



CHAPTER 1

Overview of graphics

- 1.1 Perl and graphics 4
- 1.2 The bits and pieces in graphics programming 5
- 1.3 Color spaces and palettes 7
- 1.4 Summary 14

With the introduction of the Web, the sheer volume of graphics being manipulated has increased enormously. Even without the Web the amount of image processing would have grown, simply due to the fact that computers are faster than they used to be and have more memory, which makes dealing with graphics more feasible. The advent of digital cameras and affordable frame capture hardware have also contributed to the increase in graphics manipulation. There are more and more photos and stills from videos to manipulate, scale, and index every single day.

Because computers have become more powerful, there are also more applications that create graphical output. This in turn has increased the expectancy that *your* application creates graphical output—either interactively, possibly through a graphical user interface, or noninteractively, as files or database records. Charts and graphs are created to visualize everything ranging from business throughput, stock prices, temperature fluctuations, population densities, and web site visits, to the average life expectancy of female fruit flies during the wet season on Bali, all in their subspecies incarnations.

Apart from increasing the amount of graphics needed as buttons and embellishments for web sites, the Web has had a major influence in areas of images created on the fly and mass manipulation of large sets of graphics files. Just think of all the images out there that are resized, optimized, dithered, thumbnailed and cropped for

display as part of a web site—for example as a list of available stock items. Then there are the images that are created on the fly as the result of user input; for example, charts or parts of maps or street directories. No one of course knows the exact numbers, but they are vast.

Together with the outpouring of images processed, there has been an increase in the number of software packages and modules that make their creation and manipulation possible, as well as a boom in the development rate of packages that already existed before the web came along. This proliferation of tools can make it difficult to pick the right one for the job at hand, especially since some of the tools overlap in application areas. Sometimes there is no tool available that will satisfy all the requirements of the task, and you have to write your own. Even when you do write your own tools, you need to choose, or rather, can choose, between several libraries and modules available. Which library you decide to use depends, of course, on your needs and on your familiarity with the products.

Not all graphics tasks lend themselves to automation. There are, and always will be, many that can only be achieved by sitting down at the screen of a computer, firing up your favorite drawing or graphics manipulation program, and interactively using your eyes and mouse skills to achieve the desired effect. Writing code for graphics manipulation can be quite time-consuming, and you need to take that into account when making a decision about whether you want to automate a certain task or process.

Programs that manipulate or create graphics are in many ways similar to programs for other tasks. There are, however, aspects which apply to, or which are particularly relevant to, graphics programming specifically. In this chapter we will have a look at the various elements of graphics programming that are important to programmers.

1.1 PERL AND GRAPHICS

One caveat that should be understood about Perl is that it is not a language particularly suited to large amounts of number crunching, and image manipulation requires large amounts of number crunching. So why does this book exist?

Perl is a glue language, which is one of the reasons why it is often described as the duct tape of the Internet.¹ Perl is good at snipping and gluing pieces of data, particularly text, and sticking tools and libraries together. What's more, it allows you to do all this with ease and a minimum of development time. And this is why Perl can be very useful for graphics programming.

Because of Perl's gluing ability, there are several interfaces to graphics manipulation libraries and programs available in the form of modules. Other packages have been written that make use of these modules to create graphics at a higher level, for example to create charts. Together, the number of these tools has grown sufficiently to allow the tackling of many graphics programming tasks in Perl.

¹ This is generally attributed to Hassan Schroeder, Sun's first webmaster.

Apart from the reasons just mentioned, Perl is also an attractive language with which to program, because of its flexible grammar, the large number of built-in tools and the variety of syntactical constructions. Perl can be used as a tool to quickly hack together a program that parses sets of log files, and gives some nice summaries and graphs with which to create a report. It can also be used to write large software projects, maintained by several programmers and developed over the course of a year or more. Perl is duct tape, but it is very flexible duct tape.

As a side note to the number crunching: Moore's law states that every two years the average speed of electronic computers doubles. Even if you generalize that to all computing, as Ray Kurzweil does in *The Age of the Spiritual Machine* [5], this law holds remarkably well from all the way back in the nineteenth century to today. This means that while Perl might be too slow for many tasks in computer graphics manipulation on today's computers, this will probably not hold true on computers in the not too distant future. The demands on programs that manipulate computer graphics will probably flatten out at some point, once the resolution of what is created is higher than the resolution of the human eye. At the same time, the increase in computing power will continue. Even if you don't believe this, the other reasons already stated are more than enough to see Perl as a valid programming language in the computer graphics world, if not generally, then at least for some tasks.

Many modules for Perl that concern themselves with the manipulation of graphics are written in C or C++, and provide an interface to this functionality in the Perl language. Most of these modules were, in fact, born as C libraries or programs, and their original authors probably didn't have Perl in mind at all when they wrote them. All the number crunching, array manipulation and bit-shifting happens in compiled low level code, and therefore is much faster than could be achieved in pure Perl. This does not mean that you cannot, under any circumstance, use Perl to directly manipulate pixels in an image. Chapter 12 gives a few examples of how to do this in pure Perl, but be forewarned that it will not be blindingly fast. The same chapter also explains how to write parts of your program in C, by including the C code directly in your program. This allows you to escape Perl's slowness when you need to, without losing any of its advantages.

Summarizing: while Perl isn't particularly suitable for low-level computer graphics manipulation, there are many modules available that make the most common graphics manipulation and creation task available to a Perl programmer. In cases where the need is to jump down to a low level and do some computationally intensive programming, Perl provides access to lower-level languages without too much fuss. Of course, if the majority of a program consists of these tasks, a language other than Perl should probably be considered.

1.2 THE BITS AND PIECES IN GRAPHICS PROGRAMMING

Generally speaking, programming for computer graphics consists of working with a limited set of concepts: the drawing primitives.

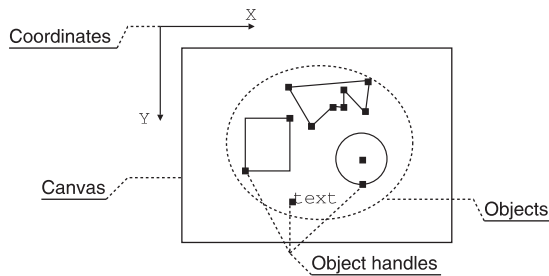


Figure 1.1
Some drawing primitives that can be created and manipulated with computer graphics programs and packages.

First of all there is the *canvas*, which is the medium that is being drawn on, or read from. In PostScript, for example, this is a page or part of a page, and for image manipulation packages such as *GD* this is a two-dimensional array of pixels. It is important to note that a canvas can be part of another canvas, and that certain images can be built up of multiple canvases, such as the layers and channels in a GIMP image.

A second element common to all graphics operations is a *frame of reference* expressed in coordinates. This indicates where on the canvas an object is located or an operation takes place. Most commonly these coordinates are Cartesian, with a horizontal and a vertical component, but sometimes it is easier to use polar coordinates (see, for example, section 10.1.1 “Coordinate transformation,” on page 180).

Thirdly, there are the *objects* which are being drawn or manipulated, e.g., a rectangle, a circle, a polygon, a photo, a group of the previous, or a pixel array. Most of the time these objects will have one or more handles, which express their center or top left corner, and some dimensions.

The fourth element consists of the *tools* used for drawing. These can be a brush, a stamp, an eraser, a paint bucket or even a filter. These are normally found on icon bars in interactive drawing programs, but they also exist in noninteractive programming packages.

While the primitives of most graphics programming fall into one of the groups mentioned above, there is certainly not always a clear distinction. Something that is the canvas for one operation might be the brush or object to be drawn for another. And in fact, a lot of drawing software allows for use of part of one canvas as a brush for another. The whole graphic for one operation could be just a layer or a drawing object for another; think, for example of the object libraries of many drawing packages that predefine pictures of all kinds of common objects for use in a drawing.

Sometimes people consider the manipulation of single pixels in an image to be a separate class of actions. However, a pixel can be seen as a primitive object, and any action on a pixel is no different from any action on, for example, a circle. A pixel is simpler and has fewer parameters, but it is still an object.

Many graphics packages or graphics operations function at a higher level than described earlier, and you often won’t need to deal with the lower level details directly. When you want to resize a set of images, you’re not really concerned with what exactly

happens in terms of canvas, coordinates, and objects. When you use one of the modules that creates a chart you are hardly interested in which low-level graphics operations it must execute to produce the picture.

In addition to the creation and handling of these drawing primitives, there are sets of operations that can be applied to the objects in a computer graphic. Some of these operations are stretching, rotating, skewing, or resizing of objects. Some others are filters that work directly on the contents of an object, the way that various convolution filters described in section 12.3 “Convolution,” on page 215, work on the pixels in an image. There are also operations that let you combine graphical objects in various manners.

SEE ALSO We will see more about drawing primitives with the various modules that are available for Perl in chapter 4. More on the manipulation of images as a whole, and using images as objects to incorporate in other images, can be found in chapter 8. Chapter 10 contains more discussion on coordinate systems and frames of reference, and the manipulation of individual pixels is discussed in chapter 12.

1.3 COLOR SPACES AND PALETTES

One of the most important concepts in computer graphics is the storing and manipulation of color. The human eye is capable of distinguishing large numbers of colors, which ideally can all be represented in computer graphics terms. However, the more information needs to be represented, the more memory and CPU power is needed to work with that information. Apart from these considerations, the hardware used to present the colors also plays a major role in the conceptualization of a color model. This section provides a short overview of the color models most frequently used, and their relationship to each other.

All colors we see can be expressed as a composition of at least three other colors. This is because we use three different types of sensors on our retina to perceive color, each most sensitive to a different part of the visual spectrum. What exactly these three colors are isn't that important, since it turns out that it doesn't matter which three colors we pick to decompose colors into, as long as they are far enough away from each other. These colors are called primary colors, and every possible combination of three specific primary colors, taken together, makes up a color space. The coordinates into that space are the relative amounts of each primary color. As children we were taught that red, yellow and blue paint can be mixed in various combinations to make up virtually any other color of the rainbow. It turns out that it is possible to come up with several other useful color spaces. Let's look at some of them.

1.3.1 RGB

In computational graphics manipulation the most commonly used primary colors are red, green and blue, forming a color space referred to as RGB. The main reason that this color space is used is due to the fact that computer monitors perform their function with phosphorizing agents that emit those three colors. RGB is called an additive

system, because each color can be expressed as a different sum of the three components, and adding more of one component adds intensity to the resulting color.

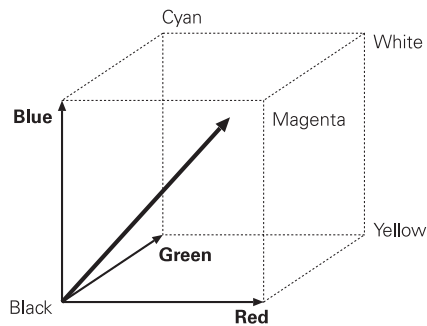


Figure 1.2 The RGB color cube illustrates the color space made up of the three colors red, green and blue. Each point inside the cube is a different valid color in RGB space.

blue-green by (0.5,0.7,0.7). The line that runs between the white and black corner represents all gray colors, and the coordinates for these points is (gr,gr,gr) where gr has a value between 0 (black) and 1 (white).

In color space coordinates, normally the three primary colors each can have a value between 0 and 1, which, for efficiency reasons, in most software applications is expressed as an integer value between 0 and 255. This means that visually, these colors can be mapped in a three-dimensional space; more precisely, to a cube. For the RGB color space this cube can be seen in figure 1.2

Each color can be represented by a vector with coordinates (r,g,b) in that cube. Looking at this cube we see that yellow is represented by the coordinates (1,1,0), black by (0,0,0), white by (1,1,1) and a particularly irritating shade of

1.3.2 CMY and CMYK

Another color space made up of three primary colors is CMY, for cyan, magenta and yellow. CMY is often used for dyes and filters. These three primaries are the opposites, or more correctly, the complementaries of the RGB colors and form a so-called subtractive system, in which an increase in any color component causes a decrease in the intensity of the resulting color. In other words, the color seen when looking at a printed page is whatever is left over after the ink has absorbed part of the spectrum. If you have another look at the RGB cube in figure 1.2, you'll note that exactly opposite of the red, green and blue corners, are cyan, magenta and yellow. The relationship between RGB and CMY can be expressed as:

$$\begin{aligned}C &= 1 - R \\M &= 1 - G \\Y &= 1 - B\end{aligned}$$

Black in CMY color space is (1,1,1), white is (0,0,0), green is (1,0,1), and our irritating blue-green is (0.5,0.3,0.3). The gray colors are in exactly the same position, but their coordinates are reversed, i.e., white is given by (0,0,0) and black by (1,1,1).

In practice it is almost impossible to mix inks correctly and consistently, which is why printers normally work in a color space that has an added component—black. This color space is called CMYK. The largest possible value for the black K component is determined by taking the lowest of the CMY components, and subtracting that value from all three components. In effect, the maximum amount of black (gray) is

subtracted from the color, and given its own coordinate. The relationship between RGB and CMYK is given by the following equations:

$$K \leq \min(1 - R, 1 - G, 1 - B)$$

$$C = (1 - R - K)$$

$$M = (1 - G - K)$$

$$Y = (1 - B - K)$$

$$R = 1 - (C + K)$$

$$G = 1 - (M + K)$$

$$B = 1 - (Y + K)$$

This, again, is an idealized representation of the way CMYK is used in the real world. Printers do not necessarily always subtract the largest amount of black from the individual colors, but they decide to use a value for K which better suits the inks they use. And even when the black component of a color is treated separately, the other inks are seldom clean or pure enough to be mixed in this ideal way. Instead, picking the correct mixing ratios is an art that printers practice for many years, and their experience will provide a much more solid foundation than the simplistic formula above.

1.3.3 HSV and HLS

Instead of using primary colors in certain combinations to identify a color, other attributes can be used. Some color spaces identify a color by its hue, which is basically the position on the rainbow, its saturation or pureness of the color, and its brightness (or value, or lightness). The two most common ones are HSV (for Hue, Saturation and Value) and HLS (Hue, Lightness and Saturation). Many people feel that a color is most naturally identified with these color spaces, because they closely reflect the way we perceive colors. When we see a color, typically we first observe its hue, i.e., whether it is green or purple or red. The next thing we notice is its saturation—whether it's a pastel tint or a pure color. And finally we take note of the brightness of the color.

Conversion from HSV or HLS to RGB is, unfortunately, not a linear process. Appendix B contains the algorithms for conversion to and from these color spaces. Before looking at those, it is probably important to understand how the HSV coordinates can be seen in terms of the RGB color cube.

Hue is normally expressed as a value from 0 to 360, which indicates a position on the color circle. The colors on the circle are arranged in the same order as they are on the rainbow, but in a circle, with pure red at an angle of 0 (and, of course, 360), green at 120 and blue at 240 degrees. All the colors with the same hue form a plane in the RGB color cube, as can be seen in figure 1.3.

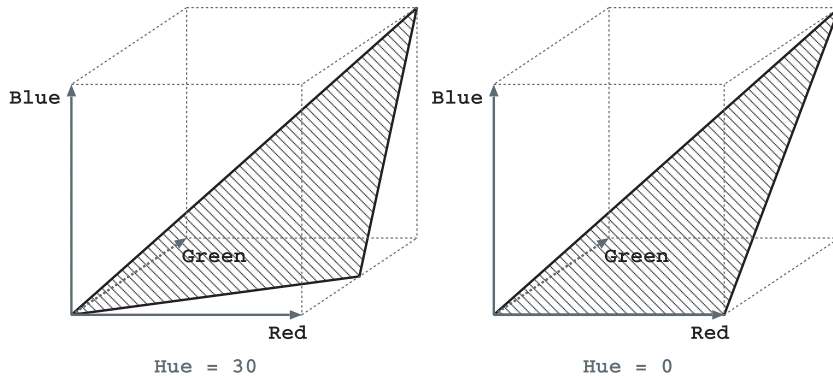


Figure 1.3 A set of colors with a constant hue forms a plane in the RGB color space. These planes form triangles inside the RGB color cube, where one of the sides is the diagonal from $(0,0,0)$ to $(1,1,1)$.

The saturation of a color expresses how pure the hue is, relative to white. A color with a saturation of 1 is pure, and a color with a saturation of 0 is gray. Pastel colors are colors with a low saturation, and the company colors picked by food chains and car dealerships normally have a high saturation. All the colors with the same saturation (but varying hue and value) fall on a cone in the RGB color cube, with the line between the white and black corners as its axis. See figure 1.4 for an illustration of this cone.

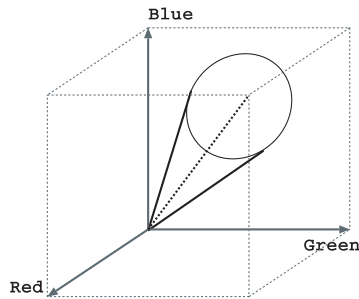


Figure 1.4 The cone of constant saturation in the RGB color cube. Note that the cube has been rotated 90 degrees around the blue axis, compared to figures 1.2 and 1.3.

The value for HSV and the lightness for HLS are both meant to express the amount of luminosity of the color. This is, however, not the same thing as the brightness of a color, which is often defined as the sum of the brightness of the individual colors. The lightness and value are slightly different quantities, and less directly mappable in the RGB space. The planes of constant lightness and value are difficult to draw, which is why I gave up trying and plotted them with gnuplot, using the subroutines in appendix B. An example of constant lightness can be seen in figure 1.5, and one of constant value in figure 1.6.

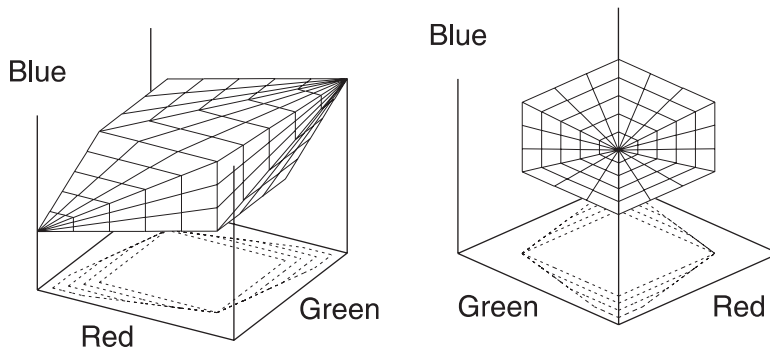


Figure 1.5 Constant HSL lightness (0.6) in the RGB color cube, seen from two angles. The three-dimensional figure becomes narrower when the lightness value decreases, and wider when it increases. The tips of the figure remain stationary.

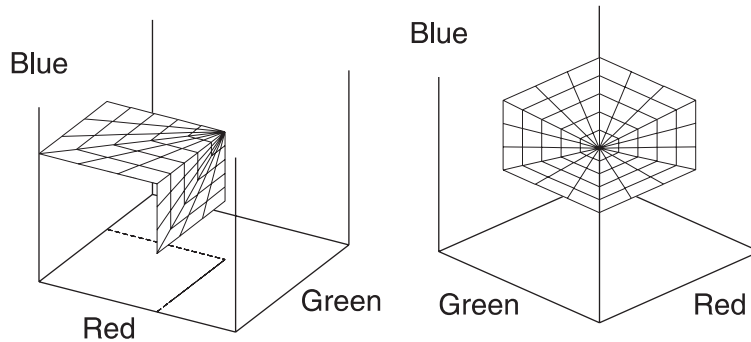


Figure 1.6 Constant HSV value (0.6) in the RGB color cube, seen from two angles. The figure grows smaller and closer to the origin when the value decreases, and larger and closer to the point (1, 1, 1) when the value increases.

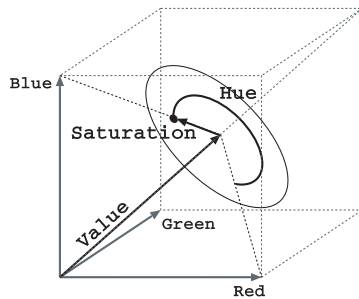


Figure 1.7 The HSV coordinates in the RGB color cube. Represented is a point that has a Hue of 240 degrees (and therefore points in the direction of the blue corner), a saturation of approximately 20 percent, and a value of 50 percent.

And finally, and then I'll stop talking about HSV, the coordinates of the HSV color system are depicted in figure 1.7.² The black dot in that figure represents a point in the HSV color space with a hue of approximately 240 degrees, a saturation of approximately 20 percent, and a value or lightness of 50 percent. This corresponds to a dark, dull blue or almost gray-blue, with RGB coordinates in the proximity of (0.4,0.4,0.5).

1.3.4 YUV, YIQ and YCbCr

One group of color spaces that is often used in video specifications expresses color as one component of luminance, and two of chrominance. This means that one of the three components expresses how bright the color is, and the other two determine its color. The most common of these are YUV, YIQ and YCbCr. YUV and YIQ are the color spaces used in the PAL (in use in Europe, among other places), and NTSC (used in the USA) video standards. Both of these are highly unportable, and it is unreliable to convert YUV or YIQ from or to RGB coordinates. The scale factors used in these color systems are just not appropriate outside their application domain. YCbCr is more appropriate. In fact, often when someone gives a formula to transform between RGB and YUV, what they have really given is the transformation from and to YCbCr.

The component that everything revolves around in these color spaces is the Luminance, Y. It can be calculated from the RGB coordinates with

$$Y = 0.299R + 0.587G + 0.114B$$

The meaning of the two chrominance components—with different scaling factors for the different color spaces—can be understood by loosely defining them as:

$$\begin{aligned}U, I, Cb &= S1 (B - Y) \\V, Q, Cr &= S2 (R - Y)\end{aligned}$$

wherein S1 and S2 are scaling factors that are defined by the respective standards.³ These color spaces will not serve the purpose of this book, so I won't discuss them any further. Suffice it to say that they normally would be encountered only when working with video.

1.3.5 Grayscale

Finally, grayscale is an often used color space which really is only one dimensional, and therefore isn't much of a color space. It is capable only of expressing the relative brightness of pixels. Conversion from a real color space to grayscale is a one-way process, because the color information is lost along the way. There are several ways to

² It should be understood that this just is a schematic indication of the coordinates. The length of the various arrows is not directly related to the HSV coordinates. However, the drawing is useful for understanding the relationship between RGB and HSV.

³ The exact magnitude of these scaling factors is not important for this discussion. See for example the reference entries [6],[7].

achieve this, some of which produce more natural results than others, depending on the nature of the original image. The most frequently used conversions are:

$$\text{Luminance} = 0.299R + 0.587G + 0.114B$$

$$\text{Brightness} = (R + G + B)/3$$

$$\text{Lightness} = [\max(R, G, B) + \min(R, G, B)]/2$$

$$\text{Value} = \max(R, G, B)$$

Luminance is, of course, the Y component of the YUV or related color spaces (also see the color FAQ [7]). This generally renders the most natural result. Brightness is simply the average of the three color components, which tends to overemphasize any blue tints in an image. The eye is much less sensitive to blue than it is to red, and much less sensitive to red than it is to green. The scaling factors in the equation for luminance attempt to reflect this difference in sensitivity. The last two are the L component of the HSL color space and the V component of the HSV color space. Both are often easily implemented in software, because they require only a desaturation (setting the S component to 0) in the appropriate color space. However, the results of these conversions does not always meet expectations.

1.3.6 Color distance

One difference between the various color models is the way the distances in color space are calculated. The distance in a color space is defined in the same way that a distance in geometrical space is defined:

$$\|c_1 - c_2\| = \sqrt{(c_{1,x} - c_{2,x})^2 + (c_{1,y} - c_{2,y})^2 + (c_{1,z} - c_{2,z})^2}$$

for a color space with three coordinates x , y and z . A different color space will yield a different distance between the same colors. This distance is used to calculate how similar two colors are, and picking a different color space can yield very different results. Some color spaces (such as YUV) provide distances that are more closely related to the way our perception of color works than the more standard RGB color space.

The color distance is most important when reducing the numbers of colors in an image. At some point this color reduction will require changing a pixel's color to one that is as close as possible to the original *and* part of the set of colors that are available after the reduction. Doing these operations in different color spaces can result in visibly different images.

1.3.7 Reducing the number of colors in an image

The main reason for wanting to reduce the number of colors in an image is to save it in an indexed image format. Indexed image formats store all available colors in a palette or color map. Each pixel in the image needs only to contain an index into that

palette, instead of the three coordinates into the color space. This can result in remarkable savings in disk space, but it comes at the cost of losing some color resolution.

Currently, one of the best known color palettes is the web-safe palette, or the Netscape color palette (also see 6.2.1 “Web safe color palettes,” on page 92). *Image::Magick* provides a built-in image type to convert images to that particular palette:

```
$map = Image::Magick->new();
$map->Read('netscape:');
$image = Image::Magick->new();
$image->Read('some_image.jpg');
$image->Map(image => $map, dither => 1);
$image->Write('some_image.gif');
```

You create an image, `$map`, that only contains the built-in netscape image type, which can then be used as a palette mapping image for *Image::Magick*'s `Map()` method. This method will remap the colors of the current image into the ones of the image argument.

1.4 SUMMARY

In this chapter we've discussed the basics of graphics programming in a general sense, with an emphasis on colors and color spaces. Chapter 4 has more practical information on how to work with graphics primitives. We haven't covered everything there is to know and have breezed over some of the details of the discussed matter; however, the introduction presented in this chapter should be sufficient to understand the rest of this book.

SEE ALSO If you are interested in a more elaborate discussion of colors in computer graphics, I suggest you look at the color space FAQ and color FAQ listed in the references as [6] and [7], and pick up one of the books recommended there, if you wish.

Appendix B contains Perl code to convert from RGB to HSV or HLS and back.