Median-Cut Color Quantization

Fitting true-color images onto VGA displays

Anton Kruger

In many instances, the number of colors in an image exceeds the number of displayable colors. For example, the output from 24-bit color scanners and ray-tracing software produces what's known as *true-color* images, which simply means that the red, green, and blue (RGB) components are each eight bits wide. It is often said that a true-color image can have up to 2^{24}, or 16 million colors, but the image dimensions normally determine the upper limit; a 512×512 true-color image, for instance, can't have more than 512^2, or 262,000 colors.

While true-color displays are common on high-end workstations and some true-color video cards are becoming available for the PC, most of us still have to be satisfied with VGA cards that can display no more than 256 colors at a time. Another reason for wanting to convert a true-color image to a 256-color image is the savings in disk space. A 512×512 true-color image requires 512×512×3≈786 Kbytes of disk space, while a 256-color version of the image requires one byte per pixel, or 512×512≈262 Kbytes of disk space. The problem then is: Given a true-color image, what 256 colors should we use to display the image?

Anton works for Truda Software, a Fortran and C consulting firm. He can be reached through the DDJ *offices.*

Note that the RGB components of a color can be viewed as lying along the axes of a *color space*, or RGB cube; see Figure 1. For a 24-bit image, the color space is (practically speaking) continuous, since the smallest difference between colors is imperceptible. This continuous color space must be mapped to 256 discrete colors, which are then used to represent the color space. Mapping a continuous variable to a discrete set of values is called *quantization*.

A simple approach to color quantization is to take the 256 most-common colors in the true-color image. For each color in the input image, we then search for the closest color in the set of 256 colors, and display that color instead. Actually, the 256 colors are loaded in the display system's color map or video look-up table (LUT), and the pixels' LUT indexes are written out to the display hardware. The algorithm that picks the 256 most-common colors to load into the LUT is the *popularity algorithm*. Its simplicity comes at a price, because it gives poor results for many images. Other color-quantization algorithms may give excellent results, but may be very time consuming. For example, as Wan et al. report, it "may take more than 20 hours on a VAX 780 computer to produce 256 clusters for a full-color image" using an algorithm known as the *K-means algorithm*. Color quantization is a post-processing procedure on an image, and should normally take only a fraction of the time to generate the image. Thus, such run times are in most cases unacceptable.

Some algorithms quantize an image without regard to its content; their dominant feature is speed. They use a fixed number of standard colors. For example, 256 colors require eight bits of color per pixel, and the bits are divided among red, green, and blue. Since the human eye is less sensitive to blue, red and green each get three bits, and blue two bits. This gives eight shades of red, eight of green, and four of blue. Other colors are mixtures of these primaries. The problem is that for a particular image, most standard colors may never be used. For example, if an image contains a lot of red and very little blue, all the entries assigned to blue are effectively wasted. This is a fundamental flaw of image-independent quantizers—if the image

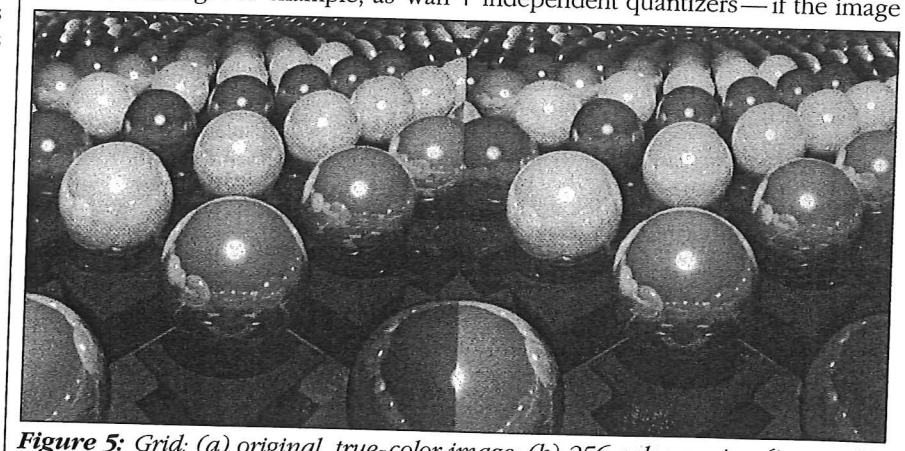


Figure 5: *Grid: (a) original, true-color image; (b) 256-color version (image-file courtesy of Dan Farmer).*

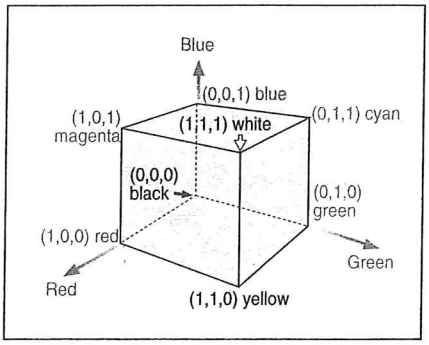
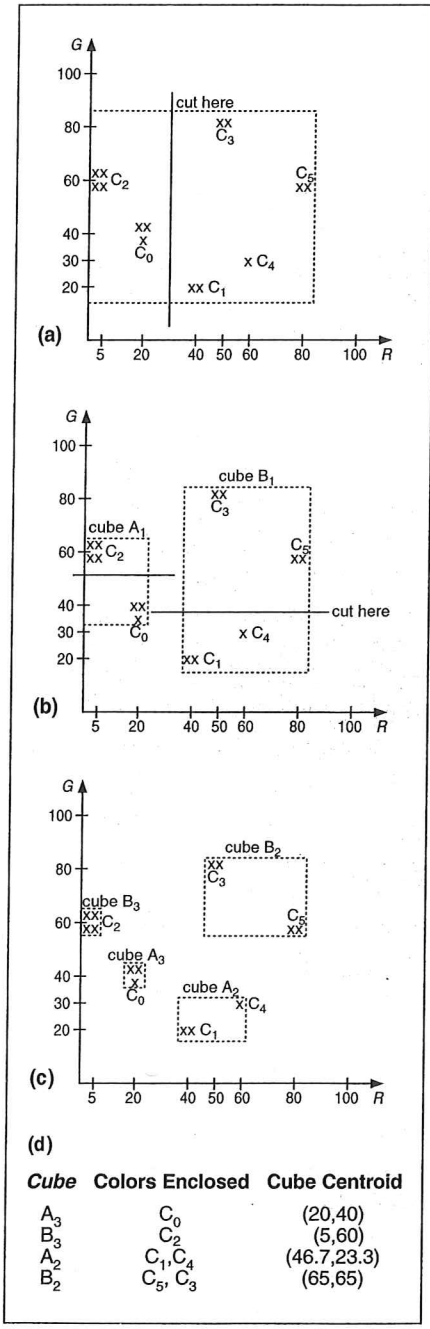


Figure 1: *The RGB cube.*



(d)

Cube	Colors Enclosed	Cube Centroid
A_3	C_0	(20,40)
B_3	C_2	(5,60)
A_2	C_1, C_4	(46.7,23.3)
B_2	C_5, C_3	(65,65)

Figure 2: *The median-cut algorithm: (a) original rectangle; (b) after splitting once; (c) after splitting three times; (d) output colors.*

(continued from page 46)

and the quantizer don't match, the results are poor.

The Median-cut Algorithm

Heckbert's *median-cut algorithm* is an image-based, color-quantization algorithm that gives good results for many images. If properly implemented, it's also quite fast. The aim of the median-cut algorithm is to have each of the 256 output colors represent the same number of pixels in the input image. The starting point is the RGB cube that corresponds to the whole image, around which a tight-fitting cube is placed. The cube is then split at the median of the longest axis. This ensures that about the same number of colors is assigned to each of the new cubes. The procedure is recursively applied to the two new cubes until 256 cubes are generated. The centroids (average values) of the cubes become the 256 output colors.

For example, Table 1 is a histogram for 14 pixels that have six unique colors; for clarity, the colors have no blue component. The task is to apply the median-cut algorithm to this histogram and find four output colors.

The initial rectangle is in Figure 2(a); crosses are used to indicate the pixels' colors. The distance from maximum to minimum along the *R*-axis is 80–5=75; along the *G*-axis, 80–20=60. Thus, the longest distance from maximum to minimum is along the *R*-axis. The median along this axis is *R*=30, so this is where the rectangle is split, and two tight-fitting rectangles are placed around the two new regions; see Figure 2(b).

When applying the procedure to the left rectangle, the axis with the longest distance is the *G*-axis. The median along this axis is *G*=50—this is where the axis is split. For the right rectangle, the axis with the longest distance from maximum to minimum is also the *G*-axis. The median along this axis is *G*=40, and this is where the axis is split. The final rectangles are shown in Figure 2(c), where each rectangle has about the same number of pixels. Following this, the centroids of the colors in each rectangle are computed. The resulting values are the output colors; see Figure 2(d).

There are two approaches to remapping an input image: *fast remapping* and *best remapping*. With fast remaps, the centroid of a cube represents all the colors enclosed by the cube. This is often good enough, but does not give the best results because the centroid of the cube in which a color falls may not be the closest centroid. Best remap searches the list of output colors for the closest color, but because searching is involved, it's slower.

Implementing the Median-cut Algorithm

The median-cut algorithm is normally described as recursive. However, it's easier and more practical to implement it as a nonrecursive routine. (Any recursive algorithm can be converted to an iterative algorithm.) To see why, apply the recursive algorithm in Figure 3(a), assuming it's used to generate four cubes.

The first division of the RGB cube results in two smaller cubes called *CubeA1* and *CubeB1*. Then *CubeA1* is split into *CubeA2* and *CubeB2*. Next *CubeA2* is split into *CubeA3* and *CubeB3*, rendering four cubes. See Figure 3(b), where *CubeB1* is never split, generating a depth-first spanning tree. To solve this problem, associate with each cube a level where the initial RGB cube has level 0, *CubeA1* and *CubeB1* have level 1, and so on. To end up with four cubes, we now require that the maximum level for any cube be $\log_2 4=2$; see Figure 3(c). Smaller cubes are also split on each recursive call. However, when the proper level is reached, the algorithm returns and splits the larger regions.

Another problem an implementation should address is the special case in which a cube encloses only one color, typically repeated many times. Obviously, such a cube cannot be split. One method of dealing with this is to split one of the other cubes; otherwise, you end up with one less cube than you want. To do this, you need a list of all the current cubes. However, this isn't available with the algorithms in Figure 3(a) and Figure 3(c), since you implicitly let the algorithm save the cubes on the program stack during the recursive calls. You can deal with this by increasing the maximum allowable level by 1 when encountering a cube with a single color. Unfortunately, this again opens the door for the problem of dividing smaller cubes while a larger cube is still waiting on the stack. Furthermore, the situation where the desired number of final cubes is not a power of 2, further complicates the process.

To address these problems, I used the following nonrecursive method. A list of the cubes computed so far is

Color	(*r*,*g*)-coordinates	Count
C_0	(20,40)	3
C_1	(40,20)	2
C_2	(5,60)	4
C_3	(50,80)	2
C_4	(60,30)	1
C_5	(80,50)	2

Table 1: *Histogram for median-cut example.*

(continued from page 48)
maintained in an array. During each iteration, the list of cubes is scanned for the cube with the smallest level, but cubes with single colors are ignored. Several cubes are normally candidates for splitting, and you could devise a rule for picking the best one to split, but I simply used the first one found. This cube is then split at the median, perpendicular to the longest axis. This increases the number of cubes by one. One of the new cubes is saved in the slot of the cube just split, and the other is added to the bottom of the list of cubes. When the desired number of cubes is generated, or all of the cubes enclose a single color, the splitting phase terminates. This algorithm is summarized in Figure 3(d).

At this point you have a list of cubes, and the next step is to compute their centroids. These colors are the output colors, or output color map. The histogram is no longer required, and you can use its space for a look-up table that holds the output colors—an *inverse color map*. The inverse color map works just like the histogram, where 24-bit colors are mapped to 15-bit colors that serve as indexes into the histogram. However, the inverse color map contains the input colors' indexes in the color map, instead of their count.

I've seen implementations of the median-cut algorithm that compute the inverse color map for the whole RGB cube. However, this is unnecessary because you know which colors each cube encloses, and they're the only ones required. Initialization of the inverse color map depends on the kind of remapping desired. With fast remap, initialization is accomplished by replacing the counts for all the colors in a cube with the centroid of the cube. With best remap, initialization is accomplished by replacing the counts for all the colors in a cube by the *closest* centroid, found by searching the list of centroids.

Finding the Median

The time-consuming part of quantization is finding the median along an axis of a cube. Algorithms exist for finding the median of a set of numbers with a run-time complexity of $O(N)$, but because of its simplicity, I used the following method. First sort the numbers in ascending order and compute their sum. Then start at the smallest number and compute a running sum. When the running sum is equal to half the total, you've found the median.

The aim of the median-cut algorithm is to have each of the 256 output colors represent the same number of pixels in the input image

I've used the C standard-library function *qsort* (based on the Quicksort algorithm with its average run-time complexity of $O(N \log_2 N)$) to do the sort. However, the algorithm has a potentially serious pitfall. When it's called to sort data that is already in order, the run-time complexity is $O(N^2)$. The difference between an algorithm with $O(N \log_2 N)$ complexity and one with $O(N^2)$ is dramatic when N is large. For example, with 32K=2^{15} colors, the $O(N^2)$ algorithm takes more than 2000 times longer. Because of the way the histogram is constructed, with five bits of each color packed into a 15-bit color, one color is always sorted. The macros in Listing One (page 91) are used to pack the 15-bit colors, which results in the image histogram being initially sorted on the blue component. A worst-case scenario is when a predominantly blue image is encountered—*qsort* is then called several times to sort a large set of colors already in order.

Another problem with Quicksort is that it is often implemented as a recursive routine, and can rapidly deplete the program stack when a large number of data points must be sorted. A nonrecursive implementation needs much less auxiliary storage. Thus, a good implementation of Quicksort is normally nonrecursive, and it randomizes the input data somewhat to provide for the case were the data is already in sorted order. Many implementations, however, don't do this. For example, I examined the Microsoft C 5.1 run-time library source, and its *qsort* is a fairly simple recursive implementation of Quicksort. For a production version of the median-cut algorithm, you might want to replace the *qsort* routine with a sort routine that's less efficient on the average, but has a better worst-case performance. Heapsort has both an average and worst-case run-time complexity of $O(N \log_2 N)$, and it needs no (or very little) auxiliary storage.

Data Structures

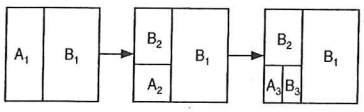
The histogram, which I call *Hist*, is accessed indirectly via the array *HistPtr* that functions as an index into the histogram. *HistPtr* contains the position of the colors in the actual histogram. When the histogram (or parts of it) must be sorted, *HistPtr* (or parts of it) is sorted. A structure is used to represent a cube. In Listing One, this structure (*cube_t*) has several members, but the essential members are *lower* and *upper*, and they point to the opposite corners of a cube in *HistPtr*.

To see how this works, return to the data in Table 1, and rework the example in terms of the *Hist* and *HistPtr*. Also, keep an eye on Figure 2 to see the correspondence between the data structures and the geometric interpretation of the median-cut algorithm. Assume that after the histogram is constructed, it looks like *Hist*; see Figure 4(a). Note

(a)

```
Split(Cube){
  if (ncubes == 4) return;
  find longest axis of Cube;
  cut Cube at median to form CubeA, CubeB;
  Split(CubeA);
  Split(CubeB);
}
```

(b)



(c)

```
maxlevel = 2;
Split(Cube,level){
  if (ncubes == 4) return;
  if (Cube's level == maxlevel) return;
  find longest axis of Cube;
  cut Cube at median to form CubeA, CubeB;
  Split(CubeA, level+1);
  Split(CubeB, level+1);
}
```

(d)

```
build initial cube from histogram;
set initial cube's level to 0;
insert initial cube in list of cubes;
ncubes = 1;
while (ncubes < maxcubes){
  search for Cube with smallest level;
  find the longest axis of Cube;
  find the median along this axis;
  cut Cube at median to form CubeA, CubeB;
  set CubeA's level = Cube's level + 1;
  set CubeB's level = Cube's level + 1;
  insert CubeA in Cube's slot;
  add CubeB to end of list of cubes;
  ncubes = ncubes + 1;
}
```

Figure 3: *(a) A flawed recursive implementation of the median-cut algorithm; (b) depth-first division as a result of the algorithm; (c) another recursive implementation; (d) pseudocode for the nonrecursive, cube-splitting algorithm.*

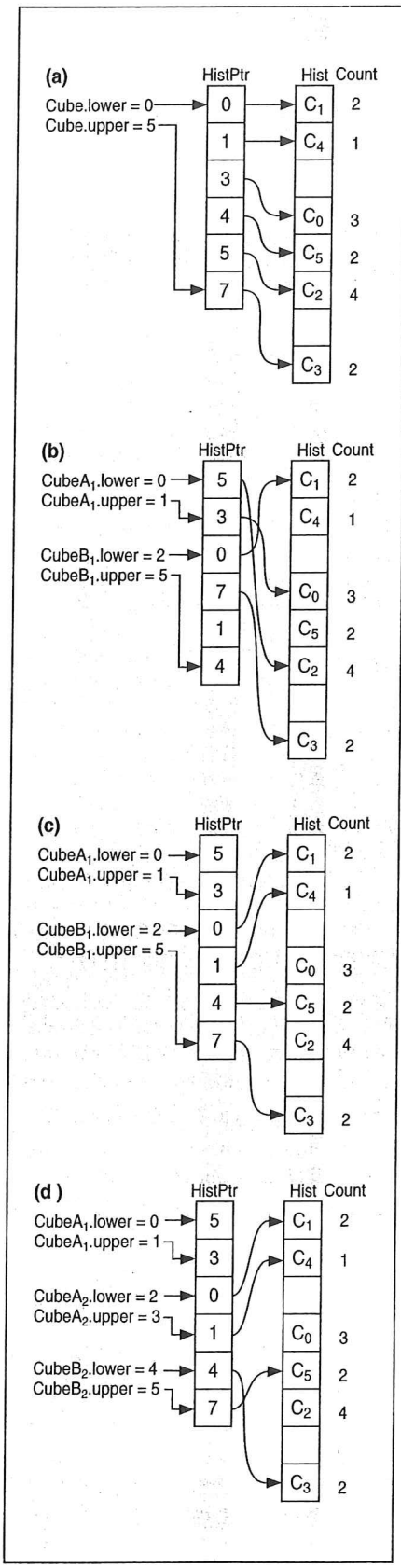


Figure 4: *(a)* Hist *and* HistPtr *after initialization; (b) sorted on red and split at the median; (c)* $CubeB_1$*, sorted on green; (d)* $CubeB_1$*, split at the median.*

(continued from page 50)

the "holes" in the histogram — this is typical. Also, this example assumes that the colors are initially sorted on green. C_1 has the smallest green component, C_4 has the second smallest, and so on. Just after *Hist* is constructed, *HistPtr* is filled as in Figure 4(a). The initial cube is the whole RGB cube, so that *Cube .lower* is set to 0, and *Cube.upper* to 5. This completes the initialization.

Now the cube must be split. The red axis is the longest, so we sort the colors along this axis. To do this, sort the array *HistPtr* instead of *Hist*. To find the median along the axis, start at *Cube .lower*; its value is 0. We look in *HistPtr[0]*; its value is 5, and this corresponds to C_2 in *Hist*. C_2's count is 4, and this is the running sum. Now we look at *Cube.lower*+1's corresponding color, C_0, and the running sum becomes 3+4=7. This is half the total for the cube, so it is split to form two cubes, $CubeA_1$, and $CubeB_1$. The result is in Figure 4(b).

Next we split $CubeB_1$. The colors in $CubeB_1$ are sorted along the longest axis, the green axis. This is shown in Figure 4(c), while Figure 4(d) depicts the situation after $CubeB_1$ is split. Finally, we split $CubeA_1$ to get the desired four cubes; see Table 2.

To compute the centroid for, say, $CubeB_2$, start at $CubeB_2.lower$, whose value is 4. Look in *HistPtr[4]* to find 4. The color at *Hist[4]* is C_5, and its count is 2. Next find the color that corresponds to $CubeB_2.lower$+1, which is also the last color in $CubeB_2$. This is color C_3, which has a count of 2. The centroid for this cube is shown in Example 1. All centroids are computed this way.

The MedianCut Program

MedianCut, the program in Listing One was developed on and for DOS PCs. To ease porting, I've tried to adhere to ANSI C. I also used *typedef*s for some variables when the number of bits was important. When moving to a different platform, you might want to change these so that they reflect the sizes of the target system. The code compiles and executes cleanly with Microsoft's 5.1 compiler (large memory model). If the /W4 compiler switch is used with version 6.0 of this compiler, some warning messages are issued, but they can safely be ignored.

MedianCut takes three arguments: the image histogram *Hist*, which contains the 15-bit colors; *maxcubes*, the desired number of output colors (this can be any number between 1 and 256, but the upper limit can be changed by altering the *#define MAXCOLORS* preprocessor directive); and a *maxcubes* × 3 array *ColMap* that MedianCut fills with the output colors, where the red component of color *i* is *r=ColMap[i][0]*; that of green, *g=ColMap[i][1];* and that of *blue*, *b=ColMap[i][2]*. MedianCut returns the number of actual colors, which may be less than *maxcubes* if the input image already contains less colors than the requested number.

The *#define FAST_REMAP* preprocessor directive controls whether the inverse color map is initialized with the fast-remap or the best-remap method. If this directive is deleted or commented out, initialization is done according to the best-remap method.

Additionally, I've written a typical driver routine for MedianCut that, in this

Cube	HistPtr.lower	HistPtr.upper	Colors Enclosed	Cube Centroid
A_3	0	0	C_0	(20,40)
B_3	1	1	C_2	(5,60)
A_2	2	3	C_1,C_4	(46.7,23.3)
B_2	4	5	C_5,C_3	(65,65)

Table 2: *Contents of the data structures after splitting three times.*

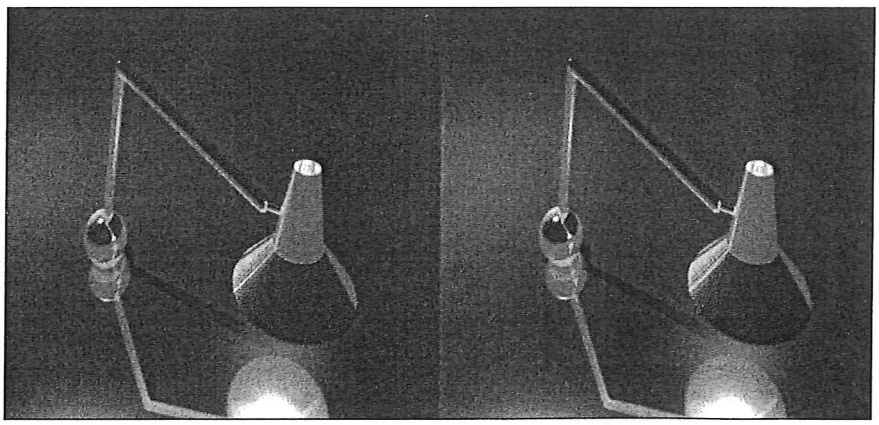


Figure 6: *Lamp: (a) original, true-color image; (b) 256-color version (anonymous image-file description).*

$$\frac{2C_5+2C_3}{2+2} = \frac{2(80,50)+2(50,80)}{4} = (65,65)$$

Example 1: *Computing cube centroids.*

(continued from page 52)
case, converts a Targa type-2 image file to a Targa type 1. Truevision's Targa type-2 image format is a popular format for true-color images, and Targa type-1 images are color-mapped, so to convert from type 1 to type 2, we must quantize the true-color image. This program is available electronically; see page 3.

Instead of dynamically allocating the space for *HistPtr* and the array for the list of cubes, I've chosen to have these static variables. These arrays occupy about 70 Kbytes. The histogram occupies another 64 Kbytes, and depending on your compiler's *qsort* routine, several Kbytes of stack may be required. Add to this about another 20 Kbytes for the file buffers, as well as the space required for the program code, and a total of about 200 Kbytes of memory are required.

Figures 5 and 6 each show two images before and after application of the median-cut algorithm (fast remap). Both were generated with the POV-Ray (Persistence of Vision) ray-tracing program (see "Ray Tracing and the POV-Ray Toolkit," by Craig A. Lindley, *DDJ*, July 1994). The 256-color version of "grid" is indistinguishable from the true-color original, which shows how effective the median-cut algorithm can be. However, with the "lamp" image, there are some false contours in areas of low contrast, but this may not show up in the photographs, since it is quite difficult to reproduce a video display faithfully in print.

This implementation of the median-cut algorithm is quite fast. With the fast-remap method, it takes about eight seconds to quantize 640×480 true-color versions of the images in Figures 5 and 6 on a 20-MHz AT clone. Much of this time is I/O, and the actual color quantization takes less than three seconds.

References

Wan, S.J., S.K.M. Wong, and P. Prusinkiewicz. "An Algorithm for Multidimensional Data Clustering." *ACM Transactions on Mathematical Software* (June 1988).

Heckbert, P. "Color Image Quantization for Frame Buffer Display." *Computer Graphics* (July 1982).

DDJ

(Listing begins on page 91.)