# 12

# Color Images

Color images are involved in every aspect of our lives, where they play an important role in everyday activities such as television, photography, and printing. Color perception is a fascinating and complicated phenomenon that has occupied the interests of scientists, psychologists, philosophers, and artists for hundreds of years [211, 217]. In this chapter, we focus on those technical aspects of color that are most important for working with digital color images. Our emphasis will be on understanding the various representations of color and correctly utilizing them when programming. Additional color-related issues, such as colorimetric color spaces, color quantization, and color filters, are covered in subsequent chapters.

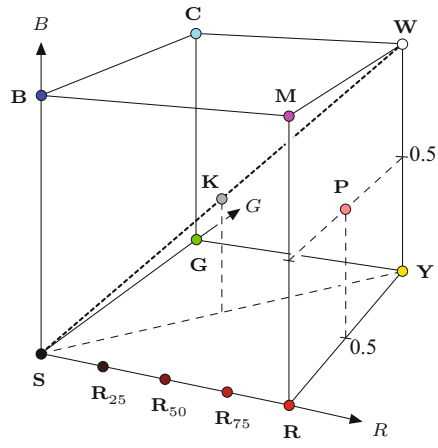
## 12.1 RGB Color Images

The RGB color schema encodes colors as combinations of the three primary colors: red, green, and blue $(R, G, B)$. This scheme is widely used for transmission, representation, and storage of color images on both analog devices such as television sets and digital devices such as computers, digital cameras, and scanners. For this reason, many image-processing and graphics programs use the RGB schema as their internal representation for color images, and most language libraries, including Java's imaging APIs, use it as their standard image representation.

RGB is an *additive* color system, which means that all colors start with black and are created by adding the primary colors. You can think of color formation in this system as occurring in a dark room where you can overlay three beams of light—one red, one green, and one blue—on a sheet of white paper. To create different colors, you would modify the intensity of each of these beams independently. The distinct intensity of each primary color beam controls the shade and brightness of the resulting color. The colors gray and white are created by mixing the three primary color beams at the same intensity. A similar operation occurs on the screen of a color television or




W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_12




RGB values

| Pt. | Color | $R$ | $G$ | $B$ |
|---|---|---|---|---|
| **S** | Black | 0.00 | 0.00 | 0.00 |
| **R** | Red | 1.00 | 0.00 | 0.00 |
| **Y** | Yellow | 1.00 | 1.00 | 0.00 |
| **G** | Green | 0.00 | 1.00 | 0.00 |
| **C** | Cyan | 0.00 | 1.00 | 1.00 |
| **B** | Blue | 0.00 | 0.00 | 1.00 |
| **M** | Magenta | 1.00 | 0.00 | 1.00 |
| **W** | White | 1.00 | 1.00 | 1.00 |
| **K** | 50% Gray | 0.50 | 0.50 | 0.50 |
| $\mathbf{R}_{75}$ | 75% Red | 0.75 | 0.00 | 0.00 |
| $\mathbf{R}_{50}$ | 50% Red | 0.50 | 0.00 | 0.00 |
| $\mathbf{R}_{25}$ | 25% Red | 0.25 | 0.00 | 0.00 |
| **P** | Pink | 1.00 | 0.50 | 0.50 |

**Fig. 12.1** Representation of the RGB color space as a 3D unit cube. The primary colors red ($R$), green ($G$), and blue ($B$) form the coordinate system. The "pure" red color (**R**), green (**G**), blue (**B**), cyan (**C**), magenta (**M**), and yellow (**Y**) lie on the vertices of the color cube. All the shades of gray, of which **K** is an example, lie on the diagonal between black **S** and white **W**.

CRT[1]-based computer monitor, where tiny, close-lying dots of red, green, and blue phosphorous are simultaneously excited by a stream of electrons to distinct energy levels (intensities), creating a seemingly continuous color image.

The RGB color space can be visualized as a 3D unit cube in which the three primary colors form the coordinate axis. The RGB values are positive and lie in the range $[0, C_{\max}]$; for most digital images, $C_{\max} = 255$. Every possible color $\mathbf{C}_i$ corresponds to a point within the RGB color cube of the form

$$\mathbf{C}_i = (R_i, G_i, B_i),$$

where $0 \leq R_i, G_i, B_i \leq C_{\max}$. RGB values are often normalized to the interval $[0, 1]$ so that the resulting color space forms a unit cube (Fig. 12.1). The point $\mathbf{S} = (0, 0, 0)$ corresponds to the color black, $\mathbf{W} = (1, 1, 1)$ corresponds to the color white, and all the points lying on the diagonal between **S** and **W** are shades of gray created from equal color components $R = G = B$.

Figure 12.2 shows a color test image and its corresponding RGB color components, displayed here as intensity images. We will refer to this image in a number of examples that follow in this chapter.

RGB is a very simple color system, and as demonstrated in Sec. 12.2, a basic knowledge of it is often sufficient for processing color images or transforming them into other color spaces. At this point, we will not be able to determine what color a particular RGB pixel corresponds to in the real world, or even what the primary colors red, green, and blue truly mean in a physical (i.e., colorimetric) sense. For now we rely on our intuitive understanding of color and will address colorimetry and color spaces later in the context of the CIE color system (see Ch. 14).

### 12.1.1 Structure of Color Images

Color images are represented in the same way as grayscale images, by using an array of pixels in which different models are used to order the

[1] Cathode ray tube.

*RGB*

*R*

*G*

*B*

**Fig. 12.2**
A color image and its corresponding RGB channels. The fruits depicted are mainly yellow and red and therefore have high values in the $R$ and $G$ channels. In these regions, the $B$ content is correspondingly lower (represented here by darker gray values) except for the bright highlights on the apple, where the color changes gradually to white. The tabletop in the foreground is purple and therefore displays correspondingly higher values in its $B$ channel.

individual color components. In the next sections we will examine the difference between *true color* images, which utilize colors uniformly selected from the entire color space, and so-called *palleted* or *indexed* images, in which only a select set of distinct colors are used. Deciding which type of image to use depends on the requirements of the application. Farbbilder werden üblicherweise, genau wie Grauwertbilder, als Arrays von Pixeln dargestellt, wobei unterschiedliche Modelle für die Anordnung der einzelnen Farbkomponenten verwendet werden. Zunächst ist zu unterscheiden zwischen *Vollfarbenbildern*, die den gesamten Farbraum gleichförmig abdecken können, und so genannten *Paletten-* oder *Indexbildern*, die nur eine beschränkte Zahl unterschiedlicher Farben verwenden. Beide Bildtypen werden in der Praxis häufig eingesetzt.

### True color images

A pixel in a true color image can represent any color in its color space, as long as it falls within the (discrete) range of its individual color components. True color images are appropriate when the image contains many colors with subtle differences, as occurs in digital photography and photo-realistic computer graphics. Next we look at two methods of ordering the color components in true color images: *component ordering* and *packed ordering*.

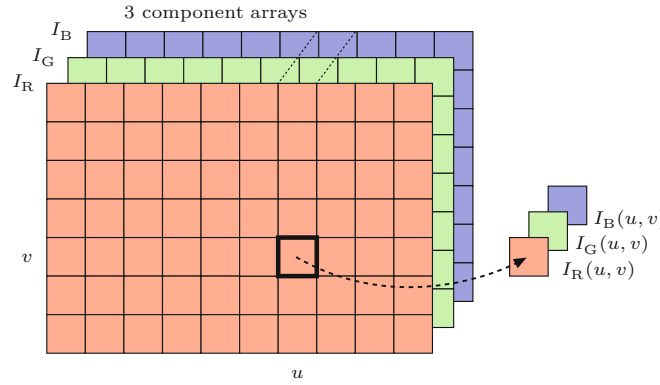



**Fig. 12.3** RGB color image in component ordering. The three color components are laid out in separate arrays $I_R$, $I_G$, $I_B$ of the same size.

*Component ordering*

In *component ordering* (also referred to as *planar ordering*) the color components are laid out in separate arrays of identical dimensions. In this case, the color image

$$\boldsymbol{I}_{\text{comp}} = (I_{\text{R}}, I_{\text{G}}, I_{\text{B}}) \tag{12.1}$$

can be thought of as a vector of related intensity images $I_R$, $I_G$, and $I_B$ (Fig. 12.3), and the RGB values of the color image $I$ at position $(u, v)$ are obtained by accessing the three component images in the form

$$\begin{pmatrix} R(u,v) \\ G(u,v) \\ B(u,v) \end{pmatrix} = \begin{pmatrix} I_{\text{R}}(u,v) \\ I_{\text{G}}(u,v) \\ I_{\text{B}}(u,v) \end{pmatrix}. \tag{12.2}$$

*Packed ordering*

In *packed ordering*, the component values that represent the color of a particular pixel are packed together into a single element of the image array (Fig. 12.4) such that

$$\boldsymbol{I}_{\text{pack}}(u,v) = (R, G, B). \tag{12.3}$$

The RGB value of a packed image $I$ at the location $(u, v)$ is obtained by accessing the individual components of the color pixel as

$$\begin{pmatrix} R(u,v) \\ G(u,v) \\ B(u,v) \end{pmatrix} = \begin{pmatrix} \mathsf{Red}(\boldsymbol{I}_{\text{pack}}(u,v)) \\ \mathsf{Green}(\boldsymbol{I}_{\text{pack}}(u,v)) \\ \mathsf{Blue}(\boldsymbol{I}_{\text{pack}}(u,v)) \end{pmatrix}. \tag{12.4}$$

The access functions Red(), Green(), Blue(), will depend on the specific implementation used for encoding the color pixels.

## Indexed images

Indexed images permit only a limited number of distinct colors and therefore are used mostly for illustrations and graphics that contain large regions of the same color. Often these types of images are stored in indexed GIF or PNG files for use on the Web. In these indexed

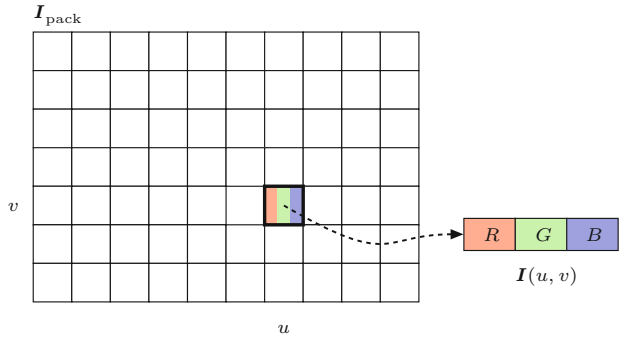



**Fig. 12.4**
RGB-color image using packed ordering. The three color components $R$, $G$, and $B$ are placed together in a single array element.

images, the pixel array does not contain color or brightness data but instead consists of integer numbers $k$ that are used to index into a *color table* or "palette"

$$\mathsf{P} = (\mathsf{P}_\text{r}, \mathsf{P}_\text{g}, \mathsf{P}_\text{g}) : [0, Q-1]^3 \;\mapsto\; [0, K-1]. \tag{12.5}$$

Here $Q$ denotes the size of the color table, equal to the maximum number of distinct image colors (typically $Q = 2, \ldots, 256$). $K$ is the number of distinct component values (typ. $K = 256$). This table contains a specific color vector $\mathsf{P}(q) = (R_q, G_q, B_q)$ for every color index $q = 0, \ldots, Q-1$ (see Fig. 12.5). The RGB component values of an indexed image $I_\text{idx}$ at position $(u, v)$ are obtained as

$$\begin{pmatrix} R(u,v) \\ G(u,v) \\ B(u,v) \end{pmatrix} = \begin{pmatrix} R_q \\ G_q \\ B_q \end{pmatrix} = \begin{pmatrix} \mathsf{P}_\text{r}(q) \\ \mathsf{P}_\text{g}(q) \\ \mathsf{P}_\text{b}(q) \end{pmatrix}, \tag{12.6}$$

with the index $q = I_\text{idx}(u, v)$. To allow proper reconstruction, the color table $\mathsf{P}$ must of course be stored and/or transmitted along with the indexed image.

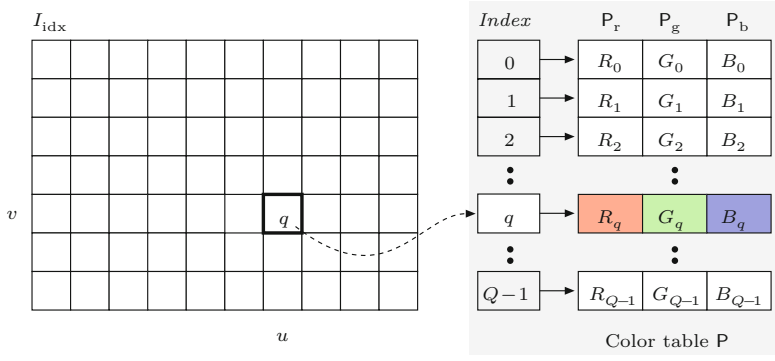



**Fig. 12.5**
RGB indexed image. The image array $I_\text{idx}$ itself does not contain any color component values. Instead, each cell contains an index $q \in [0, Q-1]$. into the associated color table ("palette") $\mathsf{P}$. The actual color value is specified by the table entry $\mathsf{P}_q = (R_q, G_q, B_q)$.

During the transformation from a true color image to an indexed image (e.g., from a JPEG image to a GIF image), the problem of optimal color reduction, or *color quantization*, arises. Color quantization is the process of determining an optimal color table and then mapping it to the original colors. This process is described in detail in Chapter 13.

### 12.1.2 Color Images in ImageJ

ImageJ provides two simple types of color images:

- RGB full-color images (24-bit "RGB color").
- Indexed images ("8-bit color").

#### RGB true color images

RGB color images in ImageJ use a packed order (see Sec. 12.1.1), where each color pixel is represented by a 32-bit `int` value. As Fig. 12.6 illustrates, 8 bits are used to represent each of the RGB components, which limits the range of the individual components to 0–255. The remaining 8 bits are reserved for the transparency,[2] or *alpha* ($\alpha$), component. This is also the usual ordering in Java[3] for RGB color images.

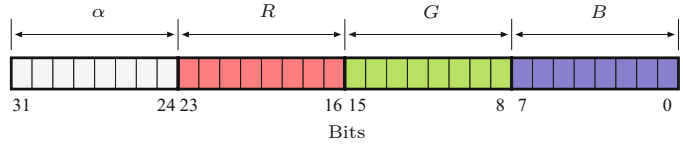



**Fig. 12.6** Structure of a packed RGB color pixel in Java. Within a 32-bit `int`, 8 bits are allocated, in the following order, for each of the color components $R$, $G$, $B$, and the transparency value $\alpha$ (unused in ImageJ).

##### *Accessing RGB pixel values*

RGB color images are represented by an array of pixels, the elements of which are standard Java `int`s. To disassemble the packed `int` value into the three color components, you apply the appropriate bitwise shifting and masking operations. In the following example, we assume that the image processor `ip` (of type `ColorProcessor`) contains an RGB color image:

```
int c = ip.getPixel(u,v);     // a packed RGB color pixel
int r = (c & 0xff0000) >> 16; // red component
int g = (c & 0x00ff00) >> 8;  // green component
int b = (c & 0x0000ff);       // blue component
```

In this example, each of the RGB components of the packed pixel `c` are isolated using a bitwise AND operation (`&`) with an appropriate bit mask (following convention, bit masks are given in hexadecimal[4] notation), and afterwards the extracted bits are shifted right by 16 (for $R$) or 8 (for $G$) bit positions (see Fig. 12.7).

The "assembly" of an RGB pixel from separate $R$, $G$, and $B$ values works in the opposite direction using the bitwise OR operator (`|`) and shifting the bits left (`<<`):

```
int r = 169;   // red component
int g = 212;   // green component
int b = 17;    // blue component
int c = ((r & 0xff) << 16) | ((g & 0xff) << 8) | b & 0xff;
ip.putPixel(u, v, c);
```

[2] The transparency value $\alpha$ (alpha) represents the ability to see through a color pixel onto the background. At this time, the $\alpha$ channel is unused in ImageJ.

[3] Java Advanced Window Toolkit (AWT).

[4] The mask `0xff0000` is of type `int` and represents the 32-bit binary pattern `00000000111111110000000000000000`.

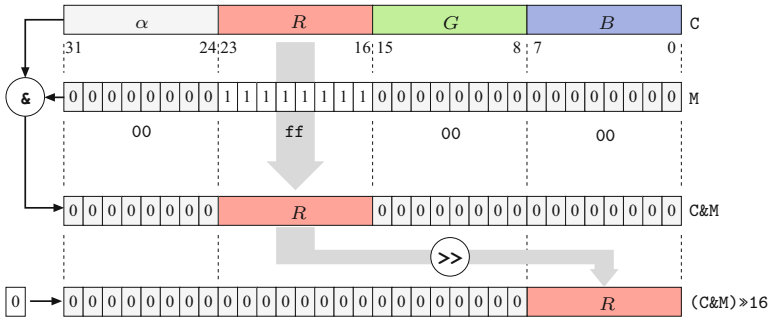



**Fig. 12.7**
Decomposition of a 32-bit RGB color pixel using bit operations. The $R$ component (bits 16–23) of the RGB pixels `C` (above) is isolated using a bitwise AND operation (`&`) together with a bit mask `M = 0xff0000`. All bits except the $R$ component are set to the value 0, while the bit pattern within the $R$ component remains unchanged. This bit pattern is subsequently shifted 16 positions to the right (`>>`), so that the $R$ component is moved into the lowest 8 bits and its value lies in the range of $0, \ldots, 255$. During the shift operation, zeros are filled in from the left.

**Prog. 12.1**
Processing RGB color data with the use of bit operations (ImageJ plugin, version 1). This plugin increases the values of all three color components by 10 units. It demonstrates the use of direct access to the pixel array (line 16), the separation of color components using bit operations (lines 18–20), and the reassembly of color pixels after modification (line 27). The value `DOES_RGB` (defined in the interface `PlugInFilter`) returned by the `setup()` method indicates that this plugin is designed to work on RGB formatted true color images (line 9).

```
1  // File Brighten_RGB_1.java
2  import ij.ImagePlus;
3  import ij.plugin.filter.PlugInFilter;
4  import ij.process.ImageProcessor;
5
6  public class Brighten_RGB_1 implements PlugInFilter {
7
8    public int setup(String arg, ImagePlus imp) {
9      return DOES_RGB;   // this plugin works on RGB images
10   }
11
12   public void run(ImageProcessor ip) {
13     int[] pixels = (int[]) ip.getPixels();
14
15     for (int i = 0; i < pixels.length; i++) {
16       int c = pixels[i];
17       // split color pixel into rgb-components:
18       int r = (c & 0xff0000) >> 16;
19       int g = (c & 0x00ff00) >> 8;
20       int b = (c & 0x0000ff);
21       // modify colors:
22       r = r + 10; if (r > 255) r = 255;
23       g = g + 10; if (g > 255) g = 255;
24       b = b + 10; if (b > 255) b = 255;
25       // reassemble color pixel and insert into pixel array:
26       pixels[i]
27         = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
28     }
29   }
30 }
```

Masking the component values with `0xff` works in this case because, except for the bits in positions $0, \ldots, 7$ (values in the range 0–255), all the other bits are already set to zero. A complete example of manipulating an RGB color image using bit operations is presented in Prog. 12.1. Instead of accessing color pixels using ImageJ's access functions, these programs directly access the pixel array for increased efficiency

The ImageJ class `ColorProcessor` provides an easy to use alternative which returns the separated RGB components (as an `int` array

**Prog. 12.2** Working with RGB color images without bit operations (ImageJ plugin, version 2). This plugin increases the values of all three color components by 10 units using the access methods `getPixel(int, int, int[])` and `putPixel(int, int, int[])` from the class `ColorProcessor` (lines 21 and 25, respectively). Exection time is approximately four times higher than that of version 1 (Prog. 12.1) because of the additional method calls.

```
1  // File Brighten_RGB_2.java
2  import ij.ImagePlus;
3  import ij.plugin.filter.PlugInFilter;
4  import ij.process.ColorProcessor;
5  import ij.process.ImageProcessor;
6
7  public class Brighten_RGB_2 implements PlugInFilter {
8    static final int R = 0, G = 1, B = 2; // component indices
9
10   public int setup(String arg, ImagePlus imp) {
11     return DOES_RGB;   // this plugin works on RGB images
12   }
13
14   public void run(ImageProcessor ip) {
15     // typecast the image to ColorProcessor (no duplication):
16     ColorProcessor cp = (ColorProcessor) ip;
17     int[] RGB = new int[3];
18
19     for (int v = 0; v < cp.getHeight(); v++) {
20       for (int u = 0; u < cp.getWidth(); u++) {
21         cp.getPixel(u, v, RGB);
22         RGB[R] = Math.min(RGB[R] + 10, 255);   // add 10 and
23         RGB[G] = Math.min(RGB[G] + 10, 255);   // limit to 255
24         RGB[B] = Math.min(RGB[B] + 10, 255);
25         cp.putPixel(u, v, RGB);
26       }
27     }
28   }
29 }
```

with three elements). In the following example, which demonstrates its use, `ip` is of type `ColorProcessor`:

```
int[] RGB = new int[3];
...
ip.getPixel(u, v, RGB); // modifies RGB
int r = RGB[0];
int g = RGB[1];
int b = RGB[2];
...
ip.putPixel(u, v, RGB);
```

A more detailed and complete example is shown by the simple plugin in Prog. 12.2, which increases the value of all three color components of an RGB image by 10 units. Notice that the plugin limits the resulting component values to 255, because the `putPixel()` method only uses the lowest 8 bits of each component and does not test if the value passed in is out of the permitted 0–255 range. Without this test, arithmetic overflow errors can occur. The price for using this access method, instead of direct array access, is a noticeably longer running time (approximately a factor of 4 when compared to the version in Prog. 12.1).

*Opening and saving RGB images*

ImageJ supports the following types of image formats for RGB true color images:

- **TIFF** (uncompressed only): 3 × 8-bit RGB. TIFF color images with 16-bit depth are opened as an image stack consisting of three 16-bit intensity images.
- **BMP**, **JPEG**: 3 × 8-bit RGB.
- **PNG**: 3 × 8-bit RGB.
- **RAW**: using the ImageJ menu File ▷ Import ▷ Raw, RGB images can be opened whose format is not directly supported by ImageJ. It is then possible to select different arrangements of the color components.

*Creating RGB color images*

The simplest way to create a new RGB image using ImageJ is to use an instance of the class `ColorProcessor`, as the following example demonstrates:

```
int w = 640, h = 480;
ColorProcessor cp = new ColorProcessor(w, h);
(new ImagePlus("My New Color Image", cp)).show();
```

When needed, the color image can be displayed by creating an instance of the class `ImagePlus` and calling its `show()` method. Since `cip` is of type `ColorProcessor`, the resulting `ImagePlus` object `cimg` is also a color image.

### Indexed color images

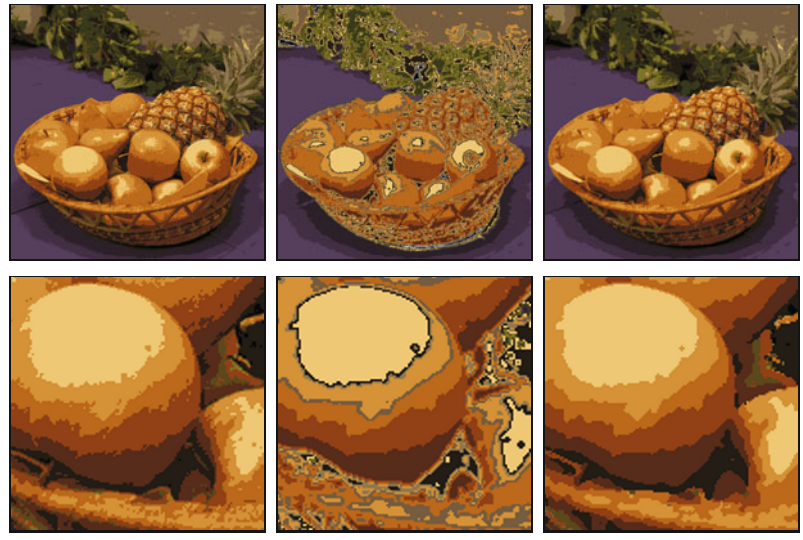
The structure of an indexed image in ImageJ is given in Fig. 12.5, where each element of the index array is 8 bits and therefore can represent a maximum of 256 different colors. When programming, indexed images are similar to grayscale images, as both make use of a color table to determine the actual color of the pixel. Indexed images differ from grayscale images only in that the contents of the color table are not intensity values but RGB values.

*Opening and saving indexed images*

ImageJ supports the indexed images in GIF, PNG, BMP, and TIFF format with index values of 1–8 bits (i.e., 2–256 distinct colors) and 3 × 8-bit color values.

*Processing indexed images*

The indexed format is mostly used as a space-saving means of image storage and is not directly useful as a processing format since an index value in the pixel array is arbitrarily related to the actual color, found in the color table, that it represents. When working with indexed images it usually makes no sense to base any numerical interpretations on the pixel values or to apply any filter operations designed for 8-bit intensity images. Figure 12.8 illustrates an example of applying a Gaussian filter and a median filter to the pixels of an indexed image. Since there is no meaningful quantitative relation between the actual colors and the index values, the results are erratic.



(a) (b) (c)

**Fig. 12.8** Improper application of smoothing filters to an indexed color image. Indexed image with 16 colors (a) and results of applying a linear smoothing filter (b) and a $3 \times 3$ median filter (c) to the pixel array (that is, the *index* values). The application of a linear filter makes no sense, of course, since no meaningful relation exists between the index values in the pixel array and the actual image intensities. While the median filter (c) delivers seemingly plausible results in this case, its use is also inadmissible because no meaningful ordering relation exists between the index values.

Note that even the use of the median filter is inadmissible because no ordering relation exists between the index values. Thus, with few exceptions, ImageJ functions do not permit the application of such operations to indexed images. Generally, when processing an indexed image, you first convert it into a true color RGB image and then after processing convert it back into an indexed image.

When an ImageJ plugin is supposed to process indexed images, its `setup()` method should return the `DOES_8C` ("8-bit color") flag. The plugin in Prog. 12.3 shows how to increase the intensity of the three color components of an indexed image by 10 units (analogously to Progs. 12.1 and 12.2 for RGB images). Notice how in indexed images only the palette is modified and the original pixel data, the index values, remain the same. The color table of `ImageProcessor` is accessible through a `ColorModel`[5] object, which can be read using the method `getColorModel()` and modified using `setColorModel()`.

The `ColorModel` object for indexed images (as well as 8-bit grayscale images) is a subtype of `IndexColorModel`, which contains three color tables (*maps*) representing the red, green, and blue components as separate `byte` arrays. The size of these tables $(2, \ldots, 256)$ can be determined by calling the method `getMapSize()`. Note that the elements of the palette should be interpreted as *unsigned* bytes with values ranging from $0, \ldots, 255$. Just as with grayscale pixel values, during the conversion to `int` values, these color component values must also be bitwise masked with `0xff` as shown in Prog. 12.3 (lines 30–32).

As a further example, Prog. 12.4 shows how to convert an indexed image to a true color RGB image of type `ColorProcessor`. Conversion in this direction poses no problems because the RGB component values for a particular pixel are simply taken from the corresponding color table entry, as described by Eqn. (12.6). On the other hand,

[5] Defined in the standard Java class `java.awt.image.ColorModel`.

**Prog. 12.3**
Working with indexed images (ImageJ plugin). This plugin increases the brightness of an image by 10 units by modifying the image's color table (palette). The actual values in the pixel array, which are indices into the palette, are not changed.

```
1  // File Brighten_Index_Image.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ImageProcessor;
6  import java.awt.image.IndexColorModel;
7
8  public class Brighten_Index_Image implements PlugInFilter {
9
10   public int setup(String arg, ImagePlus imp) {
11     return DOES_8C; // this plugin works on indexed color images
12   }
13
14   public void run(ImageProcessor ip) {
15     IndexColorModel icm =
16       (IndexColorModel) ip.getColorModel();
17     int pixBits = icm.getPixelSize();
18     int nColors = icm.getMapSize();
19
20     //retrieve the current lookup tables (maps) for R, G, B:
21     byte[] pRed = new byte[nColors];
22     byte[] pGrn = new byte[nColors];
23     byte[] pBlu = new byte[nColors];
24     icm.getReds(pRed);
25     icm.getGreens(pGrn);
26     icm.getBlues(pBlu);
27
28     //modify the lookup tables:
29     for (int idx = 0; idx < nColors; idx++){
30       int r = 0xff & pRed[idx]; // mask to treat as unsigned byte
31       int g = 0xff & pGrn[idx];
32       int b = 0xff & pBlu[idx];
33       pRed[idx] = (byte) Math.min(r + 10, 255);
34       pGrn[idx] = (byte) Math.min(g + 10, 255);
35       pBlu[idx] = (byte) Math.min(b + 10, 255);
36     }
37     //create a new color model and apply to the image:
38     IndexColorModel icm2 =
39       new IndexColorModel(pixBits,nColors,pRed,pGrn,pBlu);
40     ip.setColorModel(icm2);
41   }
42 }
```

conversion in the other direction requires *quantization* of the RGB color space and is as a rule more difficult and involved (see Ch. 13 for details). In practice, most applications make use of existing conversion methods such as those provided by the ImageJ API.

### *Creating indexed images*

In ImageJ, no special method is provided for the creation of indexed images, so in almost all cases they are generated by converting an existing image. The following method demonstrates how to directly create an indexed image if required:

```
ByteProcessor makeIndexColorImage(int w, int h, int nColors) {
```

**Prog. 12.4** Converting an indexed image to a true color RGB image (ImageJ plugin).

```
1  // File Index_To_Rgb.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ColorProcessor;
6  import ij.process.ImageProcessor;
7  import java.awt.image.IndexColorModel;
8
9  public class Index_To_Rgb implements PlugInFilter {
10   static final int R = 0, G = 1, B = 2;
11   ImagePlus imp;
12
13   public int setup(String arg, ImagePlus imp) {
14     this.imp = imp;
15     return DOES_8C + NO_CHANGES;   // does not alter original image
16   }
17
18   public void run(ImageProcessor ip) {
19     int w = ip.getWidth();
20     int h = ip.getHeight();
21
22     // retrieve the lookup tables (maps) for R, G, B:
23     IndexColorModel icm =
24         (IndexColorModel) ip.getColorModel();
25     int nColors = icm.getMapSize();
26     byte[] pRed = new byte[nColors];
27     byte[] pGrn = new byte[nColors];
28     byte[] pBlu = new byte[nColors];
29     icm.getReds(pRed);
30     icm.getGreens(pGrn);
31     icm.getBlues(pBlu);
32
33     // create a new 24-bit RGB image:
34     ColorProcessor cp = new ColorProcessor(w, h);
35     int[] RGB = new int[3];
36     for (int v = 0; v < h; v++) {
37       for (int u = 0; u < w; u++) {
38         int idx = ip.getPixel(u, v);
39         RGB[R] = 0xFF & pRed[idx];
40         RGB[G] = 0xFF & pGrn[idx];
41         RGB[B] = 0xFF & pBlu[idx];
42         cp.putPixel(u, v, RGB);
43       }
44     }
45     ImagePlus cwin =
46         new ImagePlus(imp.getShortTitle() + " (RGB)", cp);
47     cwin.show();
48   }
49 }
```

```
byte[] rMap = new byte[nColors]; // red, green, blue color maps
byte[] gMap = new byte[nColors];
byte[] bMap = new byte[nColors];
// color maps need to be filled here
byte[] pixels = new byte[w * h];
```

```
  IndexColorModel cm
    = new IndexColorModel(8, nColors, rMap, gMap, bMap);
  return new ByteProcessor(w, h, pixels, cm);
}
```

The parameter `nColors` defines the number of colors (and thus the size of the palette) and must be a value in the range of $2, \ldots, 256$. To use the above template, you would complete it with code that filled the three `byte` arrays for the RGB components (`rMap`, `gMap`, `bMap`) and the index array (`pixels`) with the appropriate values.

*Transparency*

Transparency is one of the reasons indexed images are often used for Web graphics. In an indexed image, it is possible to define one of the index values so that it is displayed in a transparent manner and at selected image locations the background beneath the image shows through. In Java this can be controlled when creating the image's color model (`IndexColorModel`). As an example, to make color index 2 in Prog. 12.3 transparent, line 39 would need to be modified as follows:

```
int tidx = 2; // index of transparent color
IndexColorModel icm2 =
  new IndexColorModel(pixBits,nColors,pRed,pGrn,pBlu,tidx);
ip.setColorModel(icm2);
```

At this time, however, ImageJ does not support the transparency property; it is not considered during display, and it is lost when the image is saved.

## 12.2 Color Spaces and Color Conversion

The RGB color system is well-suited for use in programming, as it is simple to manipulate and maps directly to the typical display hardware. When modifying colors within the RGB space, it is important to remember that the *metric*, or *measured distance* within this color space, does not proportionally correspond to our perception of color (e.g., doubling the value of the red component does not necessarily result in a color which appears to be twice as red). In general, in this space, modifying different color points by the same amount can cause very different changes in color. In addition, brightness changes in the RGB color space are also perceived as nonlinear.

Since changing any component modifies color tone, saturation, and brightness all at once, color selection in RGB space is difficult and quite non-intuitive. Color selection is more intuitive in other color spaces, such as the HSV space (see Sec. 12.2.3), since perceptual color features, such as saturation, are represented individually and can be modified independently. Alternatives to the RGB color space are also used in applications such as the automatic separation of objects from a colored background (the *blue box* technique in television), encoding television signals for transmission, or in printing, and are thus also relevant in digital image processing.

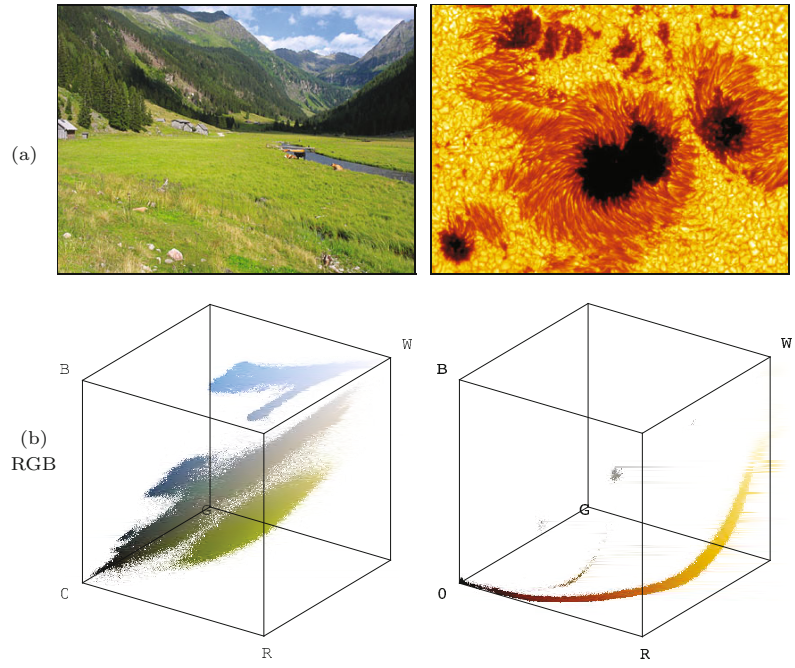



**Fig. 12.9**
Examples of the color distribution of natural images. Original images: landscape photograph with dominant green and blue components and sun-spot image with rich red and yellow components (a). Distribution of image colors in RGB-space (b).

Figure 12.9 shows the distribution of the colors from natural images in the RGB color space. The first half of this section introduces alternative color spaces and the methods of converting between them, and later discusses the choices that need to be made to correctly convert a color image to grayscale. In addition to the classical color systems most widely used in programming, precise reference systems, such as the CIEXYZ color space, gain increasing importance in practical color processing.

### 12.2.1 Conversion to Grayscale

The conversion of an RGB color image to a grayscale image proceeds by computing the equivalent gray or *luminance* value $Y$ for each RGB pixel. In its simplest form, $Y$ could be computed as the average

$$Y = \text{Avg}(R, G, B) = \frac{R + G + B}{3} \tag{12.7}$$

of the three color components $R$, $G$, and $B$. Since we perceive both red and green as being substantially brighter than blue, the resulting image will appear to be too dark in the red and green areas and too bright in the blue ones. Therefore, a weighted sum of the color components is typically used for calculating the equivalent brightness or *luminance* in the form

$$Y = \text{Lum}(R, G, B) = w_{\text{R}} \cdot R + w_{\text{G}} \cdot G + w_{\text{B}} \cdot B \tag{12.8}$$

The weights most often used were originally developed for encoding analog color television signals (see Sec. 12.2.4) are

$$w_{\text{R}} = 0.299, \qquad w_{\text{G}} = 0.587, \qquad w_{\text{B}} = 0.114, \tag{12.9}$$

and the weights recommended in ITU-BT.709 [122] for digital color encoding are

$$w_{\mathrm{R}} = 0.2126, \qquad w_{\mathrm{G}} = 0.7152, \qquad w_{\mathrm{B}} = 0.0722. \tag{12.10}$$

If each color component is assigned the same weight, as in Eqn. (12.7), this is of course just a special case of Eqn. (12.8).

Note that, although these weights were developed for use with TV signals, they are optimized for *linear* RGB component values, that is, signals with no gamma correction. In many practical situations, however, the RGB components are actually *nonlinear*, particularly when we work with sRGB images (see Ch. 14, Sec. 14.4). In this case, the RGB components must first be linearized to obtain the correct luminance values with the aforementioned weights.

In some color systems, instead of a weighted sum of the RGB color components, a nonlinear brightness function, for example the *value* $V$ in HSV (Eqn. (12.14) in Sec. 12.2.3) or the *luminance* $L$ in HLS (Eqn. (12.25)), is used as the intensity value $Y$.

### Hueless (gray) color images

An RGB image is hueless or gray when the RGB components of each pixel $\boldsymbol{I}(u, v) = (R, G, B)$ are the same; that is, if

$$R = G = B.$$

Therefore, to completely remove the color from an RGB image, simply replace the $R$, $G$, and $B$ component of each pixel with the equivalent gray value $Y$,

$$\begin{pmatrix} R_{\mathrm{gray}} \\ G_{\mathrm{gray}} \\ B_{\mathrm{gray}} \end{pmatrix} = \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix}, \tag{12.11}$$

by using $Y = \mathrm{Lum}(R, G, B)$ from Eqns. (12.8) and (12.9), for example. The resulting grayscale image should have the same subjective brightness as the original color image.

### Grayscale conversion in ImageJ

In ImageJ, the simplest way to convert an RGB color image (of type `ColorProcessor`) into an 8-bit grayscale image is to use the `ImageProcessor`-method

```
convertToByteProcessor(),
```

which returns a new image of type `ByteProcessor`. ImageJ uses the default weights $w_R = w_G = w_B = \frac{1}{3}$ (as in Eqn. (12.7)) for the RGB components, or alternatively $w_R = 0.299$, $w_G = 0.587$, $w_B = 0.114$ (as in Eqn. (12.9)) if the "Weighted RGB Conversions" option is selected in the Edit ▷ Options ▷ Conversions dialog. Arbitrary component weights can be specified for subsequent conversion operations through the static `ColorProcessor` method

```
setRGBWeights(double wR, double wG, double wB).
```

Similarly, the static method `getWeightingFactors()` of class `ColorProcessor` can be used to retrieve the current component weights as a 3-element `double`-array. Note that no *linearization* is performed on the color components, which should be considered when working with (nonlinear) sRGB colors (see Ch. 14, Sec. 14.4 for details).

### 12.2.2 Desaturating RGB Color Images

Desaturation is the uniform reduction of the amount of color in an RGB image in a *continuous* manner. It is done by replacing each RGB pixel by a desaturated color obtained by linear interpolation between the pixel's original color and the corresponding $(Y, Y, Y)$ gray point in the RGB space, that is,

$$\begin{pmatrix} R_{\text{desat}} \\ G_{\text{desat}} \\ B_{\text{desat}} \end{pmatrix} = \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix} + s \cdot \begin{pmatrix} R - Y \\ G - Y \\ B - Y \end{pmatrix}, \tag{12.12}$$

again with $Y = \text{Lum}(R, G, B)$ from Eqns. (12.8) and (12.9), where the factor $s \in [0, 1]$ controls the remaining amount of color saturation (Fig. 12.10). A value of $s = 0$ completely eliminates all color, resulting in a true grayscale image, and with $s = 1$ the color values will be unchanged. In Prog. 12.5, continuous desaturation as defined in Eqn. (12.12) is implemented as an ImageJ plugin.

In color spaces where color saturation is represented by an explicit component (such as HSV and HLS, for example), desaturation is of course much easier to accomplish (by simply reducing the saturation value to zero).

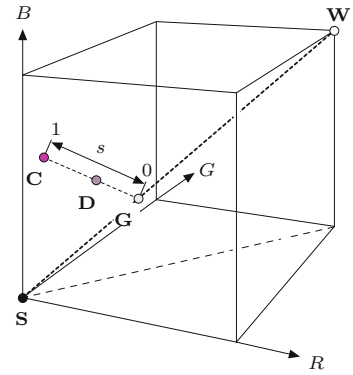



**Fig. 12.10**
Desaturation in RGB space: original color point $\mathbf{C} = (R, G, B)$, its corresponding gray point $\mathbf{G} = (Y, Y, Y)$, and the desaturated color point $\mathbf{D}$. Saturation is controlled by the factor $s$.

### 12.2.3 HSV/HSB and HLS Color Spaces

In the **HSV** color space, colors are specified by the components *hue*, *saturation*, and *value*. Often, such as in Adobe products and the Java API, the **HSV** space is called **HSB**. While the acronym is different (in this case B = *brightness*),[6] it denotes the same color space. The HSV color space is traditionally shown as an upside-down, six-sided pyramid (Fig. 12.11(a)), where the vertical axis represents the $V$ (brightness) value, the horizontal distance from the axis the $S$ (saturation) value, and the angle the $H$ (hue) value. The black point is at the tip of the pyramid and the white point lies in the center of the base. The three primary colors *red*, *green*, and *blue* and the pairwise mixed colors *yellow*, *cyan*, and *magenta* are the corner points of the

[6] Sometimes the HSV space is also referred to as the "HSI" space, where "I" stands for *intensity*.

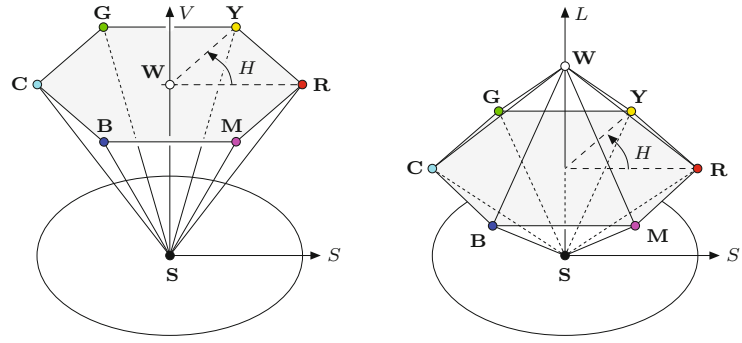
**Prog. 12.5**
Continuous desaturation of an RGB color image (ImageJ plugin). The amount of color saturation is controlled by the variable s defined in line 8 (see Eqn. (12.12)).

```
1  // File Desaturate_Rgb.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ImageProcessor;
6
7  public class Desaturate_Rgb implements PlugInFilter {
8    double s = 0.3; // color saturation value
9
10   public int setup(String arg, ImagePlus imp) {
11     return DOES_RGB;
12   }
13
14   public void run(ImageProcessor ip) {
15     //iterate over all pixels:
16     for (int v = 0; v < ip.getHeight(); v++) {
17       for (int u = 0; u < ip.getWidth(); u++) {
18
19         // get int-packed color pixel:
20         int c = ip.get(u, v);
21
22         //extract RGB components from color pixel
23         int r = (c & 0xff0000) >> 16;
24         int g = (c & 0x00ff00) >> 8;
25         int b = (c & 0x0000ff);
26
27         // compute equiv. gray value:
28         double y = 0.299 * r + 0.587 * g + 0.114 * b;
29
30         // linear interpolate (yyy) ↔ (rgb):
31         r = (int) (y + s * (r - y));
32         g = (int) (y + s * (g - y));
33         b = (int) (y + s * (b - y));
34
35         // reassemble the color pixel:
36         c = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
37         ip.set(u, v, c);
38       }
39     }
40   }
41
42 }
```

base. While this space is often represented as a pyramid, according to its mathematical definition, the space is actually a *cylinder*, as shown in Fig. 12.12.

The **HLS** color space[7] (*hue*, *luminance*, *saturation*) is very similar to the HSV space, and the *hue* component is in fact completely identical in both spaces. The *luminance* and *saturation* values also correspond to the vertical axis and the radius, respectively, but are defined differently than in HSV space. The common representation of the HLS space is as a double pyramid (Fig. 12.11(b)), with black

[7] The acronyms HLS and HSL are used interchangeably.




(a) HSV (b) HLS

**Fig. 12.11** HSV and HLS color space are traditionally visualized as a single or double hexagonal pyramid. The brightness $V$ (or $L$) is represented by the vertical dimension, the color saturation $S$ by the radius from the pyramid's axis, and the hue $h$ by the angle. In both cases, the primary colors red (**R**), green (**G**), and blue (**B**) and the mixed colors yellow (**Y**), cyan (**C**), and magenta (**M**) lie on a common plane with black (**S**) at the tip. The essential difference between the HSV and HLS color spaces is the location of the white point (**W**).

on the bottom tip and white on the top. The primary colors lie on the corner points of the hexagonal base between the two pyramids. Even though it is often portrayed in this intuitive way, mathematically the HLS space is again a cylinder (see Fig. 12.15).

### RGB→HSV conversion

To convert from RGB to the HSV color space, we first find the *saturation* of the RGB color components $R, G, B \in [0, C_{\max}]$, with $C_{\max}$ being the maximum component value (typically 255), as

$$S_{\text{HSV}} = \begin{cases} \frac{C_{\text{rng}}}{C_{\text{high}}} & \text{for } C_{\text{high}} > 0, \\ 0 & \text{otherwise} \end{cases} \tag{12.13}$$

and the brightness (*value*)

$$V_{\text{HSV}} = \frac{C_{\text{high}}}{C_{\max}}, \tag{12.14}$$

with

$$\begin{aligned} C_{\text{low}} &= \min(R, G, B)\,, \qquad C_{\text{high}} = \max(R, G, B)\,, \\ C_{\text{rng}} &= C_{\text{high}} - C_{\text{low}}. \end{aligned} \tag{12.15}$$

Finally, we need to specify the *hue* value $H_{\text{HSV}}$. When all three RGB color components have the same value ($R = G = B$), then we are dealing with an *achromatic* (gray) pixel. In this particular case $C_{\text{rng}} = 0$ and thus the saturation value $S_{\text{HSV}} = 0$, consequently the hue is undefined. To calculate $H_{\text{HSV}}$ when $C_{\text{rng}} > 0$, we first normalize each component using

$$R' = \frac{C_{\text{high}} - R}{C_{\text{rng}}}, \qquad G' = \frac{C_{\text{high}} - G}{C_{\text{rng}}}, \qquad B' = \frac{C_{\text{high}} - B}{C_{\text{rng}}}. \tag{12.16}$$

Then, depending on which of the three original color components had the maximal value, we compute a preliminary hue $H'$ as

$$H' = \begin{cases} B' - G' & \text{for } R = C_{\text{high}}, \\ R' - B' + 2 & \text{for } G = C_{\text{high}}, \\ G' - R' + 4 & \text{for } B = C_{\text{high}}. \end{cases} \tag{12.17}$$

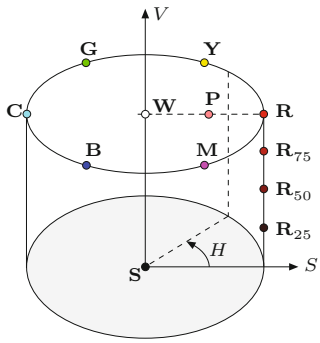
Since the resulting value for $H'$ lies on the interval $[-1, 5]$, we obtain the final hue value by normalizing to the interval $[0, 1]$ as

$$H_{\text{HSV}} = \frac{1}{6} \cdot \begin{cases} (H' + 6) & \text{for } H' < 0, \\ H' & \text{otherwise.} \end{cases} \tag{12.18}$$

Hence all three components $H_{\text{HSV}}$, $S_{\text{HSV}}$, and $V_{\text{HSV}}$ will lie within the interval $[0, 1]$. The hue value $H_{\text{HSV}}$ can naturally also be computed in another angle interval, for example, in the 0 to 360° interval using

$$H^{\circ}_{\text{HSV}} = H_{\text{HSV}} \cdot 360. \tag{12.19}$$

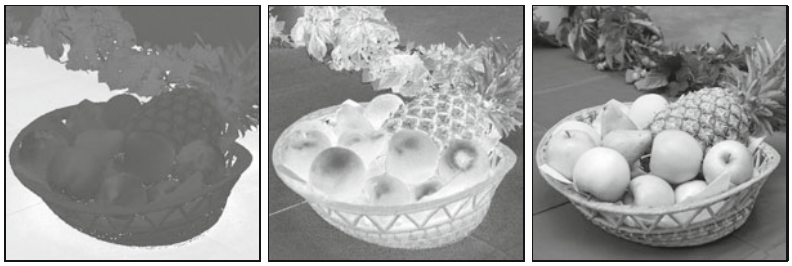
Under this definition, the RGB space unit cube is mapped to a *cylinder* with height and radius of length 1 (Fig. 12.12). In contrast to the traditional representation (Fig. 12.11), all HSB points within the entire cylinder correspond to valid color coordinates in RGB space. The mapping from RGB to the HSV space is nonlinear, as can be noted by examining how the black point stretches completely across the cylinder's base. Figure 12.12 plots the location of some notable color points and compares them with their locations in RGB space (see also Fig. 12.1). Figure 12.13 shows the individual HSV components (in grayscale) of the test image in Fig. 12.2.




RGB/HSV values

| Pt. | Color | $R$ | $G$ | $B$ | $H$ | $S$ | $V$ |
|---|---|---|---|---|---|---|---|
| **S** | Black | 0.00 | 0.00 | 0.00 | — | 0.00 | 0.00 |
| **R** | Red | 1.00 | 0.00 | 0.00 | 0 | 1.00 | 1.00 |
| **Y** | Yellow | 1.00 | 1.00 | 0.00 | 1/6 | 1.00 | 1.00 |
| **G** | Green | 0.00 | 1.00 | 0.00 | 2/6 | 1.00 | 1.00 |
| **C** | Cyan | 0.00 | 1.00 | 1.00 | 3/6 | 1.00 | 1.00 |
| **B** | Blue | 0.00 | 0.00 | 1.00 | 4/6 | 1.00 | 1.00 |
| **M** | Magenta | 1.00 | 0.00 | 1.00 | 5/6 | 1.00 | 1.00 |
| **W** | White | 1.00 | 1.00 | 1.00 | — | 0.00 | 1.00 |
| $\mathbf{R}_{75}$ | 75% Red | 0.75 | 0.00 | 0.00 | 0 | 1.00 | 0.75 |
| $\mathbf{R}_{50}$ | 50% Red | 0.50 | 0.00 | 0.00 | 0 | 1.00 | 0.50 |
| $\mathbf{R}_{25}$ | 25% Red | 0.25 | 0.00 | 0.00 | 0 | 1.00 | 0.25 |
| **P** | Pink | 1.00 | 0.50 | 0.50 | 0 | 0.5 | 1.00 |

**Fig. 12.12**
HSV color space. The illustration shows the HSV color space as a cylinder with the coordinates $H$ (*hue*) as the angle, $S$ (*saturation*) as the radius, and $V$ (*brightness value*) as the distance along the vertical axis, which runs between the black point **S** and the white point **W**. The table lists the $(R, G, B)$ and $(H, S, V)$ values of the color points marked on the graphic. Pure colors (composed of only one or two components) lie on the outer wall of the cylinder ($S = 1$), as exemplified by the gradually saturated reds ($\mathbf{R}_{25}$, $\mathbf{R}_{50}$, $\mathbf{R}_{75}$, **R**).



$H_{\text{HSV}}$ $S_{\text{HSV}}$ $V_{\text{HSV}}$

**Fig. 12.13**
HSV components for the test image in Fig. 12.2. The darker areas in the $h_{\text{HSV}}$ component correspond to the red and yellow colors, where the *hue* angle is near zero.

### *Java implementation*

In Java, the RGB→HSV conversion is implemented in the standard AWT `Color` class by the static method

`float[] RGBtoHSB (int r, int g, int b, float[] hsv)`

(HSV and HSB denote the same color space). The method takes three `int` arguments `r`, `g`, `b` (within the range $[0, 255]$) and returns a `float` array with the resulting $H, S, V$ values in the interval $[0, 1]$. When an existing `float` array is passed as the argument `hsv`, then the result is placed in it; otherwise (when `hsv` = `null`) a new array is created. Here is a simple example:

```
import java.awt.Color;
...
float[] hsv = new float[3];
int red = 128, green = 255, blue = 0;
hsv = Color.RGBtoHSB (red, green, blue, hsv);
float h = hsv[0];
float s = hsv[1];
float v = hsv[2];
...
```

A possible implementation of the Java method `RGBtoHSB()` using the definition in Eqns. (12.14)–(12.18) is given in Prog. 12.6.

### HSV→RGB conversion

To convert an HSV tuple $(H_{\text{HSV}}, S_{\text{HSV}}, V_{\text{HSV}})$, where $H_{\text{HSV}}$, $S_{\text{HSV}}$, and $V_{\text{HSV}} \in [0, 1]$, into the corresponding $(R, G, B)$ color values, the appropriate color sector

$$H' = (6 \cdot H_{\text{HSV}}) \bmod 6 \tag{12.20}$$

(with $0 \le H' < 6$) is determined first, followed by computing the intermediate values

$$\begin{aligned} c_1 &= \lfloor H' \rfloor, & x &= (1 - S_{\text{HSV}}) \cdot V_{\text{HSV}}, \\ c_2 &= H' - c_1, & y &= (1 - (S_{\text{HSV}} \cdot c_2)) \cdot V_{\text{HSV}}, \\ & & z &= (1 - (S_{\text{HSV}} \cdot (1 - c_2))) \cdot V_{\text{HSV}}. \end{aligned} \tag{12.21}$$

Depending on the value of $c_1$, the normalized RGB values $R', G', B' \in [0, 1]$ are then calculated from $v = V_{\text{HSV}}$, $x$, $y$, and $z$ as follows:[8]

$$(R', G', B') \leftarrow \begin{cases} (v, z, x) & \text{for } c_1 = 0, \\ (y, v, x) & \text{for } c_1 = 1, \\ (x, v, z) & \text{for } c_1 = 2, \\ (x, y, v) & \text{for } c_1 = 3, \\ (z, x, v) & \text{for } c_1 = 4, \\ (v, x, y) & \text{for } c_1 = 5. \end{cases} \tag{12.22}$$

Scaling the RGB components back to integer values in the range $[0, 255]$ is carried out as follows:

$$\begin{aligned} R &\leftarrow \min\bigl(\text{round}(K \cdot R'), 255\bigr), \\ G &\leftarrow \min\bigl(\text{round}(K \cdot G'), 255\bigr), \\ B &\leftarrow \min\bigl(\text{round}(K \cdot B'), 255\bigr). \end{aligned} \tag{12.23}$$

[8] The variables $x$, $y$, $z$ used here are not related to the CIEXYZ color space (see Ch. 14, Sec. 14.1).

```
1  float[] RGBtoHSV (int[] RGB) {
2    int R = RGB[0], G = RGB[1], B = RGB[2]; // R, G, B ∈ [0, 255]
3    int cHi = Math.max(R,Math.max(G,B));   // max. comp. value
4    int cLo = Math.min(R,Math.min(G,B));   // min. comp. value
5    int cRng = cHi - cLo;    // component range
6    float H = 0, S = 0, V = 0;
7    float cMax = 255.0f;
8
9    // compute value V
10   V = cHi / cMax;
11
12   // compute saturation S
13   if (cHi > 0)
14     S = (float) cRng / cHi;
15
16   // compute hue H
17   if (cRng > 0) { // hue is defined only for color pixels
18     float rr = (float)(cHi - R) / cRng;
19     float gg = (float)(cHi - G) / cRng;
20     float bb = (float)(cHi - B) / cRng;
21     float hh;
22     if (R == cHi)              // R is largest component value
23       hh = bb - gg;
24     else if (G == cHi)         // G is largest component value
25       hh = rr - bb + 2.0f;
26     else                       // B is largest component value
27       hh = gg - rr + 4.0f;
28     if (hh < 0)
29       hh = hh + 6;
30     H = hh / 6;
31   }
32   return new float[] {H, S, V};
33 }
```

**Prog. 12.6** RGB→HSV conversion (Java implementation). This Java method for RGB→HSV conversion follows the process given in the text to compute a single color tuple. It takes the same arguments and returns results identical to the standard `Color.RGBtoHSB()` method.

*Java implementation*

HSV→RGB conversion is implemented in Java's standard AWT `Color` class by the static method

```
int HSBtoRGB (float h, float s, float v),
```

which takes three `float` arguments $h, s, v \in [0, 1]$ and returns the corresponding RGB color as an `int` value with $3 \times 8$ bits arranged in the standard Java RGB format (see Fig. 12.6). One possible implementation of this method is shown in Prog. 12.7.

### RGB→HLS conversion

In the HLS model, the *hue* value $H_{\text{HLS}}$ is computed in the same way as in the HSV model (Eqns. (12.16)–(12.18)), that is,

$$H_{\text{HLS}} = H_{\text{HSV}}. \tag{12.24}$$

The other values, $L_{\text{HLS}}$ and $S_{\text{HLS}}$, are calculated as follows (for $C_{\text{high}}$, $C_{\text{low}}$, and $C_{\text{rng}}$, see Eqn. (12.15)):

**Prog. 12.7**
HSV→RGB conversion (Java implementation).

```
1  int HSVtoRGB (float[] HSV) {
2    float H = HSV[0], S = HSV[1], V = HSV[2]; // H, S, V ∈ [0, 1]
3    float r = 0, g = 0, b = 0;
4    float hh = (6 * H) % 6;    // h' ← (6 · h) mod 6
5    int   c1 = (int) hh;       // c1 ← ⌊h'⌋
6    float c2 = hh - c1;
7    float x = (1 - S) * V;
8    float y = (1 - (S * c2)) * V;
9    float z = (1 - (S * (1 - c2))) * V;
10   switch (c1) {
11     case 0: r = V; g = z; b = x; break;
12     case 1: r = y; g = V; b = x; break;
13     case 2: r = x; g = V; b = z; break;
14     case 3: r = x; g = y; b = V; break;
15     case 4: r = z; g = x; b = V; break;
16     case 5: r = V; g = x; b = y; break;
17   }
18   int R = Math.min((int)(r * 255), 255);
19   int G = Math.min((int)(g * 255), 255);
20   int B = Math.min((int)(b * 255), 255);
21   return new int[] {R, G, B};
22 }
```

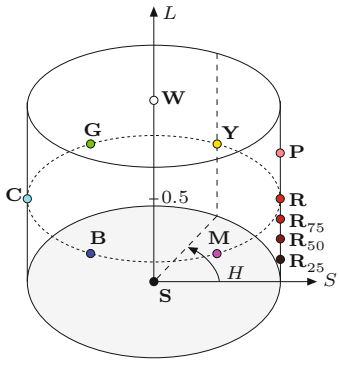
$$L_{\text{HLS}} = \frac{(C_{\text{high}} + C_{\text{low}})/255}{2}, \tag{12.25}$$

$$S_{\text{HLS}} = \begin{cases} 0 & \text{for } L_{\text{HLS}} = 0, \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{L_{\text{HLS}}} & \text{for } 0 < L_{\text{HLS}} \le 0.5, \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{1-L_{\text{HLS}}} & \text{for } 0.5 < L_{\text{HLS}} < 1, \\ 0 & \text{for } L_{\text{HLS}} = 1. \end{cases} \tag{12.26}$$

Using the aforementioned definitions, the RGB color cube is again mapped to a cylinder with height and radius 1 (see Fig. 12.15). In contrast to the HSV space (Fig. 12.12), the primary colors lie together in the horizontal plane at $L_{\text{HLS}} = 0.5$ and the white point lies outside of this plane at $L_{\text{HLS}} = 1.0$. Using these nonlinear transformations, the black and the white points are mapped to the top and the bottom planes of the cylinder, respectively. All points inside HLS cylinder correspond to valid colors in RGB space. Figure 12.14 shows the individual HLS components of the test image as grayscale images.

**Fig. 12.14**
HLS color components $H_{\text{HLS}}$ (*hue*), $S_{\text{HLS}}$ (*saturation*), and $L_{\text{HLS}}$ (*luminance*).

$H_{\text{HLS}}$ $L_{\text{HLS}}$ $S_{\text{HLS}}$




RGB/HLS values

| Pt. | Color | $R$ | $G$ | $B$ | $H$ | $S$ | $L$ |
|---|---|---|---|---|---|---|---|
| **S** | Black | 0.00 | 0.00 | 0.00 | — | 0.00 | 0.00 |
| **R** | Red | 1.00 | 0.00 | 0.00 | 0 | 1.00 | 0.50 |
| **Y** | Yellow | 1.00 | 1.00 | 0.00 | 1/6 | 1.00 | 0.50 |
| **G** | Green | 0.00 | 1.00 | 0.00 | 2/6 | 1.00 | 0.50 |
| **C** | Cyan | 0.00 | 1.00 | 1.00 | 3/6 | 1.00 | 0.50 |
| **B** | Blue | 0.00 | 0.00 | 1.00 | 4/6 | 1.00 | 0.50 |
| **M** | Magenta | 1.00 | 0.00 | 1.00 | 5/6 | 1.00 | 0.50 |
| **W** | White | 1.00 | 1.00 | 1.00 | — | 0.00 | 1.00 |
| $\mathbf{R}_{75}$ | 75% Red | 0.75 | 0.00 | 0.00 | 0 | 1.00 | 0.375 |
| $\mathbf{R}_{50}$ | 50% Red | 0.50 | 0.00 | 0.00 | 0 | 1.00 | 0.250 |
| $\mathbf{R}_{25}$ | 25% Red | 0.25 | 0.00 | 0.00 | 0 | 1.00 | 0.125 |
| **P** | Pink | 1.00 | 0.50 | 0.50 | 0/6 | 1.00 | 0.75 |

**Fig. 12.15**
HLS color space. The illustration shows the HLS color space visualized as a cylinder with the coordinates $H$ (*hue*) as the angle, $S$ (*saturation*) as the radius, and $L$ (*lightness*) as the distance along the vertical axis, which runs between the black point **S** and the white point **W**. The table lists the $(R, G, B)$ and $(H, S, L)$ values where "pure" colors (created using only one or two color components) lie on the lower half of the outer cylinder wall ($S = 1$), as illustrated by the gradually saturated reds ($\mathbf{R}_{25}$, $\mathbf{R}_{50}$, $\mathbf{R}_{75}$, **R**). Mixtures of all three primary colors, where at least one of the components is completely saturated, lie along the upper half of the outer cylinder wall; for example, the point **P** (pink).

### HLS→RGB conversion

When converting from HLS to the RGB space, we assume that $H_{\text{HLS}}$, $S_{\text{HLS}}$, $L_{\text{HLS}} \in [0, 1]$. In the case where $L_{\text{HLS}} = 0$ or $L_{\text{HLS}} = 1$, the result is

$$(R', G', B') = \begin{cases} (0,0,0) & \text{for } L_{\text{HLS}} = 0, \\ (1,1,1) & \text{for } L_{\text{HLS}} = 1. \end{cases} \tag{12.27}$$

Otherwise, we again determine the appropriate color sector

$$H' = (6 \cdot H_{\text{HLS}}) \bmod 6, \tag{12.28}$$

such that $0 \le H' < 6$, and from this

$$c_1 = \lfloor H' \rfloor, \qquad c_2 = H' - c_1, \tag{12.29}$$

$$d = \begin{cases} S_{\text{HLS}} \cdot L_{\text{HLS}} & \text{for } L_{\text{HLS}} \le 0.5, \\ S_{\text{HLS}} \cdot (1 - L_{\text{HLS}}) & \text{for } L_{\text{HLS}} > 0.5, \end{cases} \tag{12.30}$$

and the quantities

$$w = L_{\text{HLS}} + d, \qquad x = L_{\text{HLS}} - d, \tag{12.31}$$

$$y = w - (w - x) \cdot c_2, \qquad z = x + (w - x) \cdot c_2. \tag{12.32}$$

The final mapping to the RGB values is (similar to Eqn. (12.22))

$$(R', G', B') = \begin{cases} (w, z, x) & \text{for } c_1 = 0, \\ (y, w, x) & \text{for } c_1 = 1, \\ (x, w, z) & \text{for } c_1 = 2, \\ (x, y, w) & \text{for } c_1 = 3, \\ (z, x, w) & \text{for } c_1 = 4, \\ (w, x, y) & \text{for } c_1 = 5. \end{cases} \tag{12.33}$$

Finally, scaling the normalized $R'$, $G'$, $B'$ ($\in [0, 1]$) color components back to the $[0, 255]$ range is done as in Eqn. (12.23).

Prog. 12.8
RGB→HLS conversion
(Java implementation).

```
1  float[] RGBtoHLS (int[] RGB) {
2    int R = RGB[0], G = RGB[1], B = RGB[2]; // R,G,B in [0, 255]
3    float cHi = Math.max(R, Math.max(G, B));
4    float cLo = Math.min(R, Math.min(G, B));
5    float cRng = cHi - cLo;    // component range
6
7    // compute lightness L
8    float L = ((cHi + cLo) / 255f) / 2;
9
10   // compute saturation S
11   float S = 0;
12   if (0 < L && L < 1) {
13     float d = (L <= 0.5f) ? L : (1 - L);
14     S = 0.5f * (cRng / 255f) / d;
15   }
16
17   // compute hue H (same as in HSV)
18   float H = 0;
19   if (cHi > 0 && cRng > 0) {          // this is a color pixel!
20     float r = (float)(cHi - R) / cRng;
21     float g = (float)(cHi - G) / cRng;
22     float b = (float)(cHi - B) / cRng;
23     float h;
24     if (R == cHi)                    // R is largest component
25       h = b - g;
26     else if (G == cHi)               // G is largest component
27       h = r - b + 2.0f;
28     else                             // B is largest component
29       h = g - r + 4.0f;
30     if (h < 0)
31       h = h + 6;
32     H = h / 6;
33   }
34   return new float[] {H, L, S};
35 }
```

*Java implementation*

Currently there is no method in either the standard Java API or ImageJ for converting color values between RGB and HLS. Program 12.8 gives one possible implementation of the RGB→HLS conversion that follows the definitions in Eqns. (12.24)–(12.26). The HLS→RGB conversion is shown in Prog. 12.9.

## HSV and HLS compared

Despite the obvious similarity between the two color spaces, as Fig. 12.16 illustrates, substantial differences in the $V/L$ and $S$ components do exist. The essential difference between the HSV and HLS spaces is the ordering of the colors that lie between the white point **W** and the "pure" colors (**R**, **G**, **B**, **Y**, **C**, **M**), which consist of at most two primary colors, at least one of which is completely saturated.

The difference in how colors are distributed in RGB, HSV, and HLS space is readily apparent in Fig. 12.17. The starting point was a distribution of 1331 ($11 \times 11 \times 11$) color tuples obtained by uniformly

```
1  float[] HLStoRGB (float[] HLS) {
2    float H = HLS[0], L = HLS[1], S = HLS[2]; // H,L,S in [0, 1]
3    float r = 0, g = 0, b = 0;
4    if (L <= 0)        // black
5      r = g = b = 0;
6    else if (L >= 1)   // white
7      r = g = b = 1;
8    else {
9      float hh = (6 * H) % 6;        // = H'
10     int   c1 = (int) hh;
11     float c2 = hh - c1;
12     float d = (L <= 0.5f) ? (S * L) : (S * (1 - L));
13     float w = L + d;
14     float x = L - d;
15     float y = w - (w - x) * c2;
16     float z = x + (w - x) * c2;
17     switch (c1) {
18       case 0: r = w; g = z; b = x; break;
19       case 1: r = y; g = w; b = x; break;
20       case 2: r = x; g = w; b = z; break;
21       case 3: r = x; g = y; b = w; break;
22       case 4: r = z; g = x; b = w; break;
23       case 5: r = w; g = x; b = y; break;
24     }
25   } // r, g, b in [0, 1]
26   int R = Math.min(Math.round(r * 255), 255);
27   int G = Math.min(Math.round(g * 255), 255);
28   int B = Math.min(Math.round(b * 255), 255);
29   return new int[] {R, G, B};
30 }
```

**Prog. 12.9**
HLS→RGB conversion (Java implementation).

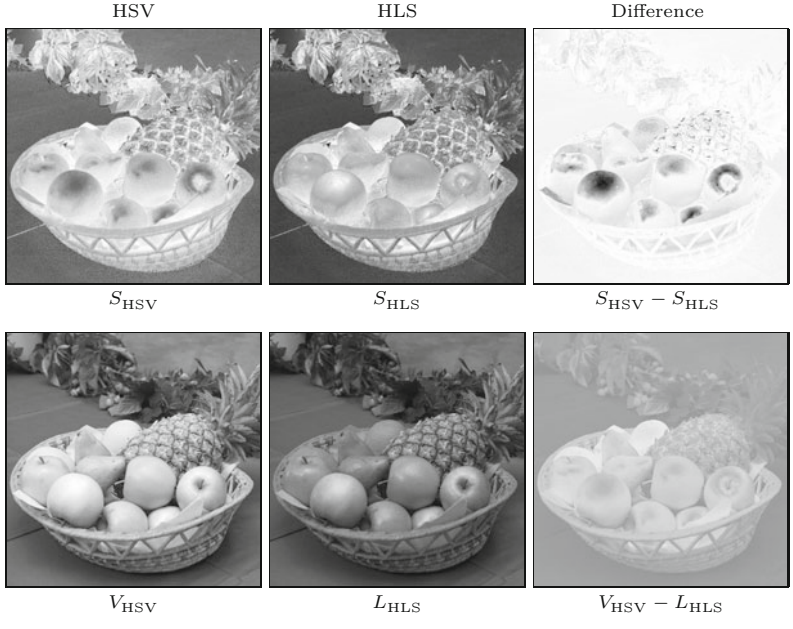



**Fig. 12.16**
HSV and HLS components compared. *Saturation* (top row) and *intensity* (bottom row). In the color *saturation* difference image $S_{\text{HSV}} - S_{\text{HLS}}$ (top), light areas correspond to positive values and dark areas to negative values. Saturation in the HLS representation, especially in the brightest sections of the image, is notably higher, resulting in negative values in the difference image. For the *intensity* (*value* and *luminance*, respectively) in general, $V_{\text{HSV}} \geq L_{\text{HLS}}$ and therefore the difference $V_{\text{HSV}} - L_{\text{HLS}}$ (bottom) is always positive. The *hue* component $H$ (not shown) is identical in both representations.

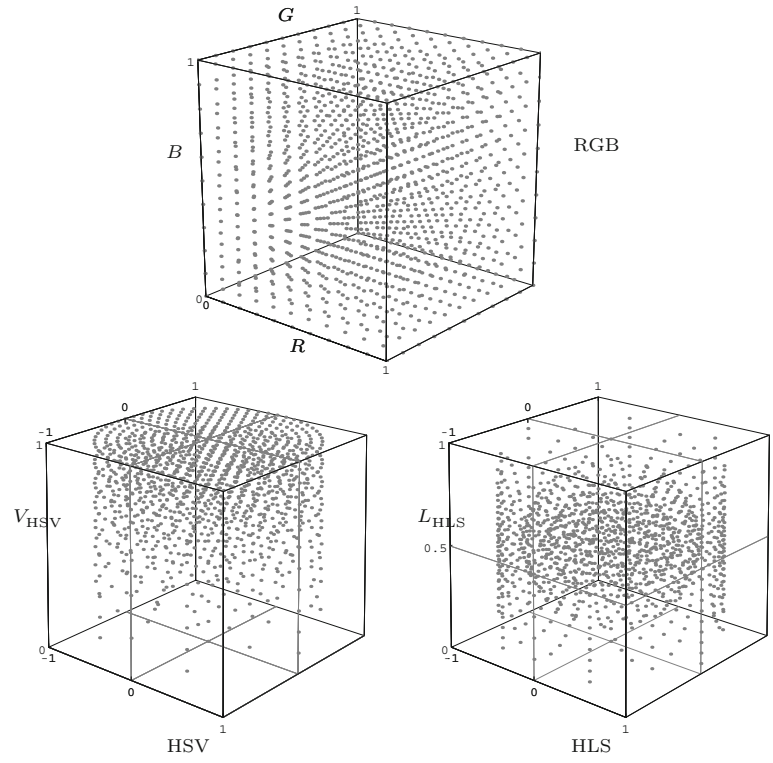



**Fig. 12.17**
Distribution of colors in the RGB, HSV, and HLS spaces. The starting point is the uniform distribution of colors in RGB space (top). The corresponding colors in the cylindrical spaces are distributed nonsymmetrically in HSV and symmetrically in HLS.

sampling the RGB space at an interval of 0.1 in each dimension. We can see clearly that in HSV space the maximally saturated colors ($s = 1$) form circular rings with increasing density toward the upper plane of the cylinder. In HLS space, however, the color samples are spread out symmetrically around the center plane and the density is significantly lower, particularly in the region near white. A given coordinate shift in this part of the color space leads to relatively small color changes, which allows the specification of very fine color grades in HLS space, especially for colors located in the upper half of the HLS cylinder.

Both the HSV and HLS color spaces are widely used in practice; for instance, for selecting colors in image editing and graphics design applications. In digital image processing, they are also used for *color keying* (i.e., isolating objects according to their *hue*) on a homogeneously colored background where the brightness is not necessarily constant.

### Desaturation in HSV/HLS color space

Desaturation of color images (cf. Sec. 12.2.2) represented in HSV or HLS color space is trivial since color saturation is available as a separate component. In particular, pixels with zero saturation are uncolored or gray. For example, HSV colors can be gradually or fully desaturated by simply multiplying the component $S$ by a fixed saturation factor $s \in [0, 1]$ and keeping $H, V$ unchanged, that is,

$$\begin{pmatrix} H_{\text{desat}} \\ S_{\text{desat}} \\ V_{\text{desat}} \end{pmatrix} = \begin{pmatrix} H \\ s \cdot S \\ V \end{pmatrix}, \tag{12.34}$$

which works analogously with HLS colors. While Eqn. (12.34) applies equally to all colors, it might be interesting to *selectively* modify only colors with certain hues. This is easily accomplished by replacing the fixed saturation factor $s$ by a hue-dependent function $f(H)$ (see also Exercise 12.6).

### 12.2.4 TV Component Color Spaces—YUV, YIQ, and $YC_bC_r$

These color spaces are an integral part of the standards surrounding the recording, storage, transmission, and display of television signals. YUV and YIQ are the fundamental color-encoding methods for the analog NTSC and PAL systems, and $YC_bC_r$ is a part of the international standards governing digital television [114]. All of these color spaces have in common the idea of separating the luminance component $Y$ from two chroma components and, instead of directly encoding colors, encoding color differences. In this way, compatibility with legacy black and white systems is maintained while at the same time the bandwidth of the signal can be optimized by using different transmission bandwidths for the brightness and the color components. Since the human visual system is not able to perceive detail in the color components as well as it does in the intensity part of a video signal, the amount of information, and consequently bandwidth, used in the color channel can be reduced to approximately 1/4 of that used for the intensity component. This fact is also used when compressing digital still images and is why, for example, the JPEG codec converts RGB images to $YC_bC_r$. That is why these color spaces are important in digital image processing, even though raw YIQ or YUV images are rarely encountered in practice.

#### YUV

YUV is the basis for the color encoding used in analog television in both the North American NTSC and the European PAL systems. The luminance component $Y$ is computed, just as in Eqn. (12.9), from the RGB components as

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{12.35}$$

under the assumption that the RGB values have already been gamma corrected according to the TV encoding standard ($\gamma_{\text{NTSC}} = 2.2$ and $\gamma_{\text{PAL}} = 2.8$, see Ch. 4, Sec. 4.7) for playback. The UV components are computed from a weighted difference between the luminance and the blue or red components as

$$U = 0.492 \cdot (B - Y) \qquad \text{und} \qquad V = 0.877 \cdot (R - Y), \tag{12.36}$$

and the entire transformation from RGB to YUV is

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \tag{12.37}$$

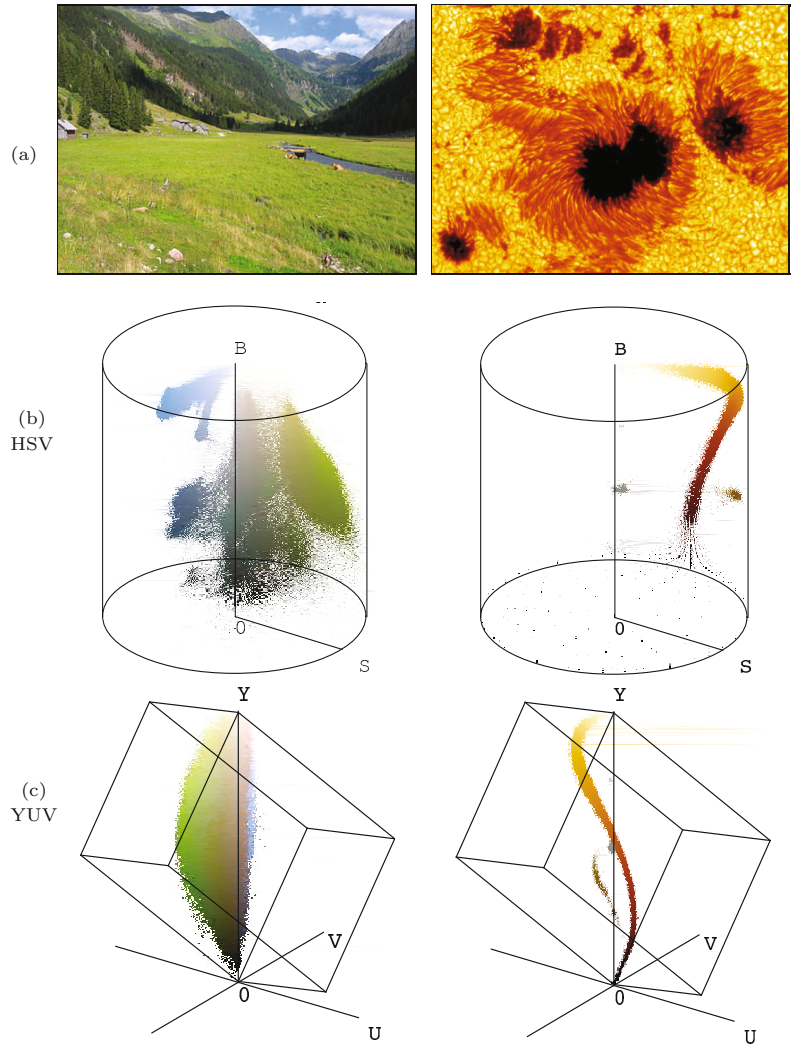



**Fig. 12.18**
Examples of the color distribution of natural images in different color spaces. Original images (a); color distribution in HSV- (b), and YUV-space (c). See Fig. 12.9 for the corresponding distributions in RGB color space.

The transformation from YUV back to RGB is found by inverting the matrix in Eqn. (12.37):

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.140 \\ 1.000 & -0.395 & -0.581 \\ 1.000 & 2.032 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ U \\ V \end{pmatrix}. \tag{12.38}$$

The color distributions in YUV-space for a set of natural images are shown in Fig. 12.18.

## YIQ

The original NTSC system used a variant of YUV called YIQ (I for "in-phase", Q for "quadrature"), where both the $U$ and $V$ color vectors were rotated and mirrored such that

$$\begin{pmatrix} I \\ Q \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} U \\ V \end{pmatrix}, \tag{12.39}$$

where $\beta = 0.576$ (33°). The $Y$ component is the same as in YUV. Although the YIQ has certain advantages with respect to bandwidth requirements it has been completely replaced by YUV [124, p. 240].

### $YC_bC_r$

The $YC_bC_r$ color space is an internationally standardized variant of YUV that is used for both digital television and image compression (e.g., in JPEG). The chroma components $C_b, C_r$ are (similar to $U, V$) difference values between the luminance and the blue and red components, respectively. In contrast to YUV, the weights of the RGB components for the luminance $Y$ depend explicitly on the coefficients used for the chroma values $C_b$ and $C_r$ [197, p. 16]. For arbitrary weights $w_B, w_R$, the transformation is defined as

$$Y = w_R \cdot R + (1 - w_B - w_R) \cdot G + w_B \cdot B, \tag{12.40}$$

$$C_b = \frac{0.5}{1 - w_B} \cdot (B - Y), \tag{12.41}$$

$$C_r = \frac{0.5}{1 - w_R} \cdot (R - Y), \tag{12.42}$$

with $w_R = 0.299$ and $w_B = 0.114$ ($w_G = 0.587$)[9] according to ITU[10] recommendation BT.601 [123]. Analogously, the reverse mapping from $YC_bC_r$ to RGB is

$$R = Y + \frac{(1 - w_R) \cdot C_r}{0.5}, \tag{12.43}$$

$$G = Y - \frac{w_B \cdot (1 - w_B) \cdot C_b + w_R \cdot (1 - w_R) \cdot C_r}{0.5 \cdot (1 - w_B - w_R)}, \tag{12.44}$$

$$B = Y + \frac{(1 - w_B) \cdot C_b}{0.5}. \tag{12.45}$$

In matrix-vector notation this gives the linear transformation

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \tag{12.46}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.403 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.773 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix}. \tag{12.47}$$

Different weights are recommended based on how the color space is used; for example, ITU-BT.709 [122] recommends $w_R = 0.2125$ and $w_B = 0.0721$ to be used in digital HDTV production. The values of $U, V$, $I, Q$, and $C_b, C_r$ may be both positive or negative. To encode $C_b, C_r$ values to digital numbers, a suitable offset is typically added to obtain positive-only values, for example, $128 = 2^7$ in case of 8-bit components.

Figure 12.19 shows the three color spaces YUV, YIQ, and $YC_bC_r$ together for comparison. The $U, V$, $I, Q$, and $C_b, C_r$ values in the

[9] $w_R + w_G + w_B = 1$.
[10] International Telecommunication Union (www.itu.int).

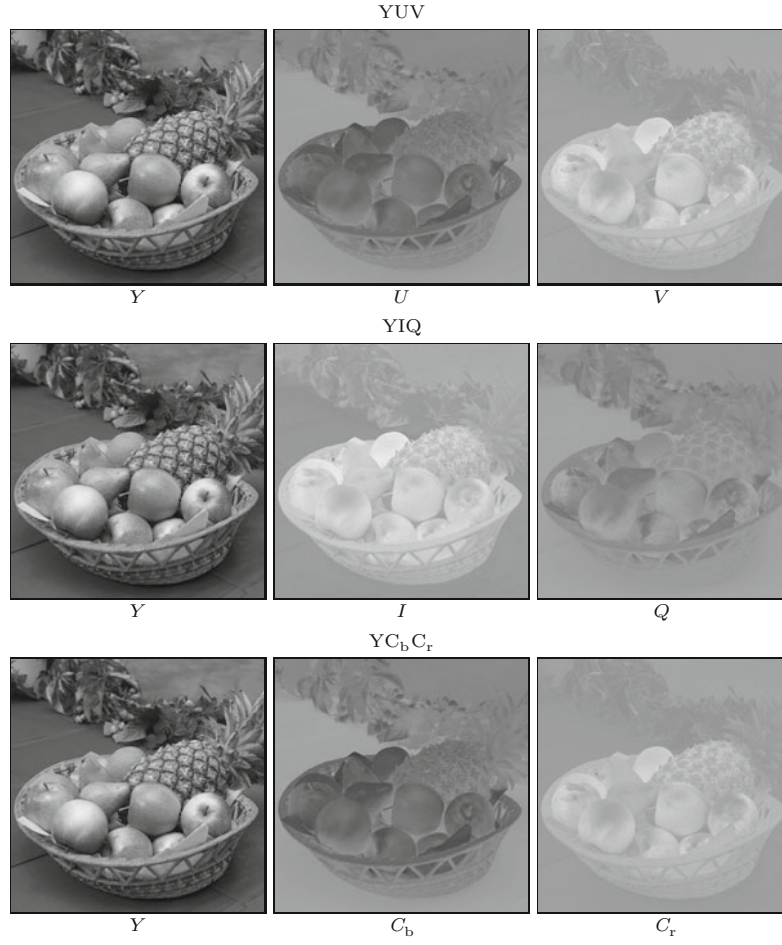



**Fig. 12.19**
Comparing YUV-, YIQ-, and $YC_bC_r$ values. The $Y$ values are identical in all three color spaces.

right two frames have been offset by 128 so that the negative values are visible. Thus a value of zero is represented as medium gray in these images. The $YC_bC_r$ encoding is practically indistinguishable from YUV in these images since they both use very similar weights for the color components.

### 12.2.5 Color Spaces for Printing—CMY and CMYK

In contrast to the *additive* RGB color scheme (and its various color models), color printing makes use of a *subtractive* color scheme, where each printed color reduces the intensity of the reflected light at that location. Color printing requires a minimum of three primary colors; traditionally *cyan* ($C$), *magenta* ($M$), and *yellow* ($Y$)[11] have been used.

Using subtractive color mixing on a white background, $C = M = Y = 0$ (no ink) results in the color *white* and $C = M = Y = 1$ (complete saturation of all three inks) in the color *black*. A cyan-colored ink will absorb *red* ($R$) most strongly, magenta absorbs *green*

[11] Note that in this case $Y$ stands for *yellow* and is unrelated to the $Y$ luma or luminance component in YUV or $YC_bC_r$.

($G$), and yellow absorbs *blue* ($B$). The simplest form of the CMY model is defined as

$$C = 1 - R, \qquad M = 1 - G, \qquad Y = 1 - B. \tag{12.48}$$

In practice, the color produced by fully saturating all three inks is not physically a true black. Therefore, the three primary colors $C, M, Y$ are usually supplemented with a black ink ($K$) to increase the color range and coverage (gamut). In the simplest case, the amount of black is

$$K = \min(C, M, Y)\,. \tag{12.49}$$

With rising levels of black, however, the intensity of the $C, M, Y$ components can be gradually reduced. Many methods for reducing the primary dyes have been proposed and we look at three of them in the following.

### CMY→CMYK conversion (version 1)

In this simple variant the $C, M, Y$ values are reduced linearly with increasing $K$ (Eqn. (12.49)), which yields the modified components as

$$\begin{pmatrix} C_1 \\ M_1 \\ Y_1 \\ K_1 \end{pmatrix} = \begin{pmatrix} C - K \\ M - K \\ Y - K \\ K \end{pmatrix}. \tag{12.50}$$

### CMY→CMYK conversion (version 2)

The second variant corrects the color by reducing the $C, M, Y$ components by $s = \frac{1}{1-K}$, resulting in stronger colors in the dark areas of the image:

$$\begin{pmatrix} C_2 \\ M_2 \\ Y_2 \\ K_2 \end{pmatrix} = \begin{pmatrix} (C-K)\cdot s \\ (M-K)\cdot s \\ (Y-K)\cdot s \\ K \end{pmatrix}, \quad \text{with} \quad s = \begin{cases} \frac{1}{1-K} & \text{for } K < 1, \\ 1 & \text{otherwise.} \end{cases} \tag{12.51}$$

In both versions, the $K$ component (as defined in Eqn. (12.49)) is used directly without modification, and all gray tones (that is, when $R = G = B$) are printed using black ink $K$, without any contribution from $C$, $M$, or $Y$.

While both of these simple definitions are widely used, neither one produces high quality results. Figure 12.20(a) compares the result from version 2 with that produced with Adobe Photoshop (Fig. 12.20(c)). The difference in the cyan component $C$ is particularly noticeable and also the exceeding amount of black ($K$) in the brighter areas of the image.

In practice, the required amounts of black $K$ and $C, M, Y$ depend so strongly on the printing process and the type of paper used that print jobs are routinely calibrated individually.

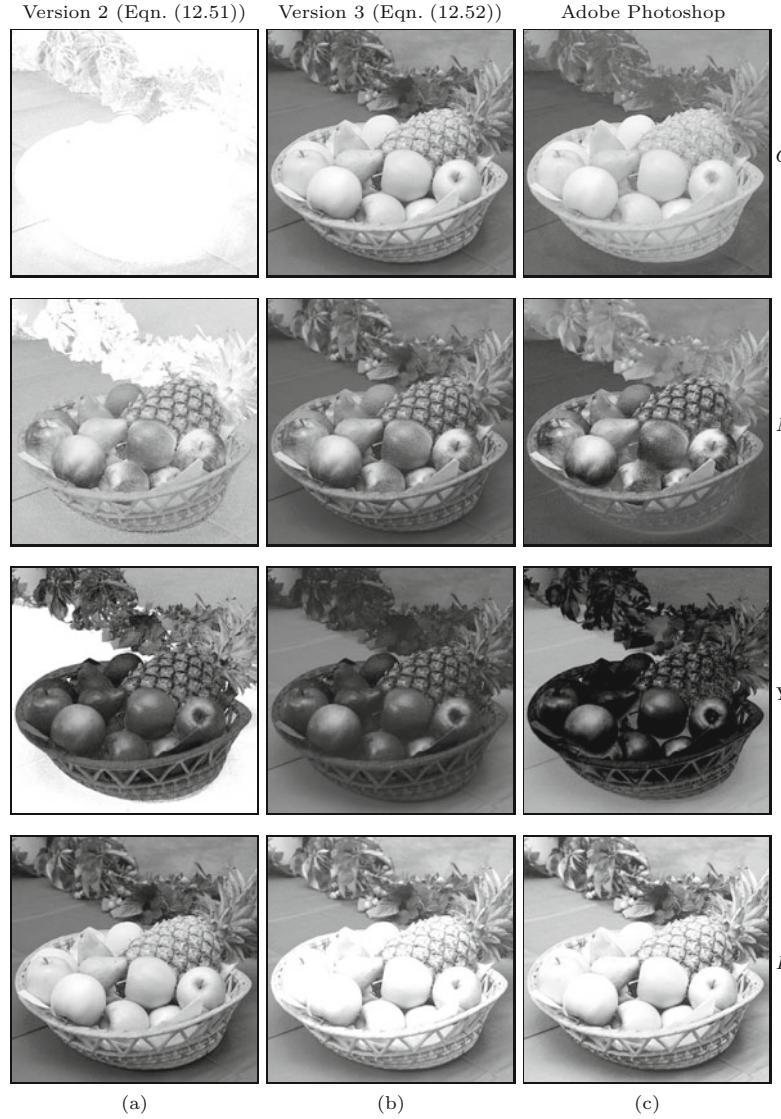



**Fig. 12.20**
RGB→CMYK conversion comparison. Simple conversion using Eqn. (12.51) (a), applying the *undercolor-removal* and *black-generation* functions of Eqn. (12.52) (b), and results obtained with Adobe Photoshop (c). The color intensities are shown inverted, that is, darker areas represent higher CMYK color values. The simple conversion (a), in comparison with Photoshop's result (c), shows strong deviations in all color components, $C$ and $K$ in particular. The results in (b) are close to Photoshop's and could be further improved by tuning the corresponding function parameters.

### CMY→CMYK conversion (version 3)

In print production, special transfer functions are applied to tune the results. For example, the Adobe PostScript interpreter [135, p. 345] specifies an *undercolor-removal function* $f_{\text{UCR}}(K)$ for gradually reducing the CMY components and a separate *black-generation function* $f_{\text{BG}}(K)$ for controlling the amount of black. These functions are used in the form

$$\begin{pmatrix} C_3 \\ M_3 \\ Y_3 \\ K_3 \end{pmatrix} = \begin{pmatrix} C - f_{\text{UCR}}(K) \\ M - f_{\text{UCR}}(K) \\ Y - f_{\text{UCR}}(K) \\ f_{\text{BG}}(K) \end{pmatrix}, \tag{12.52}$$

where $K = \min(C, M, Y)$, as defined in Eqn. (12.49). The functions $f_{\text{UCR}}$ and $f_{\text{BG}}$ are usually nonlinear, and the resulting values

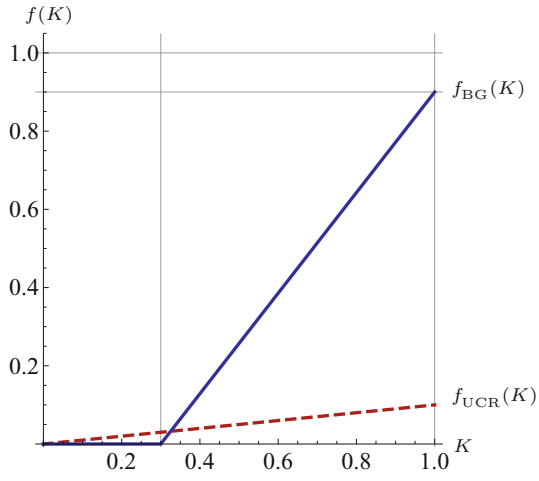



**Fig. 12.21**
Examples of *undercolor-removal function* $f_{\text{UCR}}$ (Eqn. (12.53)) and *black generation function* $f_{\text{BG}}$ (Eqn. (12.54)). The parameter settings are $s_K = 0.1$, $K_0 = 0.3$, and $K_{\max} = 0.9$.

$C_3, M_3, Y_3, K_3$ are scaled (typically by means of *clamping*) to the interval $[0, 1]$. The example shown in Fig. 12.20(b) was produced to approximate the results of Adobe Photoshop using the definitions

$$f_{\text{UCR}}(K) = s_K \cdot K, \tag{12.53}$$

$$f_{\text{BG}}(K) = \begin{cases} 0 & \text{for } K < K_0, \\ K_{\max} \cdot \frac{K - K_0}{1 - K_0} & \text{for } K \geq K_0, \end{cases} \tag{12.54}$$

where $s_K = 0.1$, $K_0 = 0.3$, and $K_{\max} = 0.9$ (see Fig. 12.21). With this definition, $f_{\text{UCR}}$ reduces the CMY components by 10% of the $K$ value (by Eqn. (12.52)), which mostly affects the dark areas of the image with high $K$ values. The effect of the function $f_{\text{BG}}$ (Eqn. (12.54)) is that for values of $K < K_0$ (i.e., in the light areas of the image) no black ink is added at all. In the interval $K = K_0, \ldots, 1.0$, the black component is increased linearly up to the maximum value $K_{\max}$. The result in Fig. 12.20(b) is relatively close to the CMYK component values produced by Photoshop[12] in Fig. 12.20(c). It could be further improved by adjusting the function parameters $s_K$, $K_0$, and $K_{\max}$ (Eqn. (12.52)).

Even though the results of this last variant (3) for converting RGB to CMYK are better, it is only a gross approximation and still too imprecise for professional work. As we discuss in Chapter 14, technically correct color conversions need to be based on precise, "colorimetric" grounds.

## 12.3 Statistics of Color Images

### 12.3.1 How Many Different Colors are in an Image?

A minor but frequent task in the context of color images is to determine how many different colors are contained in a given image.

[12] Actually Adobe Photoshop does not convert directly from RGB to CMYK. Instead, it first converts to, and then from, the CIELAB color space (see Ch. 14, Sec. 14.1).

One way of doing this would be to create and fill a histogram array with one integer element for each color and subsequently count all histogram cells with values greater than zero. But since a 24-bit RGB color image potentially contains $2^{24} = 16,777,216$ colors, the resulting histogram array (with a size of 64 megabytes) would be larger than the image itself in most cases!

A simple solution to this problem is to *sort* the pixel values in the (1D) pixel array such that all identical colors are placed next to each other. The sorting order is of course completely irrelevant, and the number of contiguous color blocks in the sorted pixel vector corresponds to the number of different colors in the image. This number can be obtained by simply counting the transitions between neighboring color blocks, as shown in Prog. 12.10. Of course, we do not want to sort the original pixel array (which would destroy the image) but a copy of it, which can be obtained with Java's `clone()` method.[13] Sorting of the 1D array in Prog. 12.10 is accomplished (in line 4) with the generic Java method `Arrays.sort()`, which is implemented very efficiently.

**Prog. 12.10** Counting the colors contained in an RGB image. The method `countColors()` first creates a copy of the 1D RGB (`int`) pixel array (line 3), then sorts that array, and finally counts the transitions between contiguous blocks of identical colors.

```
1  int countColors (ColorProcessor cp) {
2    // duplicate the pixel array and sort it
3    int[] pixels = ((int[]) cp.getPixels()).clone();
4    Arrays.sort(pixels);  // requires java.util.Arrays
5
6    int k = 1;  // color count (image contains at least 1 color)
7    for (int i = 0; i < pixels.length-1; i++) {
8      if (pixels[i] != pixels[i + 1])
9        k = k + 1;
10   }
11   return k;
12 }
```

### 12.3.2 Color Histograms

We briefly touched on histograms of color images in Chapter 3, Sec. 3.5, where we only considered the 1D distributions of the image intensity and the individual color channels. For instance, the built-in ImageJ method `getHistogram()`, when applied to an object of type `ColorProcessor`, simply computes the intensity histogram of the corresponding gray values:

```
ColorProcessor cp;
int[] H = cp.getHistogram();
```

As an alternative, one could compute the individual intensity histograms of the three color channels, although (as discussed in Chapter 3, Sec. 3.5.2) these do not provide any information about the actual colors in this image. Similarly, of course, one could compute the distributions of the individual components of any other color space, such as HSV or CIELAB.

[13] Java arrays implement the `Cloneable` interface.

A *full* histogram of an RGB image is 3D and, as noted earlier, consists of $256 \times 256 \times 256 = 2^{24}$ cells of type `int` (for 8-bit color components). Such a histogram is not only very large[14] but also difficult to visualize.

### 2D color histograms

A useful alternative to the full 3D RGB histogram are 2D histogram projections (Fig. 12.22). Depending on the axis of projection, we obtain 2D histograms with coordinates red-green ($\mathsf{h}_{\mathrm{RG}}$), red-blue ($\mathsf{h}_{\mathrm{RB}}$), or green-blue ($\mathsf{h}_{\mathrm{GB}}$), respectively, with the values

$$\begin{aligned} \mathsf{h}_{\mathrm{RG}}(r,g) &:= \text{number of pixels with } \boldsymbol{I}(u,v) = (r,g,*), \\ \mathsf{h}_{\mathrm{RB}}(r,b) &:= \text{number of pixels with } \boldsymbol{I}(u,v) = (r,*,b), \\ \mathsf{h}_{\mathrm{GB}}(g,b) &:= \text{number of pixels with } \boldsymbol{I}(u,v) = (*,g,b), \end{aligned} \tag{12.55}$$

where $*$ denotes an arbitrary component value. The result is, independent of the original image size, a set of 2D histograms of size $256 \times 256$ (for 8-bit RGB components), which can easily be visualized as images. Note that it is not necessary to obtain the full RGB histogram in order to compute the combined 2D histograms (see Prog. 12.11).

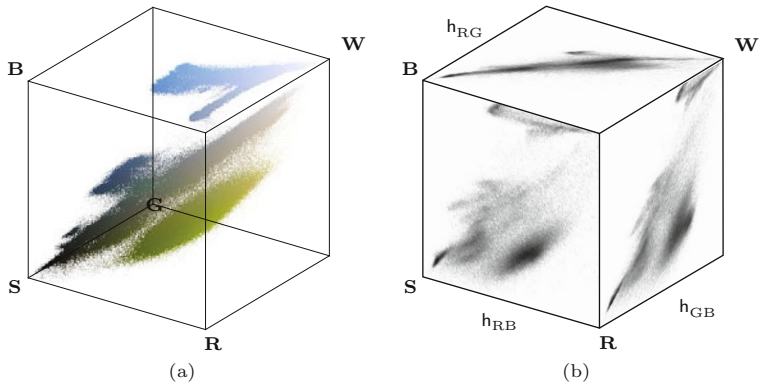



**Fig. 12.22**
2D RGB histogram projections. 3D RGB cube illustrating an image's color distribution (a). The color points indicate the corresponding pixel colors and not the color frequency. The combined histograms for red-green ($\mathsf{h}_{\mathrm{RG}}$), red-blue ($\mathsf{h}_{\mathrm{RB}}$), and green-blue ($\mathsf{h}_{\mathrm{GB}}$) are 2D projections of the 3D histogram. The corresponding image is shown in Fig. 12.9(a).

As the examples in Fig. 12.23 show, the combined color histograms do, to a certain extent, express the color characteristics of an image. They are therefore useful, for example, to identify the coarse type of the depicted scene or to estimate the similarity between images (see also Exercise 12.8).

## 12.4 Exercises

**Exercise 12.1.** Create an ImageJ plugin that rotates the individual components of an RGB color image; that is, $R \to G \to B \to R$.

**Exercise 12.2.** Pseudocolors are sometimes used for displaying grayscale images (i.e., for viewing medical images with high dynamic

[14] It may seem a paradox that, although the RGB histogram is usually much larger than the image itself, the histogram is not sufficient in general to reconstruct the original image.

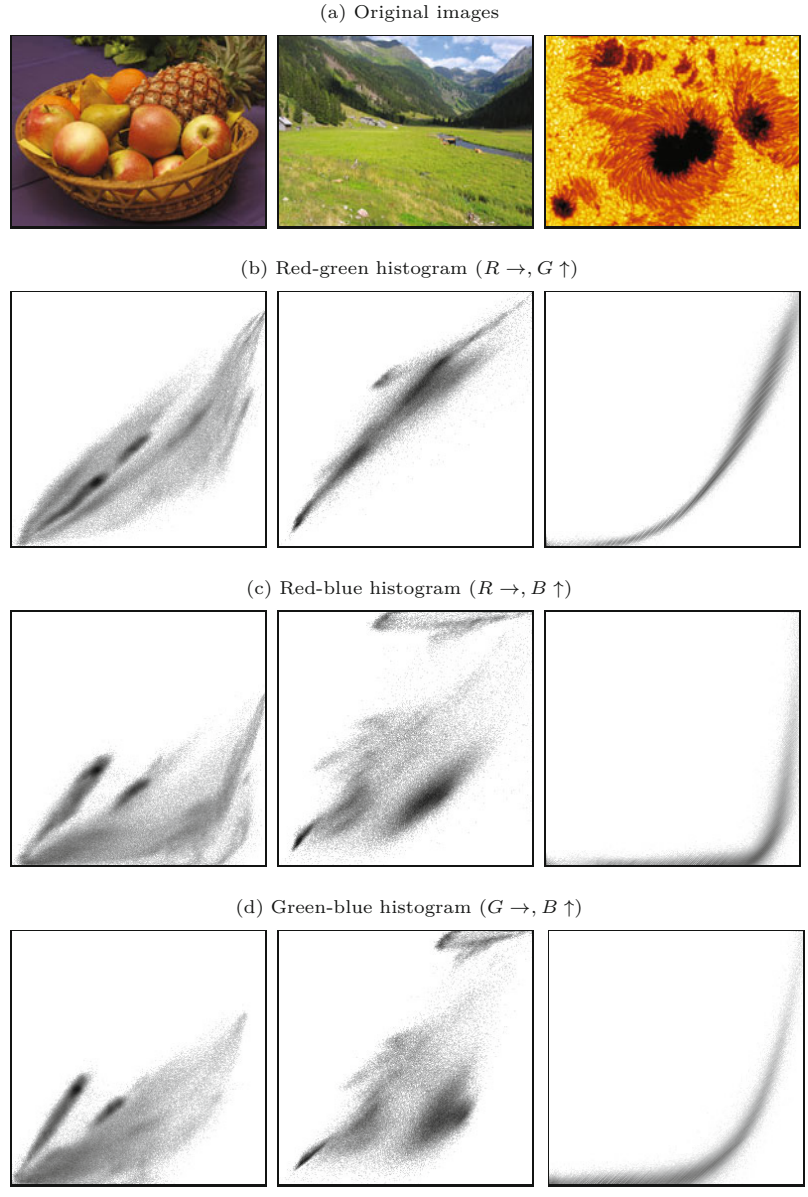



**Fig. 12.23** Combined color histogram examples. For better viewing, the images are inverted (dark regions indicate high frequencies) and the gray value corresponds to the logarithm of the histogram entries (scaled to the maximum entries).

range). Create an ImageJ plugin for converting 8-bit grayscale images to an indexed image with 256 colors, simulating the hues of glowing iron (from dark red to yellow and white).

**Exercise 12.3.** Create an ImageJ plugin that shows the color table of an 8-bit indexed image as a new image with $16 \times 16$ rectangular color fields. Mark all unused color table entries in a suitable way. Look at Prog. 12.3 as a starting point.

**Exercise 12.4.** Show that a "desaturated" RGB pixel produced in the form $(r, g, b) \rightarrow (y, y, y)$, where $y$ is the equivalent luminance value (see Eqn. (12.11)), has the luminance $y$ as well.

```
1  int[][] get2dHistogram
2      (ColorProcessor cp, int c1, int c2) {
3    // c1, c2: component index R = 0, G = 1, B = 2
4
5    int[] RGB = new int[3];
6    int[][] h = new int[256][256];   // 2D histogram h[c1][c2]
7
8    for (int v = 0; v < cp.getHeight(); v++) {
9      for (int u = 0; u < cp.getWidth(); u++) {
10       cp.getPixel(u, v, RGB);
11       int i1 = RGB[c1];
12       int i2 = RGB[c2];
13       // increment the associated histogram cell
14       h[i1][i2]++;
15     }
16   }
17   return h;
18 }
```

**Prog. 12.11**
Java method `get2dHistogram()` for computing a combined 2D color histogram. The color components (histogram axes) are specified by the parameters `c1` and `c2`. The color distribution `H` is returned as a 2D `int` array. The method is defined in class `ColorStatistics` (Prog. 12.10).

**Exercise 12.5.** Extend the ImageJ plugin for desaturating color images in Prog. 12.5 such that the image is only modified inside the user-selected region of interest (ROI).

**Exercise 12.6.** Write an ImageJ plugin that *selectively desaturates* an RGB image, preserving colors with a hue close to a given *reference color* $\mathbf{c}_{\text{ref}} = (R_{\text{ref}}, G_{\text{ref}}, B_{\text{ref}})$, with (HSV) hue $H_{\text{ref}}$ (see the example in Fig. 12.24). Transform the image to HSV and modify the colors (cf. Eqn. (12.34)) in the form

$$\begin{pmatrix} H_{\text{desat}} \\ S_{\text{desat}} \\ V_{\text{desat}} \end{pmatrix} = \begin{pmatrix} H \\ f(H) \cdot S \\ V \end{pmatrix}, \tag{12.56}$$

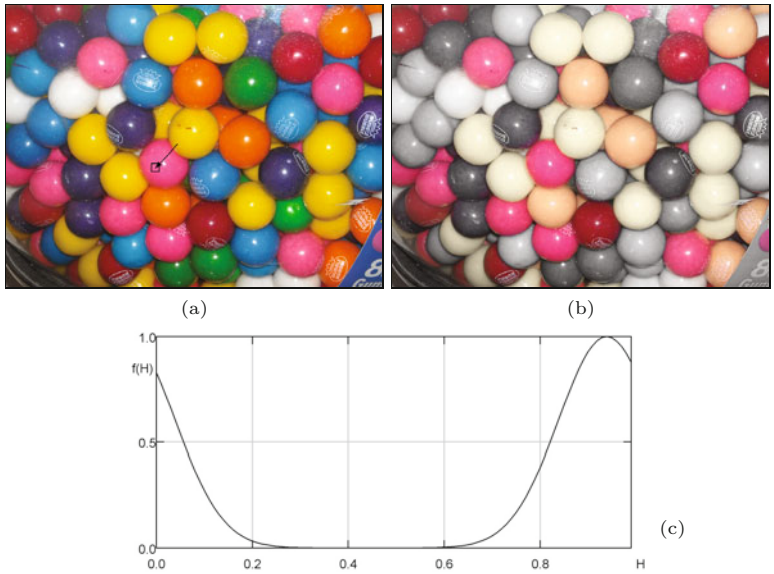



**Fig. 12.24**
Selective desaturation example. Original image with selected reference color $\mathbf{c}_{\text{ref}} = (250, 92, 150)$ (a), desaturated image (b). Gaussian saturation function $f(H)$ (see Eqn. (12.58)) with reference hue $H_{\text{ref}} = 0.9388$ and $\sigma = 0.1$ (c).

where $f(H)$ is a smooth saturation function, for example, a Gaussian function of the form

$$f(H) = e^{-\frac{(H-H_{\text{ref}})^2}{2\cdot\sigma^2}} = g_\sigma(H - H_{\text{ref}}), \tag{12.57}$$

with center $H_{\text{ref}}$ and variance $\sigma^2$ (see Fig. 12.24(c)). Recall that the $H$ component is circular in $[0, 1)$. To obtain a continuous and periodic saturation function we note that $H' = H - H_{\text{ref}}$ is in the range $[-1, 1]$ and reformulate $f(H)$ as

$$f(H) = \begin{cases} g_\sigma(H') & \text{for } -0.5 \le H' \le 0.5, \\ g_\sigma(H'+1) & \text{for } H' < -0.5, \\ g_\sigma(H'-1) & \text{for } H' > 0.5. \end{cases} \tag{12.58}$$

Verify the values of the function $f(H)$, check in particular that it is 1 for the reference color! What would be a good (synthetic) color image for validating the saturation function? Use ImageJ's color picker (pipette) tool to specify the reference color $\boldsymbol{c}_{\text{ref}}$ interactively.[15]

**Exercise 12.7.** Calculate (analogous to Eqns. (12.46)–(12.47)) the complete transformation matrices for converting from (linear) RGB colors to $\text{YC}_\text{b}\text{C}_\text{r}$ for the ITU-BT.709 (HDTV) standard with the coefficients $w_\text{R} = 0.2126$, $w_\text{B} = 0.0722$ and $w_\text{G} = 0.7152$.

**Exercise 12.8.** Determining the similarity between images of different sizes is a frequent problem (e.g., in the context of image data bases). Color statistics are commonly used for this purpose because they facilitate a coarse classification of images, such as landscape images, portraits, etc. However, 2D color histograms (as described in Sec. 12.3.2) are usually too large and thus cumbersome to use for this purpose. A simple idea could be to split the 2D histograms or even the full RGB histogram into $K$ regions (*bins*) and to combine the corresponding entries into a $K$-dimensional feature vector, which could be used for a coarse comparison. Develop a concept for such a procedure, and also discuss the possible problems.

**Exercise 12.9.** Write a program (plugin) that generates a sequence of colors with constant hue and saturation but different brightness (value) in HSV space. Transform these colors to RGB and draw them into a new image. Verify (visually) if the hue really remains constant.

**Exercise 12.10.** When applying any type of filter in HSV or HLS color space one must keep in mind that the *hue* component $H$ is circular in $[0, 1)$ and thus shows a discontinuity at the $1 \to 0$ ($360 \to 0°$) transition. For example, a linear filter would not take into account that $H = 0.0$ and $H = 1.0$ refer to the same hue (red) and thus cannot be applied directly to the $H$ component. One solution is to filter the *cosine* and *sine* values of the $H$ component (which really is an angle) instead, and composing the filtered hue array from the filtered cos / sin values (see Ch. 15, Sec. 15.1.3 for details). Based on this idea, implement a variable-sized linear Gaussian filter (see Ch. 5, Sec. 5.2.7) for the HSV color space.

[15] The current color pick is returned by the ImageJ method `Toolbar.getForegroundColor()`.