Color

This section describes the toolbox functions that help you work with color image data. Note that "color" includes shades of gray; therefore much of the discussion in this chapter applies to grayscale images as well as color images. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	**Definitions**
Approximation	The method by which the software chooses replacement colors in the event that direct matches cannot be found. The methods of approximation discussed in this chapter are colormap mapping, uniform quantization, and minimum variance quantization.
Indexed image	An image whose pixel values are direct indices into an RGB colormap. In MATLAB, an indexed image is represented by an array of class `uint8`, `uint16`, or `double`. The colormap is always an m-by-3 array of class `double`. We often use the variable name `X` to represent an indexed image in memory, and `map` to represent the colormap.
Intensity image	An image consisting of intensity (grayscale) values. In MATLAB, intensity images are represented by an array of class `uint8`, `uint16`, or `double`. While intensity images are not stored with colormaps, MATLAB uses a system colormap to display them. We often use the variable name `I` to represent an intensity image in memory. This term is synonymous with the term *grayscale*.
RGB image	An image in which each pixel is specified by three values — one each for the red, blue, and green components of the pixel's color. In MATLAB, an RGB image is represented by an m-by-n-by-3 array of class `uint8`, `uint16`, or `double`. We often use the variable name `RGB` to represent an RGB image in memory.
Screen bit depth	The number of bits per screen pixel.
Screen color resolution	The number of distinct colors that can be produced by the screen.

Working with Different Screen Bit Depths

Most computer displays use 8, 16, or 24 bits per screen pixel. The number of bits per screen pixel determines the display's *screen bit depth*. The screen bit depth determines the *screen color resolution*, which is how many distinct colors the display can produce.

Regardless of the number of colors your system can display, MATLAB can store and process images with very high bit depths: 2^{24} colors for `uint8` RGB images, 2^{48} colors for `uint16` RGB images, and 2^{159} for `double` RGB images. These images display best on systems with 24-bit color, but usually look fine on 16-bit systems as well. (For additional information about how MATLAB handles color, see the MATLAB graphics documentation.)

This section:

- Describes how to determine your system's screen bit depth
- Provides guidelines for choosing a screen bit depth

Determining Your Systems Screen Bit Depth

To determine your system's screen bit depth, enter this command at the MATLAB prompt.

```
get(0,'ScreenDepth')
ans =

     16
```

The integer MATLAB returns represents the number of bits per screen pixel:

Value	Screen Bit Depth
8	8-bit displays supports 256 colors. An 8-bit display can produce any of the colors available on a 24-bit display, but only 256 distinct colors can appear at one time. (There are 256 shades of gray available, but if all 256 shades of gray are used, they take up all of the available color slots.)
16	16-bit displays usually use 5 bits for each color component, resulting in 32 (i.e., 2^5) levels each of red, green, and blue. This supports 32,768 (i.e., 2^{15}) distinct colors (of which 32 are shades of gray). Some systems use the extra bit to increase the number of levels of green that can be displayed. In this case, the number of different colors supported by a 16-bit display is actually 64,536 (i.e. 2^{16}).
24	24-bit displays use 8 bits for each of the three color components, resulting in 256 (i.e., 2^8) levels each of red, green, and blue. This supports 16,777,216 (i.e., 2^{24}) different colors. (Of these colors, 256 are shades of gray. Shades of gray occur where R=G=B.) The 16 million possible colors supported by 24-bit display can render a life-like image.
32	32-bit displays use 24 bits to store color information and use the remaining 8 bits to store transparency data (alpha channel). For information about how MATLAB supports the alpha channel, see Transparency.

Choosing a Screen Bit Depth

Depending on your system, you may be able to choose the screen bit depth you want to use. (There may be trade-offs between screen bit depth and screen color resolution.) In general, 24-bit display mode produces the best results. If you need to use a lower screen bit depth, 16-bit is generally preferable to 8-bit. However, keep in mind that a 16-bit display has certain limitations, such as:

- An image may have finer gradations of color than a 16-bit display can represent. If a color is unavailable, MATLAB uses the closest approximation.

- There are only 32 shades of gray available. If you are working primarily with grayscale images, you may get better display results using 8-bit display mode, which provides up to 256 shades of gray.

For information about reducing the number of colors used by an image, see "Reducing the Number of Colors in an Image" on page 13-6.

Reducing the Number of Colors in an Image

This section describes how to reduce the number of colors in an indexed or RGB image. A discussion is also included about dithering, which is used by the toolbox's color-reduction functions (see below.) Dithering is used to increase the apparent number of colors in an image.

The table below summarizes the Image Processing Toolbox functions for color reduction.

Function	**Purpose**
`imapprox`	Reduces the number of colors used by an indexed image, enabling you specify the number of colors in the new colormap.
`rgb2ind`	Converts an RGB image to an indexed image, enabling you to specify the number of colors to store in the new colormap.

On systems with 24-bit color displays, RGB (truecolor) images can display up to 16,777,216 (i.e., 2^{24}) colors. On systems with lower screen bit depths, RGB images still displays reasonably well, because MATLAB automatically uses color approximation and dithering if needed.

Indexed images, however, may cause problems if they have a large number of colors. In general, you should limit indexed images to 256 colors for the following reasons:

- On systems with 8-bit display, indexed images with more than 256 colors will need to be dithered or mapped and, therefore, may not display well.
- On some platforms, colormaps cannot exceed 256 entries.
- If an indexed image has more than 256 colors, MATLAB cannot store the image data in a `uint8` array, but generally uses an array of class `double` instead, making the storage size of the image much larger (each pixel uses 64 bits).
- Most image file formats limit indexed images to 256 colors. If you write an indexed image with more than 256 colors (using `imwrite`) to a format that does not support more than 256 colors, you will receive an error.

Using rgb2ind

`rgb2ind` converts an RGB image to an indexed image, reducing the number of colors in the process. This function provides the following methods for approximating the colors in the original image:

- Quantization
  - Uniform quantization
  - Minimum variance quantization
- Colormap mapping

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See "Dithering" on page 13-13 for a description of dithering and how to enable or disable it.

Quantization

Reducing the number of colors in an image involves *quantization*. The function `rgb2ind` uses quantization as part of its color reduction algorithm. `rgb2ind` supports two quantization methods: *uniform quantization* and *minimum variance quantization*.

An important term in discussions of image quantization is *RGB color cube*, which is used frequently throughout this section. The RGB color cube is a three-dimensional array of all of the colors that are defined for a particular data type. Since RGB images in MATLAB can be of type `uint8`, `uint16`, or `double`, three possible color cube definitions exist. For example, if an RGB image is of class `uint8`, 256 values are defined for each color plane (red, blue, and green), and, in total, there will be 2^{24} (or 16,777,216) colors defined by the color cube. This color cube is the same for all `uint8` RGB images, regardless of which colors they actually use.

The `uint8`, `uint16`, and `double` color cubes all have the same range of colors. In other words, the brightest red in an `uint8` RGB image displays the same as the brightest red in a `double` RGB image. The difference is that the `double` RGB color cube has many more shades of red (and many more shades of all colors). Figure 13-1, below, shows an RGB color cube for a `uint8` image.

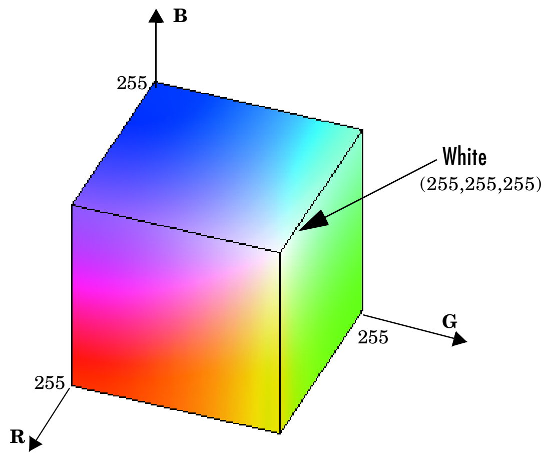

Figure 13-1: RGB Color Cube for uint8 Images

Quantization involves dividing the RGB color cube into a number of smaller boxes, and then mapping all colors that fall within each box to the color value at the *center* of that box.

Uniform quantization and minimum variance quantization differ in the approach used to divide up the RGB color cube. With uniform quantization, the color cube is cut up into equal-sized boxes (smaller cubes). With minimum variance quantization, the color cube is cut up into boxes (not necessarily cubes) of different sizes; the sizes of the boxes depend on how the colors are distributed in the image.

Uniform Quantization. To perform uniform quantization, call `rgb2ind` and specify a *tolerance*. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is [0,1]. For example, if you specify a tolerance of `0.1`, the edges of the boxes are one-tenth the length of the RGB color cube and the maximum total number of boxes is

```
n = (floor(1/tol)+1)^3
```

The commands below perform uniform quantization with a tolerance of 0.1.

```
RGB = imread('flowers.tif');
[x,map] = rgb2ind(RGB, 0.1);
```

Figure 13-2 illustrates uniform quantization of a `uint8` image. For clarity, the figure shows a two-dimensional slice (or color plane) from the color cube where Red=0, and Green and Blue range from 0 to 255. The actual pixel values are denoted by the centers of the x's.

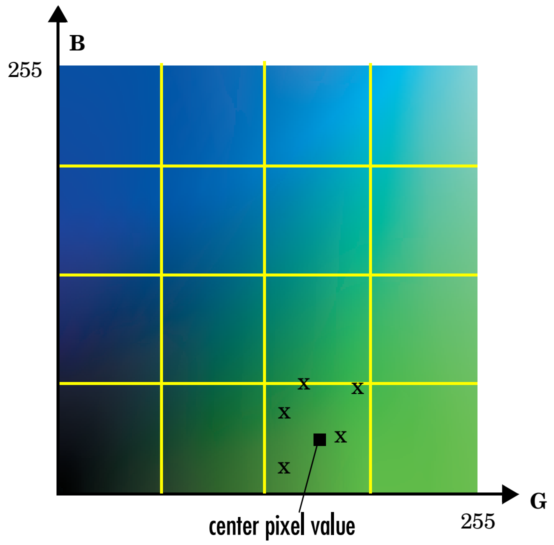

Figure 13-2: Uniform Quantization on a Slice of the RGB Color Cube

After the color cube has been divided, all empty boxes are thrown out. Therefore, only one of the boxes in Figure 13-2 is used to produce a color for the colormap. As shown earlier, the maximum length of a colormap created by uniform quantization can be predicted, but the colormap can be smaller than the prediction because `rgb2ind` removes any colors that do not appear in the input image.

Minimum Variance Quantization. To perform minimum variance quantization, call `rgb2ind` and specify the maximum number of colors in the output image's colormap. The number you specify determines the number of boxes into which the RGB color cube is divided. These commands use minimum variance quantization to create an indexed image with 185 colors.

```
RGB = imread('flowers.tif');
[X,map] = rgb2ind(RGB,185);
```

Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixel values may be grouped together because none of their values is greater than 5 from the center pixel of the group.

In minimum variance quantization, the boxes that divide the color cube vary in size, and do not necessarily fill the color cube. If some areas of the color cube do not have pixels, there are no boxes in these areas.

While you set the number of boxes, `n`, to be used by `rgb2ind`, the placement is determined by the algorithm as it analyzes the color data in your image. Once the image is divided into `n` optimally located boxes, the pixels within each box are mapped to the pixel value at the center of the box, as in uniform quantization.

The resulting colormap usually has the number of entries you specify. This is because the color cube is divided so that each region contains at least one color that appears in the input image. If the input image uses fewer colors than the number you specify, the output colormap will have fewer than `n` colors, and the output image will contain all of the colors of the input image.

Figure 13-3 shows the same two-dimensional slice of the color cube as was used in Figure 13-2 (for demonstrating uniform quantization). Eleven boxes have been created using minimum variance quantization.

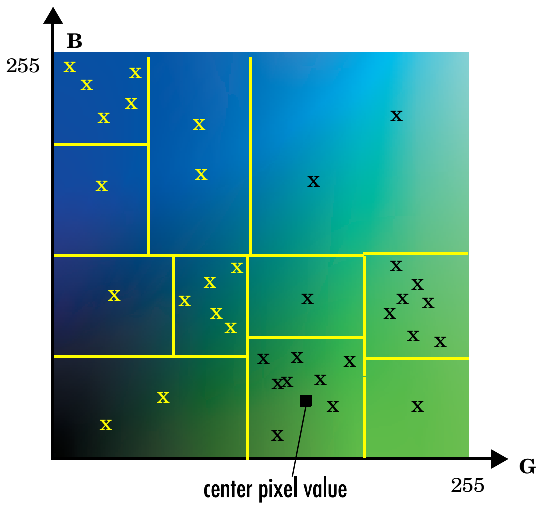

Figure 13-3: Minimum Variance Quantization on a Slice of the RGB Color Cube

For a given number of colors, minimum variance quantization produces better results than uniform quantization, because it takes into account the actual data. Minimum variance quantization allocates more of the colormap entries to colors that appear frequently in the input image. It allocates fewer entries to colors that appear infrequently. As a result, the accuracy of the colors is higher than with uniform quantization. For example, if the input image has many shades of green and few shades of red, there will be more greens than reds in the output colormap. Note that the computation for minimum variance quantization takes longer than that for uniform quantization.

Colormap Mapping

If you specify an actual colormap to use, `rgb2ind` uses *colormap mapping* (instead of quantization) to find the colors in the specified colormap that best match the colors in the RGB image. This method is useful if you need to create images that use a fixed colormap. For example, if you want to display multiple indexed images on an 8-bit display, you can avoid color problems by mapping them all to the same colormap. Colormap mapping produces a good approximation if the specified colormap has similar colors to those in the RGB image. If the colormap does not have similar colors to those in the RGB image, this method produces poor results.

This example illustrates mapping two images to the same colormap. The colormap used for the two images is created on the fly using the MATLAB function `colorcube`, which creates an RGB colormap containing the number of colors that you specify. (`colorcube` always creates the same colormap for a given number of colors.) Because the colormap includes colors all throughout the RGB color cube, the output images can reasonably approximate the input images.

```
RGB1 = imread('autumn.tif');
RGB2 = imread('flowers.tif');
X1 = rgb2ind(RGB1,colorcube(128));
X2 = rgb2ind(RGB2,colorcube(128));
```

Note The function `subimage` is also helpful for displaying multiple indexed images. For more information see “Displaying Multiple Images in the Same Figure” on page 3-20 or the reference page for `subimage`.

Reducing Colors in an Indexed Image

Use `imapprox` when you need to reduce the number of colors in an indexed image. `imapprox` is based on `rgb2ind` and uses the same approximation methods. Essentially, `imapprox` first calls `ind2rgb` to convert the image to RGB format, and then calls `rgb2ind` to return a new indexed image with fewer colors.

For example, these commands create a version of the `trees` image with 64 colors, rather than the original 128.

```
load trees
[Y,newmap] = imapprox(X,map,64);
imshow(Y, newmap);
```

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See “Dithering” on page 13-13 for a description of dithering and how to enable or disable it.

Dithering

When you use rgb2ind or imapprox to reduce the number of colors in an image, the resulting image may look inferior to the original, because some of the colors are lost. rgb2ind and imapprox both perform *dithering* to increase the apparent number of colors in the output image. Dithering changes the colors of pixels in a neighborhood so that the average color in each neighborhood approximates the original RGB color.

For an example of how dithering works, consider an image that contains a number of dark pink pixels for which there is no exact match in the colormap. To create the appearance of this shade of pink, the Image Processing Toolbox selects a combination of colors from the colormap, that, taken together as a six-pixel group, approximate the desired shade of pink. From a distance, the pixels appear to be correct shade, but if you look up close at the image, you can see a blend of other shades, perhaps red and pale pink pixels. The commands below load a 24-bit image, and then use rgb2ind to create two indexed images with just eight colors each.

```
rgb=imread('lily.tif');
imshow(rgb);
[X_no_dither,map]=rgb2ind(rgb,8,'nodither');
[X_dither,map]=rgb2ind(rgb,8,'dither');
figure, imshow(X_no_dither,map);
figure, imshow(X_dither,map);
```

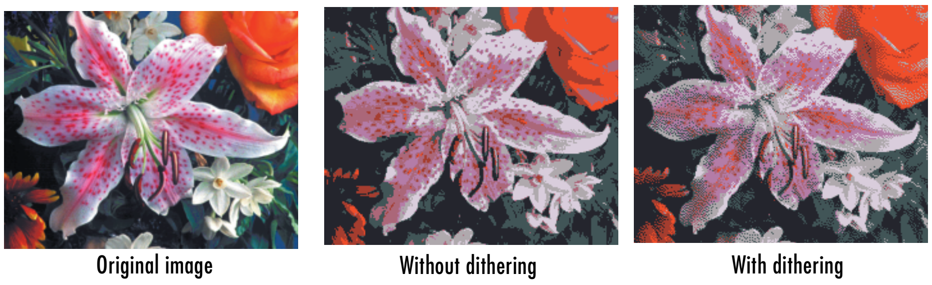

Figure 13-4: Examples of Color Reduction with and Without Dithering

Notice that the dithered image has a larger number of apparent colors but is somewhat fuzzy-looking. The image produced without dithering has fewer apparent colors, but an improved spatial resolution when compared to the dithered image. One risk in doing color reduction without dithering is that the new image my contain false contours (see the rose in the upper-right corner).

Converting to Other Color Spaces

The Image Processing Toolbox represents colors as RGB values, either directly (in an RGB image) or indirectly (in an indexed image, where the colormap is stored in RGB format). However, there are other models besides RGB for representing colors numerically. For example, a color can be represented by its hue, saturation, and value components (HSV) instead. The various models for color data are called *color spaces.*

The functions in the Image Processing Toolbox that work with color assume that images use the RGB color space. However, the toolbox provides support for other color spaces though a set of conversion functions. You can use these functions to convert between RGB and the following color spaces:

- National Television Systems Committee (NTSC)
- YCbCr
- Hue, saturation, value (HSV)

These section describes these color spaces and the conversion routines for working with them:

- "NTSC Color Space"
- "YCbCr Color Space" on page 13-16
- "HSV Color Space" on page 13-16

NTSC Color Space

The NTSC color space is used in televisions in the United States. One of the main advantages of this format is that grayscale information is separated from color data, so the same signal can be used for both color and black and white sets. In the NTSC format, image data consists of three components: luminance (Y), hue (I), and saturation (Q). The first component, luminance, represents grayscale information, while the last two components make up chrominance (color information).

The function `rgb2ntsc` converts colormaps or RGB images to the NTSC color space. `ntsc2rgb` performs the reverse operation.

For example, these commands convert the `flowers` image to NTSC format.

```
RGB = imread('flowers.tif');
YIQ = rgb2ntsc(RGB);
```

Because luminance is one of the components of the NTSC format, the RGB to NTSC conversion is also useful for isolating the gray level information in an image. In fact, the toolbox functions `rgb2gray` and `ind2gray` use the `rgb2ntsc` function to extract the grayscale information from a color image.

For example, these commands are equivalent to calling `rgb2gray`.

```
YIQ = rgb2ntsc(RGB);
I = YIQ(:,:,1);
```

Note In YIQ color space, I is one of the two color components, not the grayscale component.

YCbCr Color Space

The YCbCr color space is widely used for digital video. In this format, luminance information is stored as a single component (Y), and chrominance information is stored as two color-difference components (Cb and Cr). Cb represents the difference between the blue component and a reference value. Cr represents the difference between the red component and a reference value.

YCbCr data can be double precision, but the color space is particularly well suited to `uint8` data. For `uint8` images, the data range for Y is [16, 235], and the range for Cb and Cr is [16, 240]. YCbCr leaves room at the top and bottom of the full `uint8` range so that additional (nonimage) information can be included in a video stream.

The function `rgb2ycbcr` converts colormaps or RGB images to the YCbCr color space. `ycbcr2rgb` performs the reverse operation.

For example, these commands convert the `flowers` image to YCbCr format.

```
RGB = imread('flowers.tif');
YCBCR = rgb2ycbcr(RGB);
```

HSV Color Space

The HSV color space (hue, saturation, value) is often used by people who are selecting colors (e.g., of paints or inks) from a color wheel or palette, because it corresponds better to how people experience color than the RGB color space does. The functions `rgb2hsv` and `hsv2rgb` convert images between the RGB and HSV color spaces.

As hue varies from 0 to 1.0, the corresponding colors vary from red, through yellow, green, cyan, blue, and magenta, back to red, so that there are actually red values both at 0 and 1.0. As saturation varies from 0 to 1.0, the corresponding colors (hues) vary from unsaturated (shades of gray) to fully saturated (no white component). As value, or brightness, varies from 0 to 1.0, the corresponding colors become increasingly brighter.

Figure 13-5 illustrates the HSV color space.

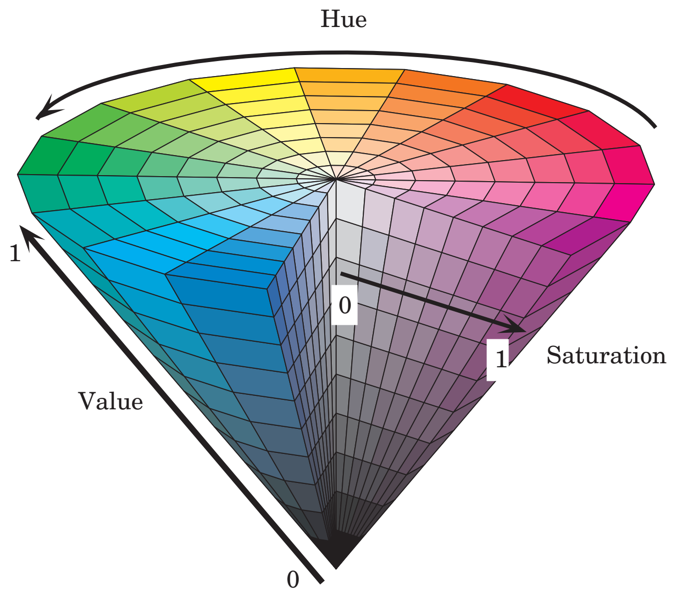

Figure 13-5: Illustration of the HSV Color Space

The function `rgb2hsv` converts colormaps or RGB images to the HSV color space. `hsv2rgb` performs the reverse operation. These commands convert an RGB image to HSV color space.

```
RGB = imread('flowers.tif');
HSV = rgb2hsv(RGB);
```

For closer inspection of the HSV color space, the next block of code displays the separate color planes (hue, saturation, and value) of an HSV image.

```
RGB=reshape(ones(64,1)*reshape(jet(64),1,192),[64,64,3]);
HSV=rgb2hsv(RGB);
H=HSV(:,:,1);
S=HSV(:,:,2);
V=HSV(:,:,3);
imshow(H)
figure, imshow(S);
figure, imshow(V);
figure, imshow(RGB);
```

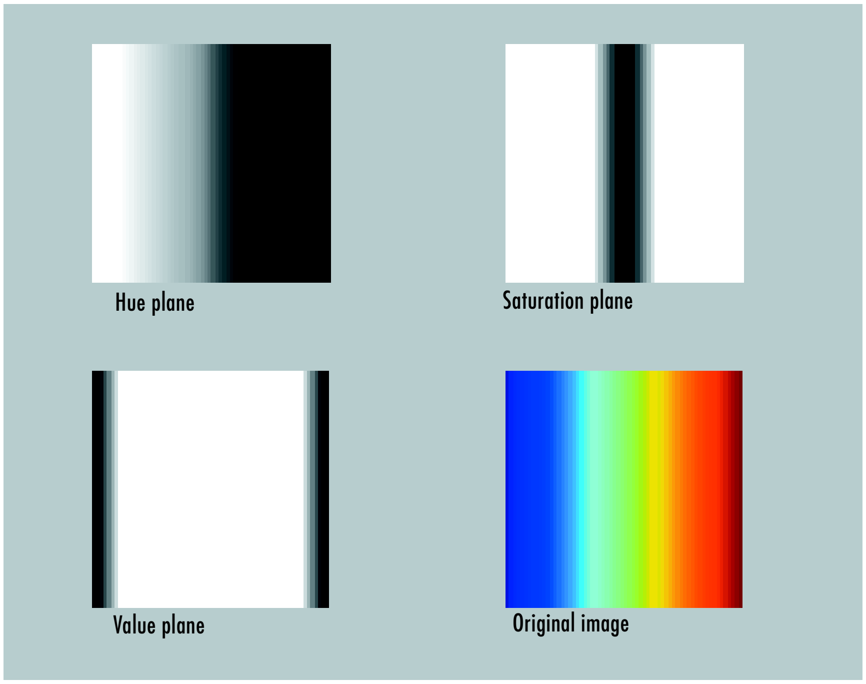

Figure 13-6: The Separated Color Planes of an HSV Image

The images in Figure 13-6 can be scrutinized for a better understanding of how the HSV color space works. As you can see by looking at the hue plane image, hue values make a nice linear transition from high to low. If you compare the hue plane image against the original image, you can see that shades of deep blue have the highest values, and shades of deep red have the lowest values. (In actuality, there are values of red on both ends of the hue scale, which you can see if you look back at the model of the HSV color space in Figure 13-5. To avoid confusion, our sample image uses only the red values from the *beginning* of the hue range.) Saturation can be thought of as the purity of a color. As the saturation plane image shows, the colors with the highest saturation have the highest values and are represented as white. In the center of the saturation image, notice the various shades of gray. These correspond to a mixture of colors; the cyans, greens, and yellow shades are mixtures of true colors. Value is roughly equivalent to brightness, and you will notice that the brightest areas of the value plane correspond to the brightest colors in the original image.