13

Color Quantization

The task of color quantization is to select and assign a limited set of colors for representing a given color image with maximum fidelity. Assume, for example, that a graphic artist has created an illustration with beautiful shades of color, for which he applied 150 different crayons. His editor likes the result but, for some technical reason, instructs the artist to draw the picture again, this time using only 10 different crayons. The artist now faces the problem of color quantization—his task is to select a "palette" of the 10 best suited from his 150 crayons and then choose the most similar color to redraw each stroke of his original picture.

In the general case, the original image I contains a set of m different colors $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_m\}$, where m could be only a few or several thousand, but at most 2^{24} for a 3×8-bit color image. The goal is to replace the original colors by a (usually much smaller) set of colors $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \ldots, \mathbf{C}'_n\}$, with $n < m$. The difficulty lies in the proper choice of the reduced color palette $\mathcal{C}'$ such that damage to the resulting image is minimized.

In practice, this problem is encountered, for example, when converting from full-color images to images with lower pixel depth or to index ("palette") images, such as the conversion from 24-bit TIFF to 8-bit GIF images with only 256 (or fewer) colors. Until a few years ago, a similar problem had to be solved for displaying full-color images on computer screens because the available display memory was often limited to only 8 bits. Today, even the cheapest display hardware has at least 24-bit depth and therefore this particular need for (fast) color quantization no longer exists.

13.1 Scalar Color Quantization

Scalar (or *uniform*) quantization is a simple and fast process that is independent of the image content. Each of the original color components c_i (e.g., R_i, G_i, B_i) in the range $[0, \ldots, m-1]$ is independently converted to the new range $[0, \ldots, n-1]$, in the simplest case by a

© Springer-Verlag London 2016
W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_13

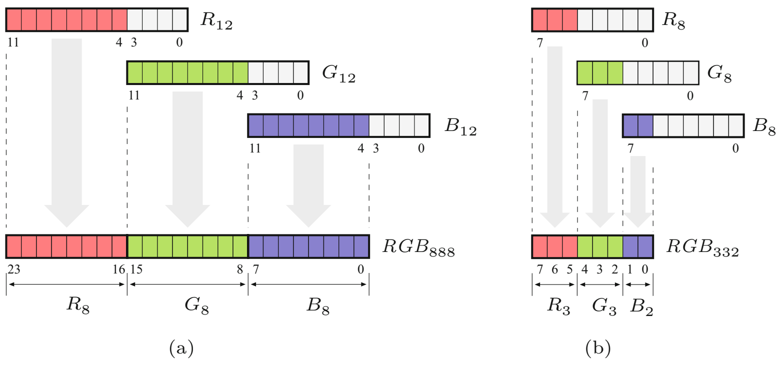


Fig. 13.1
Scalar quantization of color components by truncating lower bits. Quantization of 3 × 12-bit to 3 × 8-bit colors (a). Quantization of 3 × 8-bit to 3:3:2-packed 8-bit colors (b). The Java code segment in Prog. 13.1 shows the corresponding sequence of bit operations.

linear quantization in the form

$$c_i' \leftarrow \left\lfloor c_i \cdot \frac{n}{m} \right\rfloor \tag{13.1}$$

for all color components c_i. A typical example would be the conversion of a color image with 3 × 12-bit components ($m = 4096$) to an RGB image with 3 × 8-bit components ($n = 256$). In this case, each original component value is multiplied by $n/m = 256/4096 = 1/16 = 2^{-4}$ and subsequently truncated, which is equivalent to an integer division by 16 or simply ignoring the lower 4 bits of the corresponding binary values (see Fig. 13.1(a)). m and n are usually the same for all color components but not always.

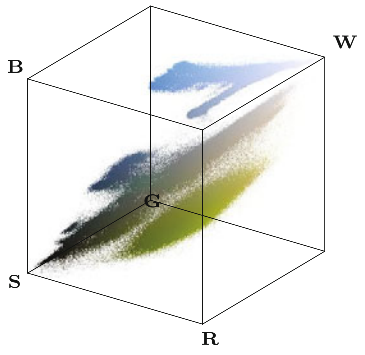
An extreme (today rarely used) approach is to quantize 3 × 8 color vectors to single-byte (8-bit) colors, where 3 bits are used for red and green and only 2 bits for blue, as shown in Prog. 13.1(b). In this case, $m = 256$ for all color components, $n_{\text{red}} = n_{\text{green}} = 8$, and $m_{\text{blue}} = 4$.

Prog. 13.1
Quantization of a 3 × 8-bit RGB color pixel to 8 bits by 3:3:2 packing.

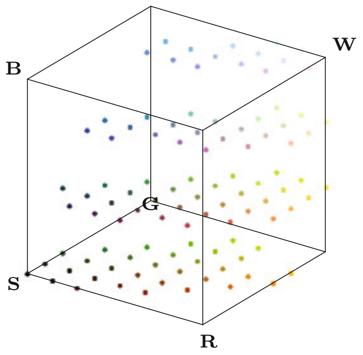
```java
ColorProcessor cp = (ColorProcessor) ip;
int C = cp.getPixel(u, v);
int R = (C & 0x00ff0000) >> 16;
int G = (C & 0x0000ff00) >> 8;
int B = (C & 0x000000ff);
// 3:3:2 uniform color quantization
byte RGB =
    (byte) ((R & 0xE0) | (G & 0xE0)>>3 | ((B & 0xC0)>>6));
```

Unlike the techniques described in the following, scalar quantization does not take into account the distribution of colors in the original image. Scalar quantization is an optimal solution only if the image colors are *uniformly* distributed within the RGB cube. However, the typical color distribution in natural images is anything but uniform, with some regions of the color space being densely populated and many colors entirely missing. In this case, scalar quantization is not optimal because the interesting colors may not be sampled with sufficient density while at the same time colors are represented that do not appear in the image at all.

(a)



(b)



(c)

Fig. 13.2
Color distribution after a scalar 3:3:2 quantization. Original color image (a). Distribution of the original 226,321 colors (b) and the remaining $8 \times 8 \times 4 = 256$ colors after 3:3:2 quantization (c) in the RGB color cube.

13.2 Vector Quantization

Vector quantization does not treat the individual color components separately as does scalar quantization, but each color vector $\mathbf{C}_i = (r_i, g_i, b_i)$ or pixel in the image is treated as a single entity. Starting from a set of original color tuples $\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m\}$, the task of vector quantization is

a) to find a set of n representative color vectors $\mathcal{C}' = \{\mathbf{c}'_1, \mathbf{c}'_2, \ldots, \mathbf{c}'_n\}$ and
b) to replace each original color $\mathbf{C}_i$ by one of the new color vectors $\mathbf{C}'_j \in \mathcal{C}'$,

where n is usually predetermined ($n < m$) and the resulting deviation from the original image shall be minimal. This is a combinatorial optimization problem in a rather large search space, which usually makes it impossible to determine a global optimum in adequate time. Thus all of the following methods only compute a "local" optimum at best.

13.2.1 Populosity Algorithm

The populosity algorithm[1] [104] selects the n most frequent colors in the image as the representative set of color vectors $\mathcal{C}'$. Being very easy to implement, this procedure is quite popular. The method described in Sec. 12.3.1, based on sorting the image pixels, can be used to determine the n most frequent image colors. Each original

[1] Sometimes also called the "popularity" algorithm.

pixel $\mathbf{C}_i$ is then replaced by the closest representative color vector in $\mathcal{C}'$; that is, the quantized color vector with the smallest distance in the 3D color space.

The algorithm performs sufficiently only as long as the original image colors are not widely scattered through the color space. Some improvement is possible by grouping similar colors into larger cells first (by scalar quantization). However, a less frequent (but possibly important) color may get lost whenever it is not sufficiently similar to any of the n most frequent colors.

13.2.2 Median-Cut Algorithm

The median-cut algorithm [104] is considered a classical method for color quantization that is implemented in many applications (including ImageJ). As in the populosity method, a color histogram is first computed for the original image, traditionally with a reduced number of histogram cells (such as $32 \times 32 \times 32$) for efficiency reasons.[2] The initial histogram volume is then recursively split into smaller boxes until the desired number of representative colors is reached. In each recursive step, the color box representing the largest number of pixels is selected for splitting. A box is always split across the longest of its three axes at the median point, such that half of the contained pixels remain in each of the resulting subboxes (Fig. 13.3).

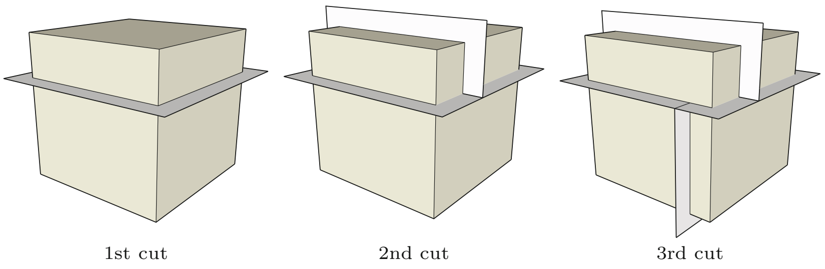


Fig. 13.3 Median-cut algorithm. The RGB color space is recursively split into smaller cubes along one of the color axes.

The result of this recursive splitting process is a partitioning of the color space into a set of disjoint boxes, with each box ideally containing the same number of image pixels. In the last step, a representative color vector (e.g., the mean vector of the contained colors) is computed for each color cube, and all the image pixels it contains are replaced by that color.

The advantage of this method is that color regions of high pixel density are split into many smaller cells, thus reducing the overall quantization error. In color regions of low density, however, relatively large cubes and thus large color deviations may occur for individual pixels.

The median-cut method is described in detail in Algorithms 13.1–13.3 and a corresponding Java implementation can be found in the source code section of this book's website (see Sec. 13.2.5).

[2] This corresponds to a scalar prequantization on the color components, which leads to additional quantization errors and thus produces suboptimal results. This step seems unnecessary on modern computers and should be avoided.

```
1: MedianCut($I, K_{\max}$)
       $I$: color image, $K_{\max}$: max. number of quantized colors
       Returns a new quantized image with at most $K_{\max}$ colors.
2:     $\mathcal{C}_q \leftarrow$ FindRepresentativeColors($I, K_{\max}$)
3:     return QuantizeImage($I, \mathcal{C}_q$)                    ▷ see Alg. 13.3
────────────────────────────────────────────────────────────────
4: FindRepresentativeColors($I, K_{\max}$)
       Returns a set of up to $K_{\max}$ representative colors for the image $I$.
5:     Let $\mathcal{C} = \{c_1, c_2, \ldots, c_K\}$ be the set of distinct colors in $I$. Each of
          the $K$ color elements in $\mathcal{C}$ is a tuple $c_i = \langle \mathsf{red}_i, \mathsf{grn}_i, \mathsf{blu}_i, \mathsf{cnt}_i \rangle$
          consisting of the RGB color components (red, grn, blu) and
          the number of pixels (cnt) in $I$ with that particular color.
6:     if $|\mathcal{C}| \le K_{\max}$ then
7:         return $\mathcal{C}$
8:     else
           Create a color box $b_0$ at level 0 that contains all image colors
           $\mathcal{C}$ and make it the initial element in the set of color boxes $\mathcal{B}$:
9:         $b_0 \leftarrow$ CreateColorBox($C, 0$)                  ▷ see Alg. 13.2
10:        $\mathcal{B} \leftarrow \{b_0\}$                          ▷ initial set of color boxes
11:        $k \leftarrow 1$
12:        $done \leftarrow$ false
13:        while $k < N_{\max}$ and $\neg done$ do
14:            $b \leftarrow$ FindBoxToSplit($\mathcal{B}$)            ▷ see Alg. 13.2
15:            if $b \ne$ nil then
16:                $(b_1, b_2) \leftarrow$ SplitBox($b$)           ▷ see Alg. 13.2
17:                $\mathcal{B} \leftarrow \mathcal{B} - \{b\}$                  ▷ remove $b$ from $\mathcal{B}$
18:                $\mathcal{B} \leftarrow \mathcal{B} \cup \{b_1, b_2\}$            ▷ insert $b_1$, $b_2$ into $\mathcal{B}$
19:                $k \leftarrow k + 1$
20:            else                                  ▷ no more boxes to split
21:                $done \leftarrow$ true
           Collect the average colors of all color boxes in $\mathcal{B}$:
22:        $\mathcal{C}_q \leftarrow \{$AverageColor($b_j$) $| \; b_j \in \mathcal{B}\}$       ▷ see Alg. 13.3
23:    return $\mathcal{C}_q$
```

Alg. 13.1
Median-cut color quantization (part 1). The input image I is quantized to up to $K_{\max}$ representative colors and a new, quantized image is returned. The main work is done in procedure FindRepresentativeColors(), which iteratively partitions the color space into increasingly smaller boxes. It returns a set of representative colors ($\mathcal{C}_q$) that are subsequently used by procedure QuantizeImage() to quantize the original image I. Note that (unlike in most common implementations) no prequantization is applied to the original image colors.

13.2.3 Octree Algorithm

Similar to the median-cut algorithm, this method is also based on partitioning the 3D color space into cells of varying size. The octree algorithm [82] utilizes a hierarchical structure, where each cube in color space may contain eight subcubes. This partitioning is represented by a tree structure (octree) with a cube at each node that may again link to up to eight further nodes. Thus each node corresponds to a subrange of the color space that reduces to a single color point at a certain tree depth d (e.g., $d = 8$ for a 3×8-bit RGB color image).

When an image is processed, the corresponding quantization tree, which is initially empty, is created dynamically by evaluating all pixels in a sequence. Each pixel's color tuple is inserted into the quantization tree, while at the same time the number of nodes is limited to a predefined value K (typically 256). When a new color tuple $\mathbf{C}_i$ is inserted and the tree does not contain this color, one of the following situations can occur:

Alg. 13.2
Median-cut color quantization (part 2).

```
1:  CreateColorBox($\mathcal{C}, m$)
        Creates and returns a new color box containing the colors $\mathcal{C}$ and
        level $m$. A color box $\boldsymbol{b}$ is a tuple ⟨colors, level, rmin, rmax, gmin,
        gmax, bmin, bmax⟩, where colors is the set of image colors repre-
        sented by the box, level denotes the split-level, and rmin, ..., bmax
        describe the color boundaries of the box in RGB space.
        Find the RGB extrema of all colors in $\mathcal{C}$:
2:      $r_{\min}, g_{\min}, b_{\min} \leftarrow +\infty$
3:      $r_{\max}, g_{\max}, b_{\max} \leftarrow -\infty$
4:      for all $\boldsymbol{c} \in \mathcal{C}$ do
            $r_{\min} \leftarrow \min\,(r_{\min},\ \mathsf{red}(\boldsymbol{c}))$
            $r_{\max} \leftarrow \max\,(r_{\max},\ \mathsf{red}(\boldsymbol{c}))$
5:          $g_{\min} \leftarrow \min\,(g_{\min},\ \mathsf{grn}(\boldsymbol{c}))$
            $g_{\max} \leftarrow \max\,(g_{\max},\ \mathsf{grn}(\boldsymbol{c}))$
            $b_{\min} \leftarrow \min\,(b_{\min},\ \mathsf{blu}(\boldsymbol{c}))$
            $b_{\max} \leftarrow \max\,(b_{\max},\ \mathsf{blu}(\boldsymbol{c}))$
6:      $\boldsymbol{b} \leftarrow \langle \mathcal{C}, m, r_{\min}, r_{\max}, g_{\min}, g_{\max}, b_{\min}, b_{\max} \rangle$
7:      return $\boldsymbol{b}$

8:  FindBoxToSplit($\mathcal{B}$)
        Searches the set of boxes $\mathcal{B}$ for a box to split and returns this
        box, or nil if no splittable box can be found.
        Find the set of color boxes that can be split (i.e., contain at least
        2 different colors):
9:      $\mathcal{B}_s \leftarrow \{\, \boldsymbol{b} \mid \boldsymbol{b} \in \mathcal{B} \wedge |\mathsf{colors}(\boldsymbol{b})| \geq 2\}$
10:     if $\mathcal{B}_s = \{\}$ then                          ▷ no splittable box was found
11:         return nil
12:     else
            Select a box $\boldsymbol{b}_x$ from $\mathcal{B}_s$, such that level($\boldsymbol{b}_x$) is a minimum:
13:         $\boldsymbol{b}_x \leftarrow \operatorname*{argmin}_{\boldsymbol{b} \in \mathcal{B}_s}(\mathsf{level}(\boldsymbol{b}))$
14:         return $\boldsymbol{b}_x$

15: SplitBox($\boldsymbol{b}$)
        Splits the color box $\boldsymbol{b}$ at the median plane perpendicular to its
        longest dimension and returns a pair of new color boxes.
16:     $m \leftarrow \mathsf{level}(\boldsymbol{b})$
17:     $d \leftarrow \mathsf{FindMaxBoxDimension}(\boldsymbol{b})$                  ▷ see Alg. 13.3
18:     $\mathcal{C} \leftarrow \mathsf{colors}(\boldsymbol{b})$                  ▷ the set of colors in box $\boldsymbol{b}$
            From all colors in $\mathcal{C}$ determine the median of the color dis-
            tribution along dimension $d$ and split $\mathcal{C}$ into $\mathcal{C}_1$, $\mathcal{C}_2$:
19:     $\mathcal{C}_1 \leftarrow \begin{cases} \{\boldsymbol{c} \in \mathcal{C} \mid \mathsf{red}(\boldsymbol{c}) \leq \operatorname*{median}_{\boldsymbol{c} \in \mathcal{C}}(\mathsf{red}(\boldsymbol{c}))\} & \text{for } d = \mathsf{Red} \\ \{\boldsymbol{c} \in \mathcal{C} \mid \mathsf{grn}(\boldsymbol{c}) \leq \operatorname*{median}_{\boldsymbol{c} \in \mathcal{C}}(\mathsf{grn}(\boldsymbol{c}))\} & \text{for } d = \mathsf{Green} \\ \{\boldsymbol{c} \in \mathcal{C} \mid \mathsf{blu}(\boldsymbol{c}) \leq \operatorname*{median}_{\boldsymbol{c} \in \mathcal{C}}(\mathsf{blu}(\boldsymbol{c}))\} & \text{for } d = \mathsf{Blue} \end{cases}$
20:     $\mathcal{C}_2 \leftarrow \mathcal{C} \setminus \mathcal{C}_1$
21:     $\boldsymbol{b}_1 \leftarrow \mathsf{CreateColorBox}(\mathcal{C}_1, m+1)$
22:     $\boldsymbol{b}_2 \leftarrow \mathsf{CreateColorBox}(\mathcal{C}_2, m+1)$
23:     return $(\boldsymbol{b}_1, \boldsymbol{b}_2)$
```

1. If the number of nodes is less than K and no suitable node for the color $\mathbf{c}_i$ exists already, then a new node is created for $\mathbf{C}_i$.
2. Otherwise (i.e., if the number of nodes is K), the existing nodes at the maximum tree depth (which represent similar colors) are merged into a common node.

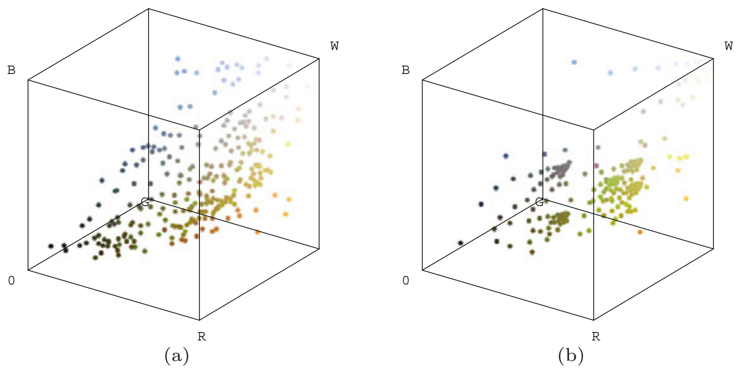
Alg. 13.3 Median-cut color quantization (part 3).

```
1:  AverageColor($\boldsymbol{b}$)
        Returns the average color $\boldsymbol{c}_{\mathrm{avg}}$ for the pixels represented by the
        color box $\boldsymbol{b}$.
2:      $\mathcal{C} \leftarrow$ colors($\boldsymbol{b}$)                    ▷ the set of colors in box $\boldsymbol{b}$
3:      $n \leftarrow 0$
4:      $\Sigma_{\mathrm{r}} \leftarrow 0, \quad \Sigma_{\mathrm{g}} \leftarrow 0, \quad \Sigma_{\mathrm{b}} \leftarrow 0$
5:      for all $\boldsymbol{c} \in \mathcal{C}$ do
6:          $k \leftarrow$ cnt($\boldsymbol{c}$)
7:          $n \leftarrow n + k$
8:          $\Sigma_{\mathrm{r}} \leftarrow \Sigma_{\mathrm{r}} + k \cdot$ red($\boldsymbol{c}$)
9:          $\Sigma_{\mathrm{g}} \leftarrow \Sigma_{\mathrm{g}} + k \cdot$ grn($\boldsymbol{c}$)
10:         $\Sigma_{\mathrm{b}} \leftarrow \Sigma_{\mathrm{b}} + k \cdot$ blu($\boldsymbol{c}$)
11:     $\bar{\boldsymbol{c}} \leftarrow (\Sigma_{\mathrm{r}}/n, \Sigma_{\mathrm{g}}/n, \Sigma_{\mathrm{b}}/n)$
12:     return $\bar{\boldsymbol{c}}$

13: FindMaxBoxDimension($\boldsymbol{b}$)
        Returns the largest dimension of the color box $\boldsymbol{b}$ (Red, Green, or
        Blue).
14:     $d_{\mathrm{r}} =$ rmax($\boldsymbol{b}$) − rmin($\boldsymbol{b}$)
15:     $d_{\mathrm{g}} =$ gmax($\boldsymbol{b}$) − gmin($\boldsymbol{b}$)
16:     $d_{\mathrm{b}} =$ bmax($\boldsymbol{b}$) − bmin($\boldsymbol{b}$)
17:     $d_{\max} = \max(d_{\mathrm{r}}, d_{\mathrm{g}}, d_{\mathrm{b}})$
18:     if $d_{\max} = d_{\mathrm{r}}$ then
19:         return Red.
20:     else if $d_{\max} = d_{\mathrm{g}}$ then
21:         return Green
22:     else
23:         return Blue

24: QuantizeImage($\boldsymbol{I}, \mathcal{C}_{\mathrm{q}}$)
        Returns a new image with color pixels from $\boldsymbol{I}$ replaced by their
        closest representative colors in $\mathcal{C}_{\mathrm{q}}$.
25:     $\boldsymbol{I}' \leftarrow$ duplicate($\boldsymbol{I}$)                    ▷ create a new image
26:     for all image coordinates $(u, v)$ do
            Find the quantization color in $\mathcal{C}_{\mathrm{q}}$ that is “closest” to the cur-
            rent pixel color (e.g., using the Euclidean distance in RGB
            space):
27:         $\boldsymbol{I}'(u,v) \leftarrow \underset{\boldsymbol{c} \in \mathcal{C}_{\mathrm{q}}}{\operatorname{argmin}} \|\boldsymbol{I}(u,v) - \boldsymbol{c}\|$
28:     return $\boldsymbol{I}'$
```

A key advantage of the iterative octree method is that the number of color nodes remains limited to K in any step and thus the amount of required storage is small. The final replacement of the image pixels by the quantized color vectors can also be performed easily and efficiently with the octree structure because only up to eight comparisons (one at each tree layer) are necessary to locate the best-matching color for each pixel.

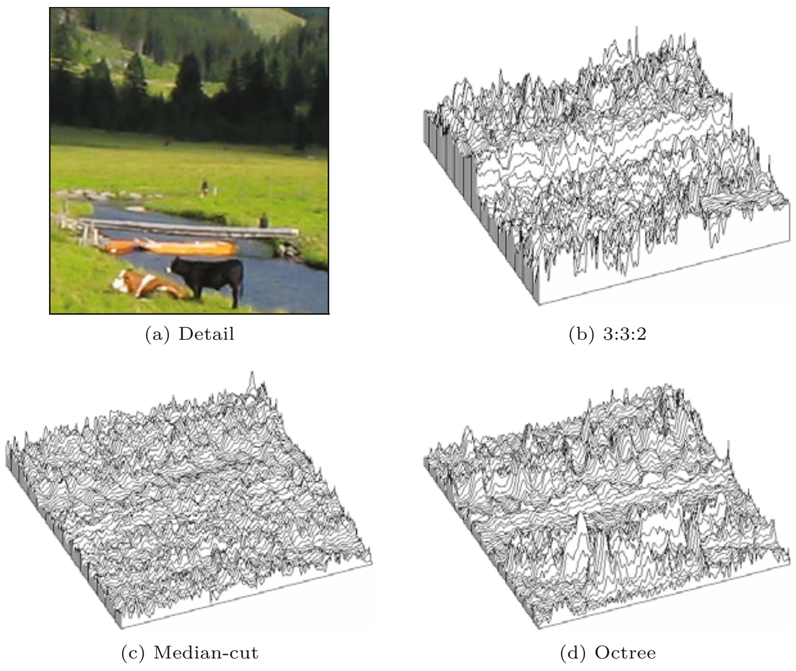
Figure 13.4 shows the resulting color distributions in RGB space after applying the median-cut and octree algorithms. In both cases, the original image (Fig. 13.2(a)) is quantized to 256 colors. Notice in particular the dense placement of quantized colors in certain regions of the green hues. For both algorithms and the (scalar) 3:3:2 quan-

Fig. 13.4
Color distribution after application of the median-cut (a) and octree (b) algorithms. In both cases, the set of 226,321 colors in the original image (Fig. 13.2(a)) was reduced to 256 representative colors.



tization, the resulting distances between the original pixels and the quantized colors are shown in Fig. 13.5. The greatest error naturally results from 3:3:2 quantization, because this method does not consider the contents of the image at all. Compared with the median-cut method, the overall error for the octree algorithm is smaller, although the latter creates several large deviations, particularly inside the colored foreground regions and the forest region in the background. In general, however, the octree algorithm does not offer significant advantages in terms of the resulting image quality over the simpler median-cut algorithm.

Fig. 13.5
Quantization errors. Original image (a), distance between original and quantized color pixels for scalar 3:3:2 quantization (b), median-cut (c), and octree (d) algorithms.



13.2.4 Other Methods for Vector Quantization

A suitable set of representative color vectors can usually be determined without inspecting all pixels in the original image. It is often sufficient to use only 10% of randomly selected pixels to obtain a high probability that none of the important colors is lost.

In addition to the color quantization methods described already, several other procedures and refined algorithms have been proposed. This includes statistical and clustering methods, such as the classical *k-means* algorithm, but also the use of neural networks and genetic algorithms. A good overview can be found in [219].

13.2.5 Java Implementation

The Java implementation[3] of the algorithms described in this chapter consists of a common interface `ColorQuantizer` and the concrete classes

- `MedianCutQuantizer`,
- `OctreeQuantizer`.

Program 13.2 shows a complete ImageJ plugin that employs the class `MedianCutQuantizer` for quantizing an RGB full-color image to an indexed image. The choice of data structures for the representation of color sets and the implementation of the associated set operations are essential to achieve good performance. The data structures used in this implementation are illustrated in Fig. 13.6.

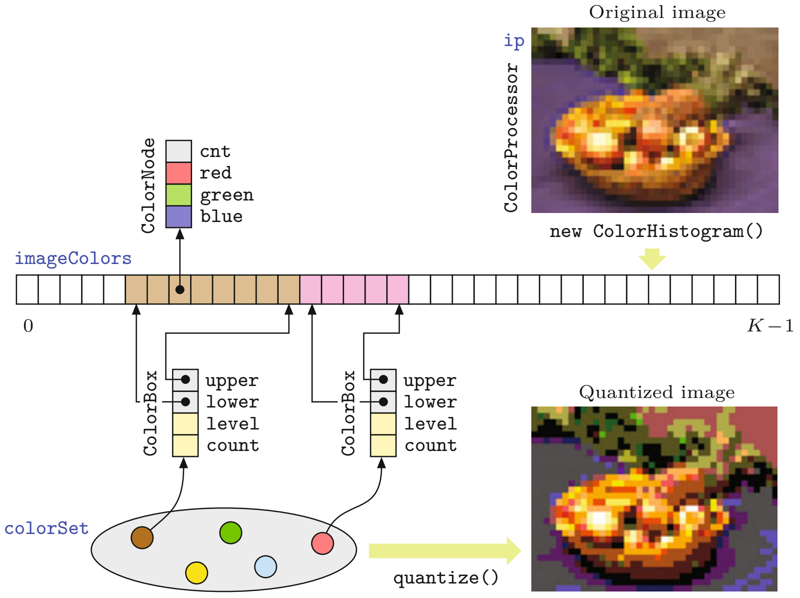
Initially, the set of all colors contained in the original image (`ip` of type `ColorProcessor`) is computed by `new ColorHistogram()`. The result is an array `imageColors` of size K Each cell of `imageColors` refers to a `colorNode` object ($\boldsymbol{c}_i$) that holds the associated color (`red`, `green`, `blue`) and its frequency (`cnt`) in the image. Each `colorBox` object (corresponding to a color box $\boldsymbol{b}$ in Alg. 13.1) selects a contiguous range of image colors, bounded by the indices `lower` and `upper`. The ranges of elements in `imageColors`, indexed by different `colorBox` objects, never overlap. Each element in `imageColors` is contained in exactly one `colorBox`; that is, the color boxes held in `colorSet` ($\mathcal{B}$ in Alg. 13.1) form a partitioning of `imageColors` (`colorSet` is implemented as a list of `ColorBox` objects). To split a particular `colorBox` along a color dimension $d =$ Red, Green, or Blue, the corresponding subrange of elements in `imageColors` is *sorted* with the property `red`, `green`, or `blue`, respectively, as the sorting key. In Java, this is quite easy to implement using the standard `Arrays.sort()` method and a dedicated `Comparator` object for each color dimension. Finally, the method `quantize()` replaces each pixel in `ip` by the closest color in `colorSet`.

13.3 Exercises

Exercise 13.1. Simplify the 3:3:2 quantization given in Prog. 13.1 such that only a single bit mask/shift step is performed for each color component.

[3] Package `imagingbook.pub.color.quantize`.

Fig. 13.6
Data structures used in the implementation of the median-cut quantization algortihm (class `MedianCutQuantizer`).



Exercise 13.2. The median-cut algorithm for color quantization (Sec. 13.2.2) is implemented in the *Independent JPEG Group's*[4] `libjpeg` open source software with the following modification: the choice of the cube to be split next depends alternately on (a) the number of contained image pixels and (b) the cube's geometric volume. Consider the possible motives and discuss examples where this approach may offer an improvement over the original algorithm.

Exercise 13.3. The *signal-to-noise ratio* (SNR) is a common measure for quantifying the loss of image quality introduced by color quantization. It is defined as the ratio between the average *signal energy* P_{signal} and the average *noise energy* P_{noise}. For example, given an original color image $\boldsymbol{I}$ and the associated quantized image $\boldsymbol{I}'$, this ratio could be calculated as

$$\mathrm{SNR}(\boldsymbol{I},\boldsymbol{I}') = \frac{P_{\mathrm{signal}}}{P_{\mathrm{noise}}} = \frac{\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} \|\boldsymbol{I}(u,v)\|^2}{\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} \big\|\boldsymbol{I}(u,v)-\boldsymbol{I}'(u,v)\big\|^2} \,. \tag{13.2}$$

Thus all deviations between the original and the quantized image are considered "noise". The signal-to-noise ratio is usually specified on a logarithmic scale with the unit *decibel* (dB), that is,

$$\mathrm{SNR}_{\log}(\boldsymbol{I},\boldsymbol{I}') = 10 \cdot \log_{10}(\mathrm{SNR}(\boldsymbol{I},\boldsymbol{I}')) \;\; [\mathrm{dB}]. \tag{13.3}$$

Implement the calculation of the SNR, as defined in Eqns. (13.2)–(13.3), for color images and compare the results for the median-cut and the octree algorithms for the same number of target colors.

[4] www.ijg.org.

```java
import ij.ImagePlus;
import ij.plugin.filter.PlugInFilter;
import ij.process.ByteProcessor;
import ij.process.ColorProcessor;
import ij.process.ImageProcessor;
import imagingbook.pub.color.quantize.ColorQuantizer;
import imagingbook.pub.color.quantize.MedianCutQuantizer;

public class Median_Cut_Quantization implements
    PlugInFilter {
  static int NCOLORS = 32;

  public int setup(String arg, ImagePlus imp) {
    return DOES_RGB + NO_CHANGES;
  }

  public void run(ImageProcessor ip) {
    ColorProcessor cp = ip.convertToColorProcessor();
    int w = ip.getWidth();
    int h = ip.getHeight();

    // create a quantizer:
    ColorQuantizer q =
        new MedianCutQuantizer(cp, NCOLORS);

    // quantize cp to an indexed image:
    ByteProcessor idxIp = q.quantize(cp);
    (new ImagePlus("Quantized Index Image", idxIp)).show();

    // quantize cp to an RGB image:
    int[] rgbPix = q.quantize((int[]) cp.getPixels());
    ImageProcessor rgbIp =
          new ColorProcessor(w, h, rgbPix);
    (new ImagePlus("Quantized RGB Image", rgbIp)).show();
  }
}
```

Prog. 13.2
Color quantization by the median-cut method (ImageJ plugin). This example uses the class `MedianCutQuantizer` to quantize the original full-color RGB image into (a) an indexed color image (of type `ByteProcessor`) and (b) another RGB image (of type `ColorProcessor`). Both images are finally displayed.