Colormap:
A Color Image Quantizer†

Adam Rosen

87-845
June 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

†This work was supported in part by the National Science Foundation under grants DMC-8312096 and DCR-8502568 and the Office of Naval Research under contract N-00014-86-K-0281.
Colormap - A Color Image Quantizer

"A picture is worth 1024 words."
- Don Greenberg

ABSTRACT

This paper describes the Colormap program, designed and implemented for use in Cornell University's Computer Science Department. The purpose of Colormap is to quantize arbitrary 24-bit per pixel color images down to only 8 or 9 bits per pixel with minimal perceived image degradation. Uses for this include quantizing arbitrary pictures, mathematically generated images, and images which contain too many colors for display as a result of anti-aliasing. Uniform and adaptive techniques for generating the new color table are described, and then a 4-phase adaptive procedure for quantizing an image is presented. Factors including speed, accuracy, and complexity are examined in each phase, especially the construction of the color table and the mapping of these "representative" colors back upon the original image. These factors are presented in the context of a main k-d tree data structure and the various color space dissimilarity measures employed. Dithering the resulting image is an option for reducing any noticeable contouring. Finally, some initial results are presented along with a comparison between the uniform and adaptive methods implemented. Specific information on running and modifying the Colormap program is contained in the appendices.

1. INTRODUCTION

Color image quantization is the process by which color images (say those taken with a camera) are converted into a digital representation capable of being stored in a computer for later manipulation and display. It is helpful to think of the original image as consisting of an infinite number of points, each any possible color. Then the quantized image will consist of a fixed number of points, or pixels, each with its color represented numerically from a finite set of values. Color image quantization is the Engineer's way of playing "color by numbers."

For the purposes of this paper and most discussions on the subject, it is assumed that the images are already originally quantized to 24 bits of color information per pixel - one
8-bit byte for each of the red, green, and blue (rgb) phosphors of the graphic display terminals. It is also assumed that there is no perceptible error in this original quantization. Color image quantization therefore usually refers to shrinking the bits needed for each pixel by reducing the set of possible colors in the quantized image. Why would we want to decrease the bits/pixel if we already have a digitized image? The main reason for doing this is a hardware constraint: often the graphics screens can only hold 8 or 9 bits of color information per pixel. The goal of color image quantization is thus to take this input image and shrink the memory by about two-thirds to the maximum number of bits/pixel, allowing the image to be displayed. If the algorithms are chosen thoughtfully this quantized image can look almost identical to the original. (There are also the benefits of smaller memory requirements, especially if run-length encoding is used, and faster data transfer.) The resulting colors which are used in the quantized image are called the color representatives, since they are algorithm's choice of the "best" subset of colors to represent the original image.

Since there is no fast optimal solution to the color image quantization problem known (or multi-dimensional quantization problem), heuristic techniques are used to approximate the optimal [1]. The two general approaches use uniform and adaptive quantization methods. The uniform technique, as its name implies, uniformly truncates an approximately equal number of bits from each of the three color components in the original pixel. This always results in a uniform spread of color component values, each separated by equal increments. For example, to quantize a 24-bit pixel down to 9 bits, 5 bits would be lopped off each of the red, green, and blue components. Note that since this is done independently of the color composition of the original image, it may perform particularly poorly if these colors are not uniformly distributed. If the colors occur in concentrated groups (a frequent possibility due to image/object coherence) many of the calculated color representatives will be unused! Uniform quantization has the advantage, however, of being conceptually simple and is straightforward to implement. Colormap can perform uniform quantization, useful as a basis for comparison with other quantization methods.

Adaptive quantization techniques, unlike uniform, attempt to somehow adapt the representative colors to the color distribution of the original image. Two important adaptive methods are the popularity algorithm and the median-cut algorithm [1]. The popularity algorithm scans through all the image pixels and picks the N most often occurring or "popular" colors to be the representatives for the image. If one imagines the colors charted in a histogram format by number of occurrences for each distinct color, then the popularity algorithm has the effect of choosing the colors from the densest regions of the distribution. Although this works well for many images, it functions poorly on images with a wide range of colors or when quantizing to a relatively small number of representatives because it ignores colors in sparse regions of the color space.
What is needed then is a method of considering all the colors in the original image. The "median-cut" algorithm, developed by Paul Heckbert at NYIT, performs well in this regard because it chooses representative colors which vary widely in adapting to the input color distribution [1]. Geometrically, the median-cut algorithm first maps pixels into 3-dimensional color space along the red, green, and blue axes. Each axis extends from 0 to 255, as that is the range of the corresponding rgb component in a color point. All of these color points are then enclosed in a "box", which is shrunk to insure a snug fit against the outermost contained points. The box is then cut in half at the median along its greatest dimension (hence the name of the algorithm), these two boxes are shrunk to tightly enclose their points, each of these boxes is further split into two approximately equal sized smaller boxes, and so on recursively until we have $2^n$ little boxes where $n$ is the number of splits. The color points inside these boxes are then averaged to give the desired $2^n$ representative colors, which all the original pixels will be mapped to.

Colormap uses the median-cut algorithm as part of the process of quantizing an image. The total process is logically separated into four phases: sampling the image for color statistics, choosing the color representatives (median-cut), mapping the pixels to their closest representatives, and finally quantizing and redrawing the image.

One final concept should be discussed here, and that is the image data format and the graphics terminals the images are targeted for. A frame buffer is the hardware memory attached to graphics terminals which holds the picture (frame) to be displayed. There are two types of frame buffers we wish to distinguish between: segregated and integrated. A segregated frame buffer is composed of three independent, segregated memories for the red, green, and blue components of the pixels. Each memory is usually about 8 bits deep. Thus, the composite values at each pixel make up a 24-bit color. An integrated frame buffer consists of a single memory array of pixels and a colormap. Each pixel is no longer the color itself but rather an index into a lookup table of color values - the colormap (hence the clever name of the program). On a Vectrix terminal, for example, each pixel is 9 bits so the colormap may hold up to 512 colors. With $n$ bits/pixel we can access any one of $2^n$ colors in the colormap, and these $2^n$ colors (24-bit rgb) make up the entire, quantized image. Since every pixel now points to its representative color, we can instantly alter the entire image just by changing the corresponding colors in the colormap.

In either case, whether the colors are directly or indirectly specified, the resolution of the frame buffer in $h \times w$ pixels must be at least that of the image, while the depth in bits determines the number of colors to quantize to. Colormap takes as input a 24-bit/pixel rgb image and quantizes it down to $n$ index bits/pixel (4..12, usually 8 or 9) with a $2^n$ color colormap intended for an integrated frame buffer. The image produced is in a generic data format, and an additional routine is used to encode it for the desired graphics terminal.
The remainder of this report is devoted to the relevant algorithms, data structures, computational issues, and perceptual factors involved. The implementation of Colormap is based on work done by Heckbert [1], following his suggestion to use a k-d tree for unifying the different phases of the quantization process. Colormap also differs in that the initial prequantization is to 18 bits/pixel (rather than 15) for greater color resolution. To compensate for the resulting increase in computation, phases III and IV are combined so that the nearest neighbor search is only performed once for each distinct color used in the quantized image. The mathematical formulation of the color image quantization problem is discussed in Heckbert. The motivation for using k-d trees and information on constructing, searching, and empirically analyzing them can be found in Friedman, Bentley, and Finkel [2]. An analysis of various data structures for range searching can be found in Bentley and Friedman [3]. Information on running the Colormap program is contained in the appendices.

2. THE QUANTIZATION PROCESS

The two main steps in quantizing a color image are to choose the "best" colors to be used, and to map all the different pixels to these colors. The total process is divided into three phases:

I. Sample the input image to determine its color distribution.
II. Determine the colormap based on this distribution.
III. Redraw (quantize) the image by mapping the original 24-bit pixel colors to their closest color representatives (nearest neighbors) in the colormap. Perform dithering, if requested.

The underlying data structure used to implement this is a K-D tree, or k-th dimensional binary search tree. A thorough explanation of k-d trees is presented in Friedman et al. [2], along with algorithms to build and search the trees and an analysis of their asymptotic complexity. The essential difference between a regular binary tree (1-dim.) and our k-dimensional k-d tree is that the partition value stored at each node is now for any one of the k dimensions (here 3 dimensions - red, green, and blue). For example, mapping a pixel to its nearest representative in the color table (phase III) is now equivalent to searching the k-d tree by comparing at each level in the tree the node’s split dimension and value to the relevant red, green, or blue component of the query pixel.
The three quantization phases may now be thought of in terms of k-d trees:

I. Sample the image to determine the color points to be indexed by the k-d tree.

II. Construct an "optimal" k-d tree based on these color points. The leaves of the tree point to the boxes which contain the color points. The points within each box are then averaged to determine the corresponding color representative for that box.

III. Quantize the image by simply looking up each pixel in a mapping to find the index of its color representative. If that particular pixel color has not yet been mapped, search the k-d tree for its nearest neighbor in the color table.

**Phase I: Sampling the Image**

In one pass over the input image, each pixel is prequantized to 18 bits by truncating the color components to 6 bits red, 6 bits green, and 6 bits blue, and the corresponding slot in a color bit-vector is set to TRUE. As discussed in the sections on Dissimilarity Measures and Future Considerations, no "pixel count" is kept because all colors are weighted the same regardless of how many times they occur in the image, thereby not favoring popular colors over the more sparse ones. The prequantization reduces the number of different colors to consider, and increases the frequency of use of the remaining ones. This is important in that memory usage and processing time is proportional to the number of distinct colors that must be considered. With 18 bits the number of distinct colors will be under 256K (262,144). Note that the image itself has not yet been altered, only sampled, and the color sampling only takes into account 18 of the 24 bits of each pixel. If we were using uniform quantization, we would merely "prequantize" down to the desired bits/pixel and be finished.

We will assume that the initial transformation from video signal to 24 bit/pixel image yields no perceptible quantization errors such as contouring, off-colors, and the like. Empirical data also suggests that the prequantization to 15 bits results in minimal image degradation if the colors in the colormap are chosen correctly [1], so the results for 18 bits should be as good if not better. This prequantized image will now be referred to as the "original" image. In what follows we let \( N_p \) = total number of pixels in the image, and \( N_c \) = number of distinct colors upon prequantization. Processing time for phase I is then \( O(N_p) \).
Phase II: Building the Color Table

What is now desired is to find a "best" subset of colors to represent the colors in the original image, according to some chosen dissimilarity measure D (discussed under Dissimilarity Measures). In the Colormap program $D(c_1, c_2)$ is the maximum coordinate distance function between the two color points $c_1$ and $c_2$ i.e. the absolute distance along the coordinate of greatest range. We minimize quantization error by reducing this distance. The resulting subset of colors - the color representatives - is stored in the color table. Quantizing to $n$ bits/pixel allows us to access up to $2^n$ distinct representatives.

All the color points determined in phase I are mapped into 3-dimensional color space along the red, green and blue axes. The median-cut algorithm is then used to choose representative colors which adapt greatly to these color points. By enclosing all the points in a box, shrinking the box to tightly enclose them, splitting it into two along the median of the greatest dimension, shrinking each one of those boxes and so on we cut in half the dimension of greatest quantization error each time [2].

The data structures built in this phase and used throughout the quantization process are illustrated on the following page.
Main data structures used by Colormap

**The K-D Tree**

- **Red**
- **Green**
- **Blue**

- 2^n leaves (2^n boxes)
- n levels (n colorbits)
- (No child pointers needed since it's a heap)

**Boxes & their Color Points**

- **Colorpoint[0]**
  - 0 1 2
  - 2^{18}_1

- **Colorpoint[1]**
  - 0 1 2
  - 2^{18}_1

- **Box**
  - start
  - count
  - a Color consists of 3 bytes (red, green, blue)
  - ColorTable(i) = representative color for Box i
  - TableSize = 2^n different colors possible with n colorbits

- **Bin**
  - 0 1 j 2^6_1
  - bin(j) = beginning of the thread of points whose rgb sorting component = j.
  - With 6 bits in a component, 64 different values are possible.

Each **Box** points to a contiguous sequence of color points.
This represents a geometric "box" in 3D color space.
Perhaps this requires some explanation! For starters, the motivation for these data structures is to provide both time and especially space efficiency in representing and manipulating the image data.

*Colorpoint* consists of two identical arrays of 256K elements each, for holding and sorting all colors possible by the prequantization to 18 bits. Having two arrays allows a fast, simple bin sort since we need not copy the color points into bin queues - just thread them in place and copy groups of points bin by bin to the alternate array. Each color point consists of a red, green, and blue byte and has a slot associated with it in the *next* vector.

The *next* vector is basically an array of indices which link related color points, although it serves different functions depending on which quantization phase we are in. When sampling the image, these slots are used as boolean elements of a bit vector indicating whether or not the color point represented by that index occurs in the image. When partitioning color space into finer and finer boxes, the *next* fields are used in conjunction with the *bin* array to implement an efficient O(n) bin sort. When mapping pixels to their nearest representatives these slots contain the indices of the representatives in the colormap.

Each record in the *Box* array represents one of the current boxes in rgb color space. It consists of a starting index and a count of the number of points in the box. As color points in a box are stored in contiguous memory locations, all we need to know is the box index number to have access to all the points contained therein.

To split a box at its median we do the following. The longest dimension of the box is determined in one pass through its points by finding the maximum and minimum components along each of the red, green, and blue axes. The points in the box are then sorted along this dimension. (The median is simply the middle element in the sorted sequence.) Splitting the box at its median is equivalent to dividing the sequence into two at this middle element: we add a corresponding record to the *Box* array to represent the latter half of the sequence, and adjust the count of the original *Box* record to represent only the first half of the sequence. The "new" boxes each contain about half the points of the parent and are mutually exclusive of each other.

To implement an efficient O(count) bin sort, the *bin* array and *next* vector are used to thread together (in one pass) points with identical components along the sorting dimension. Since each color component has been prequantized to 6 bits, an array of 64 bins will suffice. Once the points in the box are threaded, they are "sewn" (copied) bin by bin into the other *Colorpoint* "run". The points in the box are now sorted along the desired dimension. In this manner each time the color space is recursively partitioned into twice as many boxes,
the color points physically alternate between the two Colorpoint runs.

Thus we have a straightforward O(Nc) algorithm for determining the median of an array and rearranging the elements around it. Other time-efficient methods considered either required too much thought of the author [8], or only gave approximations of the median [7] and further processing was required to rearrange the elements.

How does the k-d tree fit in to the picture (no pun intended)? In essence the k-d tree is a history of the partitioning of color space, and the leaf nodes represent the current configuration. Each node in the tree contains a dimension key and a partition value along that dimension to record how a box is split. When a box is split in two, the split dimension ('RED', 'GREEN', or 'BLUE') and its value (the median) are stored as the node's key and partition value, respectively. These two smaller boxes are then the roots of the node's left and right subtrees. If the node is a leaf (no more splitting the box), the key equals 'LEAF' and the partition value contains the index of the corresponding Box record. Since every internal node has 2 sons, the k-d tree is a complete binary tree and child pointers can be eliminated via a heap representation. Friedman et al. refers to this as an optimized k-d tree, where the discriminating key is the dimension of greatest range, and the partition value is the median [2]. The bucket size or number of points per box is determined indirectly by the user through the number of colorbits specified:

\[
\text{number of points in a box} = \frac{(\text{number of color points } N_c)}{(2^{\text{colorbits}})}
\]

So, for example, with 9 colorbits and \(N_c = \) a maximum \(2^{18}\) colors, we have at most 512 pixels per box.

Specifying \(n\) bits per pixel means we recursively partition color space \(n\) times, resulting in \(2^n\) boxes and \(n\) levels in the k-d tree. The points within each box are then averaged to give that box's color representative, stored in the ColorTable slot of the same index as the Box record. Every box therefore has an associated color representative which represents an equal number of distinct pixels in the original image. This representative will be a full 24 bits long, but only 18 bits are significant as that was the degree of accuracy of the sampling in phase I. At each successive level in the tree, the color points are moved to alternating runs during the bin sort. Thus, the current and final runs are completely determined by whatever level we are at in the tree.

It may be apparent at this point that the k-d tree is not really needed to partition up color space and determine the color representatives. However, in phase III the tree becomes a very efficient and convenient mechanism for mapping pixel colors to their nearest representatives in the color table. Given a color point, we can search down through the tree
and find the point's representative color in $O(\log N_c)$ time. Additionally, the overhead required to build the tree is minimal since the same processing must be performed to split up color space regardless.

Let us, for fun, analyze the time and storage complexity required in phase II for the k-d tree. At each level in the tree we must scan through and sort all points in every box, for $O(N_c)$ time. Since there are at most $\log N_c$ levels, total asymptotic time to build the tree is $O(N_c \log N_c)$. In practice the number of boxes $2^{n_l}$ is determined by the number of colorbits $n$, so construction time is bounded by $O(n N_c)$ where $n$ is usually around 8 or 9 (versus 18). This corresponds to the fact that each leaf represents a box containing many color points, not just one, so fewer tree levels need to be built. Also, on many images the colors are not distributed across the entire spectrum, so $N_c$ will often be $\cdot 2^{18}$. Hence, simpler images are processed faster.

The storage required for the k-d tree is $O($number of boxes$)$. Since every leaf corresponds to a box, there are $B$ leaf nodes where $B$ is the number of boxes. Then there are $B-1$ internal nodes, for a total of $2B-1$ nodes. As $B = 2^n$ (or $N_c$ if there are fewer colors), storage for the k-d tree is $O(2^{n+1})$.

**Phase III: Quantizing The Image** (Mapping pixels to the color table)

In this final phase we map all the pixels in the image to their closest representatives in the color table. This effectively quantizes the image, our goal.

To speed things up, after finding the nearest representative for a pixel we store its color table index in some kind of mapping. This way, future occurrences of that same pixel need not entail another search. The only cost of this is the memory required to hold a mapping of $2^{18}$ possible pixel colors, a cost we avoid by using the (suspiciously) convenient next vector to hold these indices. Our upper bound for the number of searches is then the number of distinct colors in the image, $N_c$, where $N_c \leq N_p$ depending on the resolution and color composition of the image. Total processing time for phase III is $O(N_p$ pixels $+ N_c$ searches $\cdot$ cost/search).

The height x width array of pixel indices and $2^{n_l}$ entry x 24-bit color table represent the image quantized to $n$ bits per pixel, and will be passed to the integrated frame buffer for display.
The Nearest Neighbor Search

How do we actually search the k-d tree for a pixel’s closest color representative (nearest neighbor) in the colormap, and what is the cost? Basically, we search the k-d tree according to the red, green, and blue components of the pixel. At each successive level in the tree, the new node’s partition value is compared to the pixel component corresponding to the dimension of split, and the appropriate subtree is recursively searched. It is useful to think of the accumulation of these partitions along the red, green, and blue axes as geometrically forming the current “box”. Upon reaching a leaf, we know which box in color space contains the pixel.

But merely using the color representative of the box containing the search color point is insufficient, as there may be closer representatives in other boxes! Geometrically the problem can be conceptualized as a ball (sphere) centered at the query point, with radius equal to the dissimilarity between the query point and the closest color representative found so far. If this ball is totally enclosed by the current box, then no other color representative can possibly be closer, and we are done. However, if the ball does not fit within the current box there may be a color representative in another box whose dissimilarity from the query point is smaller. The Ball Within Box test [2] determines the dissimilarity from the query point to the closest side of the current box and compares it to the ball’s radius. If the radius is greater, the Ball Within Box test fails and we must backtrack through the tree and search neighboring boxes.

At any given node during the search we keep a record of the geometric boundaries delimited by that node. When searching a child node these boundaries are adjusted accordingly. If the Ball Within Box test fails for the closer child, we then test the further child before searching it to see if it can possibly contain a nearer color representative. This Box Overlaps Ball test is done by calculating the distance to the closest side of that box. If this distance is less than the dissimilarity between the query color point and its nearest color representative found so far, then the farther child may contain a closer representative and it is searched. When that is done the Ball Within Box test is again performed to see if we must back up still further in the tree.

Of course, the brute force way to do this ("Rambo" method) would be to search all the color representatives in the table each time and then to choose the one with the smallest dissimilarity from the query color point. This will work, but it is $O(2^N)$ representatives per search, for $O(N_c2^N) = O(N_c^2)$ total. Friedman et al. has shown that for optimized k-d trees the amount of backtracking is independent of file size $N_c$, so that at $O(logN_c)$ per search the total pixel search time would be $O(N_clogN_c)$. This is much better than exhaustive search, particularly since $N_c$ may be on the order of $2^{18} = 256K$ colors.
**Dissimilarity Measures** (The shortest distance between two points)

An important concept largely ignored thus far has been the actual dissimilarity measure employed - what it means for one 3-dimensional color point to be "near" another. Common mathematical metric distances to be considered are of the form [2]:

\[
D_p(x_1, x_2) = \left[ \sum_{i=1}^{k} |x_1(i) - x_2(i)|^p \right]^{1/p} \quad [\text{vector space p-norms}]
\]

Of these, the most common are

- \( p = 1 \): taxicab or city block distance
- \( p = 2 \): Euclidean distance
- \( p = \infty \): maximum coordinate distance = \( \max |x_1(i) - x_2(i)| \) for \( 1 \leq i \leq k \)

The \( p = \infty \) maximum coordinate distance function is used in Colormap, based on perceptual factors and ease of implementation, while the \( p=2 \) euclidean distance metric was used in Heckbert's implementation. Ideally we would like a *perceptually* based quantizer to determine the partitioning of color space, resulting color representatives, and nearest neighbors chosen. The maximum coordinate distance function approximates this by selecting the color coordinate of maximum dissimilarity. Reducing distance along this dimension gains the most color resolution, even though the resulting colors may be slightly off from the original. It is these relative color differences between pixels rather than the absolute color differences between images that we are more likely to perceive. While accounting for multiple occurrences of the same color point might make the color representatives closer to the original colors, this tends towards the popularity algorithm in that it doesn't give much weight to colors in sparse areas of the color space, decreasing color resolution in the quantized image.

The maximum coordinate distance function is also straightforward to implement. Calculating \( D_{p=\infty}(x_1, x_2) \) entails choosing the maximum linear distance between \( x_1 \) and \( x_2 \) along the three coordinate axes. Since \( D_{p=\infty} \leq D_{p=2} \) (euclidean distance), a ball with radius determined by this dissimilarity measure will in general be smaller than a ball with radius determined by euclidean distance. This results in fewer boxes searched. Tests conducted by Friedman et al. confirm this for database applications, and similar results are expected for the analogous nearest neighbor search in color space [2].

The section on Future Considerations discusses other possible dissimilarity measures.
Dithering

When quantizing to only a few representative colors or with a poorly chosen colormap, contouring (color banding) in the resulting image can be quite noticeable. Such images can often be improved by dithering. Dithering attempts to compensate for the quantization error introduced at a pixel by trading its intensity resolution for neighboring spatial resolution. In the Floyd-Steinberg dithering method the quantization error at the current pixel is propagated to neighboring pixels below and to its right (see [1], [5], and [6]). This allows us to quantize and dither the image in one top to bottom pass, with 3/8 of the quantization error distributed to the right pixel, 3/8 to the lower pixel, and 1/4 to the lower right pixel:

\[ e_i = (r, g, b) \text{ quantization error at pixel } p_{i,j} \]
\[ e_i = [\text{original pixel color}]_{i,j} - [\text{color representative}]_{i,j} \]

Since dithering alters the colors of successive pixels in the image, it is now possible that there are new colors which were not in the original image. In this case we have additional pixel colors to perform nearest neighbor searches on \( N_C \) is effectively increased. Dithering will therefore somewhat lengthen the search time required in phase III by that required to search for these additional colors. Conversely, depending on the color distribution of the image dithering may actually reduce the number of distinct colors in the quantized image. This would result in fewer searches. In any event, there is still the added \( O(10 N_p) \) time to propagate pixel error. In practice this along with the additional nearest neighbor searching approximately doubles the total processing time.

Also note that even if every box contains exactly one color point (the ideal partitioning of color space), the representative colors in the color table may not be identical to the original colors of the image, because the sampling in phase I was only significant to 18 bits/pixel. Dithering against the original 24 bit/pixel image will therefore always have some effect, and may in some cases even hide contouring in the original image.
3. COMPLEXITY ANALYSIS

**Key:** \( n = \) number of bits/pixel to quantize to (usually about 8 or 9).

Then \( 2^n \) is the number of boxes and hence the size of the color table.

- \( N_p = \) number of pixels in the image (eg. 322,560 for a 480 x 672 screen).
- \( N_C = \) number of colors in the image after prequantization (max \( 2^{18} = 262,144 \)).

The asymptotic time complexity of phase I is bounded by the pixel sampling, for \( O(N_p) \). Constructing the k-d tree is incorporated into the cost of building the color table, which calculated to its full extent is \( O(N_C \log N_C) \) to index \( N_C \) color points. However, the tree is not actually built to that fine an indexing, only to the first \( n \) levels to give \( 2^n \) boxes. Thus the time to build the k-d tree is \( O(n N_C) \). Mapping pixels to their nearest representatives is proportional to \( \log N_C \) time per k-d tree search, for \( O(N_C \log N_C) \). The time spent translating a pixel color to a color table index is constant, so in addition to the searches performed phase III also requires \( O(1) \) time per pixel. Total processing time to quantize an image is therefore \( O(N_p) + O(n N_C) + O(N_p + N_C \log N_C) = O(2N_p + nN_C + N_C \log N_C) = O(\max(N_p, N_C \log N_C)) \). Using typical values for \( n \) and \( N_p \), this means that the time to quantize an image is \( O(N \log N) \) with the number of (prequantized) colors in the image - a respectable speed considering the sorting and searching involved. Images with a narrow color distribution have an even faster processing time, proportional with the number of pixels (the best we could hope for). If dithering is performed, the constant factor here is somewhat increased due to the additional computation at each pixel but the asymptotic running time remains the same.

In analyzing the space complexity of the given data structures, we ignore the minimum \( O(N_p) \) memory necessary to hold the image. The two most important structures in respect to size are the Colorpoint / next arrays and the k-d tree heap. Since the Colorpoint runs and next vector are used to hold and manipulate the points, they require \( O(N_C) \) space. The Box array and ColorTable each require \( O(\text{number of boxes} \times B = 2^n) \) space, while the k-d tree heap uses \( O(2B) \) space. Total space complexity is therefore \( O(N_p) + O(2^n) + O(N_C) = O(N_p) \), linear with the number of pixels.

4. FUTURE CONSIDERATIONS on improving image reproduction

"The goal of computer science is to build something that will last at least until we've finished building it."

Given a fixed number of colors to quantize to, the dissimilarity measure is the most important factor in determining the faithfulness of reproduction in the resulting image.
Future improvements to Colormap's image quantization may benefit from the following considerations:

1. Alter the dissimilarity measure

   • Use a perceptual-based quantizer (see references in [1])
   • Try euclidean distance in rgb space (p = 2 metric)
   • Split by minimizing variance rather than at the median
   • Method of least squares on a local level

2. Weighting colors by number of occurrences of that pixel in the image.

   In Colormap, each pixel is weighted the same even if one occurs a thousand times more than another. If weighting is desired, a pixel count would need to be associated with each distinct color. This weighting of pixels could then be considered when splitting a box and/or averaging color points within a box to obtain the color representative.

3. Apply unused boxes to split the largest remaining boxes in color space.

   When building the k-d tree and recursively subdividing color space, a box is not split further once it is shrunk to containing only one color point. This results in "unused" boxes which Colormap does not currently take advantage of, since boxes are allocated by level in the tree, not free boxes available. To utilize these boxes, one could associate two extra fields with each box: size (by some dissimilarity measure) and leaf_node (pointing to box's leaf in k-d tree). Now, after the k-d tree is built but before the color representatives are determined, we would iteratively pull the next largest box off a priority queue of the current m largest boxes, split it, and reinsert it back on the queue for as many extra boxes as we have available.

   However, since all the boxes contain an approximately equal number of points (by nature of the median split), the fact that a given box contains only a single color point indicates the other boxes contain at most a few points. It has yet to be determined whether the increased resolution gained by splitting these boxes is worth the additional cost involved.

4. Alter dither error propagation ratios.

   Perhaps modifying the pixel error propagation ratios to neighboring pixels (presently 3/8 right, 3/8 down, and 1/4 lower right) might increase the effectiveness
of dithering.

To assist in the evaluation of future quantizers, Colormap can display various color space statistics via the -x option. These include the following:

- the number of distinct colors \( N_c \) and how often each occurs in the original image
- the number of boxes in color space \( B \) and average number of points contained in each
- the average number of pixels each color representative is mapped to
- the color table itself

5. INITIAL RESULTS

Using the algorithms described, Colormap can quantize a 480 x 672 x 24-bit image to 512 colors in under one minute on a Vax 11/780, and under 2 minutes with dithering. If uniform quantization is used, the respective quantization times are about 20 seconds faster.

Thus far, the results have been as expected. For 512 colors (9 bits/pixel), the uniform method produces contouring which is quite noticeable in the quantized images, while adaptive quantization via the median-cut algorithm produces very faithful replications with virtually no visible contouring. Dithering is a nice option for smoothing out excessive banding, and results in a particularly noticeable improvement for uniformly quantized images.

In general, it is expected that the median-cut algorithm will perform consistently better than uniform quantization on all types of images. It would be interesting to see an extensive comparison done on the relationship between the quantization methods, bits/pixel, processing time involved, and resulting image quality.

ACKNOWLEDGMENTS

The author wishes to thank Dean Krafft for his introduction to color image quantization and for his direction on a senior project which entailed a bit more work than this student was excited about. Special thanks also go out to Rod Recker for offering his unique perspective on the project ("Bunk dude", "out of control"), Tom Kelly for his creative typing assistance and awful quantization jokes, and Debbie Mathews for her suggestion to use verbs to connect subjects and predicates in a sentence.
REFERENCES


Appendix A: How to run the program

Invocation:

colormap [-uxc] <imagefile> <# colorbits>

Quantizes the input rgb imagefile from 24 bits/pixel down to n colorbits per pixel using up to $2^n$ colors. Outputs resulting pixel index and color table to the file imagefile.map in a generic format intended for an integrated frame buffer.

Options:

-u Uniform quantization. Calculates the color table by uniformly truncating each of the three rgb color components of a pixel. The resulting picture quality will usually be poor since the color table entries are calculated independently of the actual color distribution of the image. Mostly for comparison and debugging purposes; default quantization method is the median-cut algorithm.

-d Dither image. Dithering attempts to compensate for the quantization error introduced at a pixel by distributing it to neighboring pixels. Works best for a poorly chosen colormap (i.e., by uniform quantization), or when quantizing to only a few colors.

-x Display status messages and color space statistics. Useful for debugging information and evaluating the quantization.

Input file: imagefile

Output file: imagefile.map

| Header | height, width must be preset  
| TableSize must be zero  
| Screen Array | Color ScreenArray[height][width]  
|             | (24 bit/pixel image)  
| PixelIndex | colors_used is set by program  
| TableSize = 2^colorbits  
| ColorTable | short PixelIndex[height][width]  
|             | (indices into the ColorTable)  
|             | Color ColorTable[TableSize]  
|             | (colors in the quantized image) |
The first \textit{colors\_used} slots in the \textit{ColorTable} contain valid colors; the remaining \textit{[TableSize - colors\_used]} slots are set to black (0,0,0). See appendix B for relevant data structures \textit{Header} and \textit{Color}.

The \textit{Header} determines the status of imagefile:

- TableSize = 0 on input: file contains rgb ScreenArray.
- TableSize > 0 on output: file contains PixelIndex and ColorTable.
- TableSize = -1 if error.

On an input or processing error, the colormap program prints an appropriate message and halts. The output file is not created until the quantized image is finished and about to be written to disk. If an I/O error occurs at this point, the write is halted (but TableSize is not currently set to -1 due to implementation difficulties).
Appendix B: Implementation details

The Quantization Process

Program Configuration

colormap.h - Global constants and data structures. Constants usually represent parameter boundary conditions, and can be modified somewhat.

ex: MINBITS ≤ n ≤ MAXBITS /* number of bits allowed per pixel */

The relevant data structures Header and Color of the file interface (app. A) are the following:

typedef struct {
    /* Color components of a pixel */
    unsigned char red, green, blue;
} Color;

typedef struct {
    /* File header */
    short height, width; /* of graphics screen */
    char unused[4];
    short TableSize; /* of resulting ColorTable */
    short colors_used; /* will be ≤ TableSize */
    char reserved[4];
    char pad[112]; /* to make 128 bytes */
} Header;
In uniform quantization, if the number of bits to quantize to is not a multiple of three, red gets the first extra bit and green gets the next. For example, with 10 colorbits the rgb allocation will be (4,3,3), while with 11 bits it will be (4,4,3). For a fixed number of bits, this will affect the quality of quantization depending on the dominant color component(s) in the original image. In median-cut, a tie for greatest dimension also favors red, then green, then blue.

It may be desirable to alter these for perceptual reasons.

**Machine Dependencies & Memory Usage**

An integer must contain at least 19 bits, to represent positive array indices $0..2^{18}-1$ and NIL (-1). Although MAXBITS is currently set to 12 bits/pixel, it can be increased to allow the user up to 15 bits/pixel.

Of concern are the large data structures and arrays, which must be declared up to their maximum possible sizes in relation to the program constants mentioned above. The memory requirements are the following:

- **ScreenArray:** $512 \times 672 \times 4$ bytes each (aligned) = 1.3 M
- **Colorpoint:** $256K \times 2 \times 4$ bytes = 2 M
- **next:** $256K \times 4$ bytes = 1 M
- **Box:** $2^{12} \times 8$ bytes = 32 K
- **ColorTable:** $2^{12} \times 4$ bytes = 16 K
- **K-D Tree:** $2 (B = 2^{12}) \times 4$ bytes = 32 K

The total run-time memory requirement is about 4.3 MB. PixelIndex is mapped onto ScreenArray, saving about 600K.
Appendix C: Relevant Terms

*adaptive partitioning* - the partitioning of color space adapts to the input color distribution.

*box, hyperrectangle* - k-dimensional abstraction of the basic rectangle (here k = 3). Every (final) box in color space contains \( \frac{N_C}{2^{\text{colorbits}}} \) points, and has an associated color representative in the color table.

*colormap, color table* - the \( 2^{\text{colorbits}} \times 24 \)-bit table containing the color representatives for the image. This is the "best" subset of colors as determined by the median-cut algorithm.

*component* - the individual red, green, or blue values in a color point coordinate. With 6 bits a component can have 64 distinct values. The longest dimension of a box is equivalent to the component of greatest range.

*coordinate* - the \((r, g, b)\) triple which represents a point in color space. An 18-bit (prequantized) triple will have 6 bits in each component. With 18 bits there are 256K different color points possible.

*dithering* - averaging intensities of neighboring pixels to reduce visual contouring.

*frame buffer* - hardware memory buffer connected to the graphics screen which holds the image being displayed. It is this limitation on the bit capacity per pixel that necessitates color image quantization.

*k-d tree* - kth dimensional binary search tree: partition at each node is along one of the k dimensions (here red, green, and blue).

*nearest neighbor, range searching* - search for the closest representative color in color space.

*quantization* - color by numbers.

*rgb* - red, green, and blue. Refers to the components of a pixel and the corresponding phosphors in a graphics screen.