area they all cover, but there are still some areas that don’t quite work. To get even this close, however, we must analyze the rendering carefully and we still cannot quite get a perfect appearance. The code for all these
examples is in the file alphacolor.c, which is presented with blending disabled and the adjusted values of alpha for the right-hand image in Figure 3.10.

A word to the wise...

Recall that the alpha value represents a blending proportion, not transparency. This blending is applied by comparing the color of an object with the current color of the image at a pixel, and coloring the pixel by blending the current color and the new color according to one of several rules that you can choose with $\text{glBlendFunc}(\ldots)$ as noted above. This capability allows you to build up an image by blending the colors of parts in the order in which they are rendered, and we saw that the results can be quite different if objects are received in a different order. Blending does not treat parts as being transparent, and so there are some images where OpenGL simply does not blend colors in the way you expect from the concept.

If you really do want to try to achieve the illusion of full transparency, you are going to have to do some extra work. You will need to be sure that you draw the items in your image starting with the item at the very back and proceeding to the frontmost item. This process is sometimes called Z-sorting and it can be tricky because objects can overlap or the sequence of objects in space can change as you apply various transformations to the scene. In the example above, the squares actually intersect each other and could only be sorted if each were broken down into four separate sub-squares. And even if you can get the objects sorted once, the order would change if you rotated the overall image in 3-space, so you would possibly have to re-sort the objects after each rotation. In other words, this would be difficult.

As always, when you use color you must consider carefully the information it is to convey. Color is critical to convey the relation between a synthetic image and the real thing the image is to portray, of course, but it can be used in many more ways. One of the most important is to convey the value of some property associated with the image itself. As an example, the image can be of some kind of space (such as interstellar space) and the color can be the value of something that occupies that space or happens in that space (such as jets of gas emitted from astronomical objects where the value is the speed or temperature of that gas). Or the image can be a surface such as an airfoil (an airplane wing) and the color can be the air pressure at each point on that airfoil. Color can even be used for displays in a way that carries no meaning in itself but is used to support the presentation, as in the Chromadepth™ display we will discuss below in the texture mapping module. But never use color without understanding the way it will further the message you intend in your image.

Code examples

Example: A model with parts having a full spectrum of colors

The code in the file cubecolors.c uses a technique you have not yet seen to create all the small cubes that make up the visible edges of the RGB cube. In this code, we use only a simple cube we defined ourselves (not that it was too difficult!) to draw each cube, setting its color by its location in the space:

```c
typedef GLfloat color[4];
color cubecolor;

cubecolor[0] = r; cubecolor[1] = g; cubecolor[2] = b;
cubecolor[3] = 1.0;
glColor4fv(cubecolor);
```

However, we don’t change the geometry of the cube before we draw it, so how does this cube move all over the space? In fact, we only use the cube we defined, which is a cube whose sides all have length two and which is centered on the origin. So how do we get the small cubes we actually draw? Our technique is to use transformations to define the size and location of each of the
cubes, with \textit{scaling} to define the size of the cube and \textit{translation} to define the position of the cube, as follows:

\begin{verbatim}
    glPushMatrix();
    glScalef(scale, scale, scale);
    glTranslatef(-SIZE+(float)i*2.0*scale*SIZE, SIZE, SIZE);
    cube((float)i/(float)NUMSTEPS, 1.0, 1.0);
    glPopMatrix();
\end{verbatim}

Note that we include a technique of pushing the current modeling transformation onto the current transformation stack, applying the translation and scaling transformations to the transformation in use, drawing the cube, and then popping the current transformation stack to restore the previous modeling transformation. This is discussed further in a later module where we discuss the general capabilities of transformations.

\textbf{Example: the HSV cone}

There are two functions of interest here. The first is the conversion from HSV colors to RGB colors; this is taken from [1] as indicated, and is based upon a geometric relationship between the cone and the cube, which is much clearer if you look at the cube along a diagonal between two opposite vertices. The second function does the actual drawing of the cone with colors generally defined in HSV and converted to RGB for display, and with color smoothing handling most of the problem of shading the cone. The full code for this program is included with this module.

\begin{verbatim}
void convertHSV2RGB(float h, float s, float v, float *r, float *g, float *b)
{
    // conversion from Foley et.al., fig. 13.34, p. 593
    float f, p, q, t;
    int   k;

    if (s == 0.0) { // achromatic case
        *r = *g = *b = v;
    }
    else { // chromatic case
        if (h == 360.0) h=0.0;
        h = h/60.0;
        k = (int)h;
        f = h - (float)k;
        p = v * (1.0 - s);
        q = v * (1.0 - (s * f));
        t = v * (1.0 - (s * (1.0 - f)));
        switch (k) {
            case 0: *r = v; *g = t; *b = p; break;
            case 1: *r = q; *g = v; *b = p; break;
            case 2: *r = p; *g = v; *b = t; break;
            case 3: *r = p; *g = q; *b = v; break;
            case 4: *r = t; *g = p; *b = v; break;
            case 5: *r = v; *g = p; *b = q; break;
        }
    }
}

void HSV(void)
{
    #define NSTEPS 36
    #define steps (float)NSTEPS
    #define TWOPI 6.28318
}
\end{verbatim}
int i;
float r, g, b;

glBegin(GL_TRIANGLE_FAN); // cone of the HSV space
    glColor3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 0.0, -2.0);
    for (i=0; i<=NSTEPS; i++) {
        convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
        glColor3f(r, g, b);
        glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                    2.0*sin(TWOPI*(float)i/steps),2.0);
    }
    glEnd();

glBegin(GL_TRIANGLE_FAN); // top plane of the HSV space
    glColor3f(1.0, 1.0, 1.0);
    glVertex3f(0.0, 0.0, 2.0);
    for (i=0; i<=NSTEPS; i++) {
        convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
        glColor3f(r, g, b);
        glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                    2.0*sin(TWOPI*(float)i/steps),2.0);
    }
    glEnd();

Example: the HLS double cone

The conversion itself takes two functions, while the function to display the double cone is so close
to that for the HSV model that we do not include it here. The source of the conversion functions is
again Foley et al.

void convertHLS2RGB(float h, float l, float s, float *r, float *g, float *b) {
    // conversion from Foley et.al., Figure 13.37, page 596
    float m1, m2;
    if (l <= 0.5) m2 = l*(1.0+s);
    else m2 = l + s - l*s;
    m1 = 2.0*l - m2;
    if (s == 0.0) { // achromatic cast
        *r = *g = *b = l;
    }
    else { // chromatic case
        *r = value(m1, m2, h+120.0);
        *g = value(m1, m2, h);
        *b = value(m1, m2, h-120.0);
    }
}

float value( float n1, float n2, float hue) {
    // helper function for the HLS->RGB conversion
    if (hue > 360.0) hue -= 360.0;
    if (hue < 0.0) hue += 360.0;
    if (hue < 60.0) return( n1 + (n2 - n1)*hue/60.0 );
    if (hue < 180.0) return( n2 );
    if (hue < 240.0) return( n1 + (n2 - n1)*(240.0 - hue)/60.0 );
    return( n1 );
}
Example: An object with partially transparent faces

The code in the file alphacolor.c has a few properties that are worth noting. This code draws three squares in space, each centered at the origin and lying within one of the coordinate planes. These three squares are colored according to the following declaration:

```c
GLfloat color0[]={1.0, 0.0, 0.0, 0.5}, // R
    color1[]={0.0, 1.0, 0.0, 0.5}, // G
    color2[]={0.0, 0.0, 1.0, 0.5}; // B
```

These colors are the full red, green, and blue colors with a 0.5 alpha value, so when each square is drawn it uses 50% of the background color and 50% of the square color. You will see that blending in Figure 3.8 for this example.

As we saw in the example above, the color of each part is specified just as the part is drawn:

```c
point3 plane0[4]= {(-1.0, 0.0, -1.0), // X-Z plane
                   (-1.0, 0.0, 1.0),
                   (1.0, 0.0, 1.0),
                   (1.0, 0.0, -1.0) };
```

This is not necessary if many of the parts had the same color; once a color is specified, it is used for anything that is drawn until the color is changed.

Science projects

- Create objects having a property that can be encoded with color, and display the object using color to demonstrate that property. For example, consider the problem of heat transport in a bar, with some points of the bar being held at different temperatures by heat sources or sinks, and draw the bar with colors illustrating the temperatures at different parts of the bar. (Note that there are conventions about colors representing heat — red represents heat, and a “cooler” color such as blue or green (or cyan, which is (0,1,1) ) representing cold.)

- There are some fields in which colors have standardized meanings. In chemistry, there are standard colors for some atoms in molecular displays, for example. Identify some such fields and examine how the colors are used to show structures or other properties in this case.

- If you do not have lighting and shading, it can be difficult to see shapes in graphic displays. It is possible to display shapes by using color to give hints to the eye. For example, topographic maps use colored lines at constant elevations to tell us where a landscape is largely flat (the lines are far apart) or very steep (the lines are very close together). If we are drawing a surface and have defined the surface in terms of a fine grid in the domain, it can be very helpful if we use color to describe height by using different colors for different heights. This allows our eyes to identify shapes much as they did for topographic maps.