

A Few Good Colors

by Dave Pomerantz

Nowadays, digitized images use 24-bits per pixel. That's sixteen million colors. The problem is that most color monitors show only 256 colors at a time. I discovered this last summer as I struggled to get a video digitizer ready for the MacWorld trade show. Almost too close to the deadline, I found out that it's a long step between capturing a 24-bit image and making it pretty on an 8-bit screen.

Somehow I had to pick the 256 colors that best represented all the subtle shades of the original image. Then I had to map those thousands of shades into my chosen 256 colors. It's a tough problem, it's common to most image-handling software, and I think you'll find the solution a fascinating brain-twister in solid geometry.

I have to confess that I didn't invent the color-selection algorithm. I got it from Paul Heckbert's excellent theoretical article, *Color Image Quantization for Frame Buffer Display*.¹ In here, I'll show you how his algorithm works and how you could apply it to any limited-palette monitor, like the EGA and VGA displays. My own program runs on the Mac II, in 68000 assembler.

The Scanned Image

First, let's take a close look at a color image. Fresh off the digitizer, a typical image uses 24-bit pixels having one byte per color component:

RED		GREEN		BLUE	
23	16	15	8	7	0

Fig 1. 24-bit Pixel

Each pixel is one of 16 million variations of red, green, and blue (256 x 256 x 256). The pixels are laid out in a rectangle, 640 x 480, like this:

¹ Paul Heckbert, *Color Image Quantization for Frame Buffer Display*, *Computer Graphics* 16:3, pp 297-307. A publication of the Association for Computing Machinery.

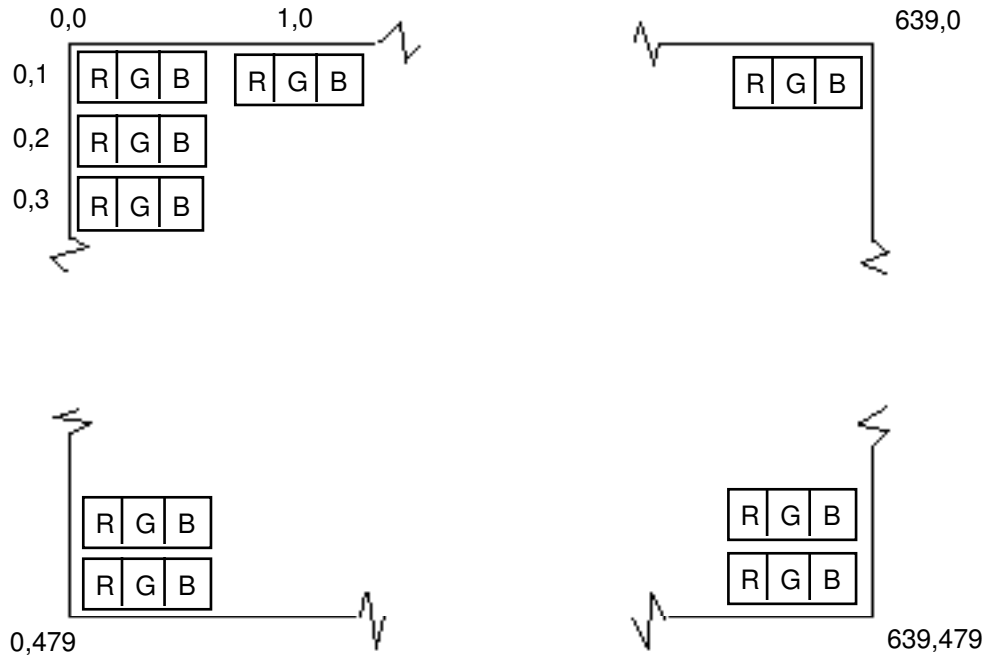


Fig 2. Scanned Image

This gives us a grand total of 307200 pixels per image, with three bytes per pixel, or just under a megabyte of data.

The Displayed Image

For simplicity, let's assume the display buffer for the color monitor is also 640x480. That way, the only difference between the image buffer and the display buffer is the size of the pixel. The display buffer uses 8-bit pixels which are not actually colors, but indexes into a small palette of colors.

On an 8-bit display card, like a VGA card for the PC or an Apple color card for the Macintosh, we choose a small palette of 256 colors from some larger range determined by the hardware. Then we load these colors into the color lookup table on the display card.² The lookup table translates an 8-bit palette index into the corresponding video display color.

² On the Macintosh, you don't directly load the video card. Instead, you go through the Palette Manager. This little piece of indirection leaves in Apple's hands some of the more subtle problems like hardware independence and multiple palettes.

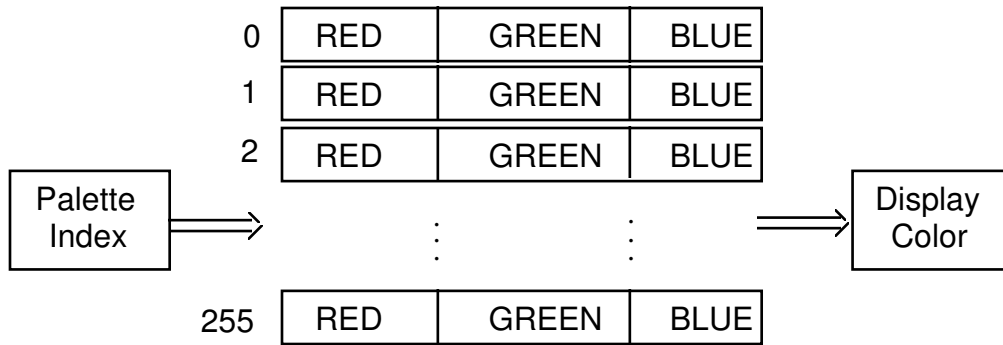


Fig 3. Hardware Color Lookup Table

Translating to the Display Buffer

Now we've set up the hardware to correctly translate an image composed of 8-bit palette indexes. Our digitized image, however, is built from 24-bit pixels. We need to translate these pixels into palette indexes and put them in the video display buffer.

Let's say our image pixel is described by its three color components, $R_iG_iB_i$. We want to find the palette index k such that the palette color, $R_kG_kB_k$, is the best match for the image pixel.

That means we want to minimize the difference equation:

$$d = \text{abs}(R_i - R_k) + \text{abs}(G_i - G_k) + \text{abs}(B_i - B_k)$$

Unfortunately, we need to calculate d for each of 256 palette entries, for each of 307,200 pixels. That's about 78 million calculations just to translate the image into palette indexes. Later, I'll show you how we can apply a strategy that gets around all these calculations.

So Here's the Problem

1. We need to choose a small palette of colors that best represents the full range of colors in the image.
2. We need to translate the image from 24-bit colors to 8-bit palette indexes.

Choosing the Palette

The first step in choosing the palette is to find out which colors are used in the image. Every pixel could be the same shade of blue, or every pixel could be a different color. To find out, we'll count how many times each color occurs, using a histogram.

A Color Histogram

If we collected a histogram for every possible color, we'd need 16 million one-word entries, that is, a 32MB table. To conserve memory, we'll limit the range of our histogram to 2^{15} entries, by using only the upper 5 bits of each color component.

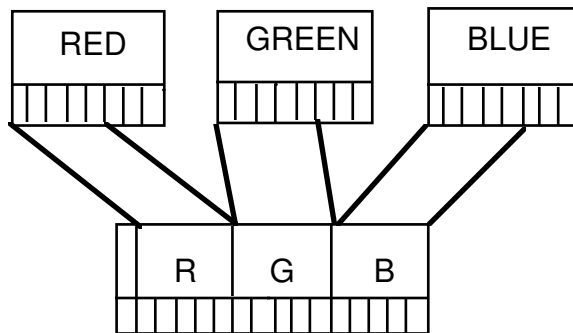


Fig 4. 24-bit to 15-bit Pixel Mapping

Naturally, this will cost us in color resolution. By ignoring the low 3 bits, we've lumped together the nearest 8 shades of each color component. Each histogram entry represents a group of colors, consisting of the 512 neighboring shades of red, green, and blue ($8 \times 8 \times 8$).

Does this hurt the appearance of our displayed image? Sure it does. But as we move from 16 million colors to 256 colors, we'll be losing quite a bit of color resolution anyway.

To calculate the histogram, first we set all entries to zero. Then we make one pass through the original image, converting every 24-bit pixel to a 15-bit index and incrementing the count at that index. Here's the code in C.³

³ I don't advocate you use my C code, it's intended only to illustrate the solution. In my client's application, I calculate the histogram while I'm scanning the image, so most of the actual code is in 68000 assembler.

```

#define MAXCOLORS    32768    /* 215 colors */
#define IMAGESIZE (640 * 480)

typedef unsigned char byte;

typedef struct {
    byte red;
    byte green;
    byte blue;
} RGB;

RGB    image[IMAGESIZE], *imageP;
short  histogram[MAXCOLORS], *histP;
short  15bitColor;

for (histP=histogram; histP < &histogram[MAXCOLORS]; ++histP)
    *histP = 0;

for (imageP=image; imageP < &image[IMAGESIZE]; ++imageP)
    {
    15bitColor = ((imageP->red & 0xF8) << 7) |
                ((imageP->green & 0xF8) << 2) |
                (imageP->blue >> 3);
    ++histogram[15bitColor];
    }

```

Listing 1. Converting to 15-bit Pixels

Listing the Colors

Most images don't contain every color. In my experience, a typical image captured with a good camera uses less than 5000 of the 32K colors available in the histogram. These are the colors we're interested in, and we don't need to waste time on the others. So we'll create an array called `colorList`, containing only those 15-bit color indexes that appeared in the image.

```
short      *colorList;
short 15bitColor;

colorList = malloc(sizeof(short) * MAXCOLORS);

currentSize = 0;
for (15bitColor=0; 15bitColor < MAXCOLORS; ++15bitColor)
    {
        if (histogram[15bitColor] != 0)
            colorList[currentSize++] = 15bitColor;
    }

colorList = realloc(colorList,currentSize);
```

Listing 2. Creating the Color List

Finally, we're ready to choose the colors.

The RGB Color Cube

Let's get into that solid geometry I promised you earlier. Imagine that all possible colors are represented with a three-dimensional matrix, where the three axes are the color components red, green, and blue. Each component has 5 bits of resolution, as in our histogram, or 32 shades. The cube looks something like this:

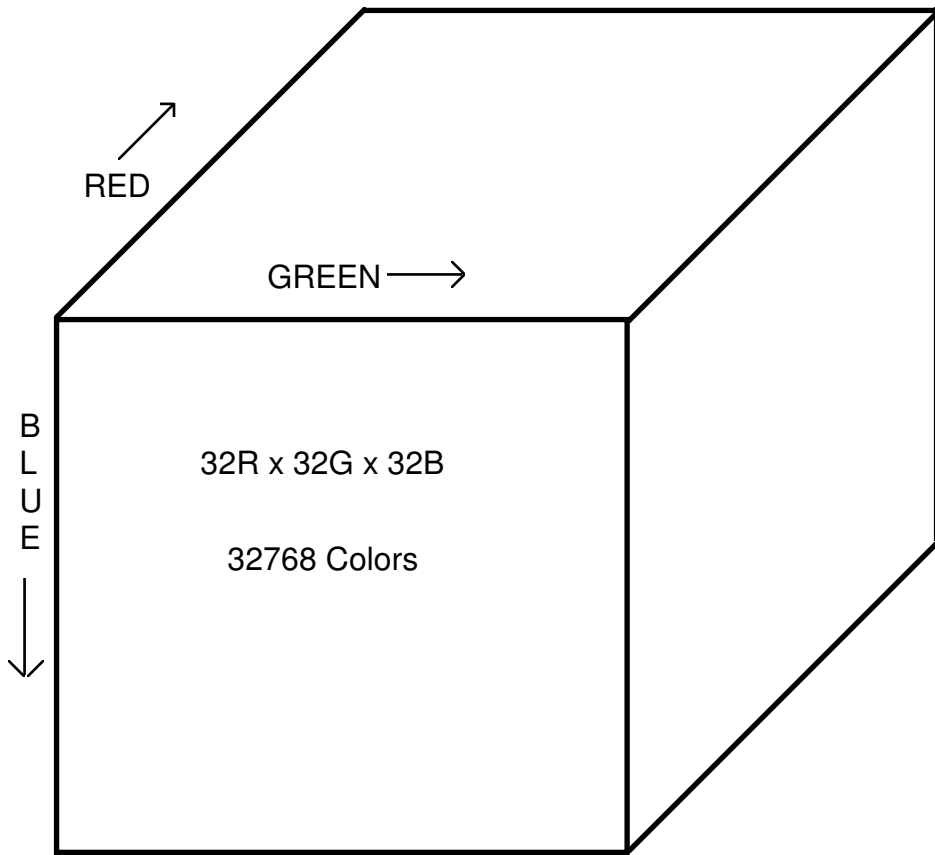


Fig 5. The RGB Color Cube

Our digitized image will typically use less than 5000 of the 32K colors. In fact, the color histogram is this cube, with the non-zero entries representing the colors in our image. What we'd like to do now is carve up the cube into 256 equal regions. But what do we mean by equal?

If we divide the color cube into regions of equal volume, then we'll get a uniform distribution of all the colors. This is called a uniform palette. It's a useful first approximation, but it doesn't take into account the actual distribution of colors in our image. That is, it treats a blue seaside vista the same as a crimson summer sunset.

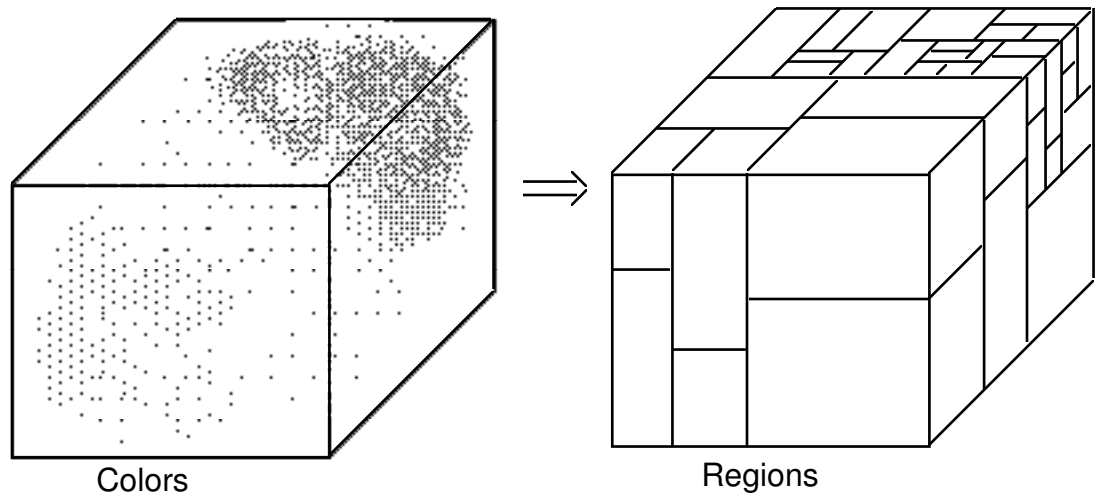


Fig 6. Creating Regions of Equal Color Count

Instead, we'd like to divide the cube into regions of equal color count, where the color count is the sum of the histogram counts enclosed by each region. For example, the sunset image has heavily shaded mixtures of red and green. That means the colors occur most frequently toward the far-upper-right corner, as shown above. Since we'll want our palette to have more colors in that area, we should carve more regions from that area. That means the regions should be smaller and more numerous where the histogram color counts are highest.

Now how do we write an algorithm that'll create those regions?

The Data Structure

Let's start by representing a 3-D rectangle within the color cube as a data structure called *colorRect*.

```
typedef struct {
    long first, last;           /* indexes into colorList      */
    long histSum;             /* total color count for region */

    byte redMin,redMax;       /* min's and max's            */
    byte greenMin,greenMax;   /* for the color components    */
    byte blueMin,blueMax;     /* define the region           */

    byte widestColor;        /* color with the widest range */
    byte red,green,blue;     /* our eventual palette color  */
} colorRect;
```

Listing 3. The colorRect structure

first, last: Remember that the colorList contains all colors that appeared in the image.

First and *last* define a range within colorList, the image colors that fell within the region. In the example shown in figure 7, *first* is 271 and *last* is 276.

histSum: This is the sum of the color counts for the colors in the region. For the example in figure 7, *histSum* is 820.

min's and max's: These are the geometric coordinates of the region. The min's represent the near-upper-left corner. The max's represent the far-lower-right corner. A typical region within the cube might look something like this:

colorList index	colorList entry	histogram count
271	(10,8,17)	315
272	(11,8,18)	118
273	(11,9,17)	104
274	(11,9,18)	33
275	(12,8,17)	98
276	(13,10,19)	152

Fig 7. Typical Region

The colorList for this region has only 6 colors, shown with their associated histogram counts. Notice how the list is sorted by red, then green, then blue. That's because the *widestColor* is red.

widestColor: The *widestColor* is the color component with the longest leg of the region. In the above example, red is the *widestColor*, with 4 shades.

red, green, blue: This is the palette color. When we've created every region, we'll come back and figure out what color we want in the palette for this region. Every image color that maps into this region will be displayed with this palette color.

The Algorithm

To initialize the algorithm, we'll create a single colorRect that encloses the entire RGB cube. We'll set the *min*'s to 0 and the *max*'s to 31, enclosing the cube. Then we'll set *first* to 0 and *last* to the end of colorList, to include all the colors that appeared in the image.

Now we're ready to divide the cube, in this fashion:

Scan the colorList. We'll scan the colorList from *first* to *last*, finding the minimums and maximums of each color component, and adding the color counts to *histSum*. After doing this, we can determine the *widestColor* by comparing the min's and max's and selecting the component with the biggest difference.

Sort the colorList. Now we'll sort the colors within the region in order of the *widestColor*. This prepares us for the next step.

Find the median. Scan through the colorList again, from *first* to *last*, adding the color counts to a running sum. When the running sum equals half the *histSum*, we've found the median point. We'll adjust this median point such that it always breaks between two shades of the *widestColor*.

In the example above, the best median is between (11,8,18) and (11,9,17), two colors that have the same shade of red. But since we need to break between shades of red, we'll set the median between (10,8,17) and (11,8,18).

Split into two regions. Create two colorRect's from the current colorRect, dividing the colorList range at the median.

We've created two regions from one region, by splitting the region on its longest axis at the point where the color counts in the two halves were roughly equal.

To build all 256 regions, we just continue this process until we've used 256 colorRect's. The time-consuming part is the number of times we sort the colorList. We sort it each time we create a new region, each time sorting successively smaller portions of the list. I used the 'quicker-sort' method, from a previous issue of *Computer Language*.⁴

What? No Recursion?

Any eager student of computer science would latch onto this as one of the rare opportunities to use recursion. As eager as the next guy, that's how I first tried to solve this problem, using this approach:

```
splitColorRect(colorRect *region)
{
    if 256 regions exist
        return;
    scan the colorList
    sort by widestColor
    find the median
    split into two regions
    call splitColorRect(lesser region)
    call splitColorRect(greater region)
}
```

Listing 4. An (Incorrect) Recursive Approach

As I discovered, recursion doesn't work. It creates what, I later remembered, is called a depth-first spanning tree, subdividing only the lesser regions in each recursive call. It never gets around to subdividing the greater regions.

Instead, at each iteration we want to choose the largest region of all remaining regions, for the next division. The method I wound up using chooses the region with the

⁴ Carl Reynolds, *Sorts of Sorts*, *Computer Language*, March 1988, p.56.

highest color count. To find that region, I built an index list for the colorRect structures, and wrote routines to insert and remove colorRect structures in order of their *histSum*.

Now our algorithm looks like this:

```
initialize first colorRect:
    colorRect.first = 0;
    colorRect.last = last index in colorList;
    scan colorList from first to last, to calculate
        widestColor and histSum
    insert first colorRect into indexed list

while (number of colorRects < 256)
    {
    get colorRect from top of list
    sort from colorRect.first to colorRect.last
        by colorRect.widestColor
    find the median point
    remove current colorRect from list
    split the colorRect at the median into two colorRects

    scan the colorList to define the min's, max's,
        and histSum of the first colorRect
    insert first colorRect in order of it's histSum

    scan the colorList to define the min's, max's,
        and histSum of the second colorRect
    insert second colorRect in order of it's histSum
    }
```

Listing 5. Correctly Dividing the RGB Cube

Assigning Colors to the Regions

Once you've created the regions, it's easy to assign them color values. Just add up all the colors that occurred in the region, multiply each by its histogram count, and divide by the total count for the region:

```

/* These macros split a 15-bit color into it's */
/* three 5-bit color components          */
#define RED(x) (((x) & 0x7C00) >> 10)
#define GREEN(x) (((x) & 0x03E0) >> 5)
#define BLUE(x) ((x) & 0x001F)

calcColor(colorRect *cr)
{
    short    15bitColor;
    byte  red, green, blue;

    red = green = blue = 0;
    for (i = cr->first; i < cr->last; ++i)
        {
            15bitColor = colorList[i];
            red  += RED(15bitColor) * histogram[15bitColor];
            green += GREEN(15bitColor) * histogram[15bitColor];
            blue += BLUE(15bitColor) * histogram[15bitColor];
        }
    red /= cr->histSum;
    green /= cr->histSum;
    blue /= cr->histSum;
}

```

Listing 6. Calculating the Colors

Calculate Palette Lookup Table

Up to this point we've been determining which colors appear in the palette. Now we need a lookup table that translates from an arbitrary 24-bit color to the index of the closest palette color. We saw earlier that it can take 78 million iterations to translate a 640x480 image. And I said we'd find a way around that.

Somehow we need to map the colors in our RGB cube to the palette indexes. But that's exactly what the colorRect structure does, with min's and max's that define the geometric bounds of each palette color. All we need to do is fill another RGB cube with palette indexes and we've got our lookup table.

Given an arbitrary 24-bit color, we first convert it to a 15-bit color. Then we use that 15-bit color as an index into the RGB cube, and look up the corresponding palette index. Any color that falls within that region gets the same palette index.

Here's the code to initialize the palette lookup table for one region:

```
bldLookup(colorRect *cr, byte paletteIndex)
{
    byte red, green, blue;
    short 15bitColor;

    for (red = cr->redMin; red <= cr->redMax; ++red)
        {
            for (green = cr->greenMin; green <= cr->greenMax;
                ++green)
                {
                    for (blue = cr->blueMin; blue <= cr->blueMax;
                        ++blue)
                        {
                            15bitColor = red << 10 | green << 5 | blue;
                            lookup[15bitColor] = paletteIndex;
                        }
                }
        }
}
```

Listing 7. Building the Palette Lookup Table

This algorithm is strictly linear, which means that the inner loop executes only once for each lookup table entry.

That's it. We've chosen 256 good colors from our original image and we've built a map to translate that image into the smaller palette.

But is it fast?

In this article we bridged the gap between 24-bit scanners and 8-bit monitors. We did this by modeling the data geometrically, and applying classic divide-and-conquer techniques. More to the point, perhaps, we let people like Paul Heckbert do this for us, and we concentrated on a good engineering implementation.

I can tell you the results in hard numbers. This algorithm executes in 6 seconds on a Mac IIx, converting a 640x480 image from 24-bit pixels to 8-bit indexes, yielding a

high-quality displayed image. A well-written but brute-force solution would takes about 70 seconds and would yield a lesser-quality image. It took me about three weeks to write and debug the code, resulting in 30 pages of assembler (1500 lines) and a 9K object file.

Future directions

Could we do any better? I think so. We chose to model the color space as an RGB cube because the three color components map into it so effortlessly. In terms of human perception, however, rectangular regions are not a good way to group colors.

What if we chose polar coordinates instead? Then we might represent a color as a vector, that is, an angle and a distance. A line of constant hue is any straight line that crosses the origin. The intensity is the distance from the origin. Color saturation is the difference between the angle of the vector and 45 degrees, since the 45-degree vector represents all shades of gray.

So how would we divide up this color space? What kind of geometry would we apply? Spheres have equal distance, but color perception isn't necessarily linear with intensity. I'd suggest that each color region is best represented as an ellipsis with an axis of constant hue. The ellipsis isn't a true ellipsis (my sixth-grade geometry is failing me, now) but has a width that expands with it's distance from the origin.

Now all we need is an algorithm that records a histogram in terms of polar coordinates, and splits the color space into these strangely-shaped ellipses. For now, I'll stick with rectangles.

Acknowledgements

I want to thank Dave Pratt and Vivian Russell of Digital Vision, Inc. for giving me permission to write this article. All the software described in this article was written under contract to Digital Vision as part of their ComputerEyes video digitizer.