A Real-Time Scalable Color Quantizer Trainer/Encoder

Navin Chaddha, Wee-Chiew Tan and Teresa H.Y. Meng
Computer Systems Laboratory
Stanford University, Stanford, CA 94305.

Abstract

In many color-imaging applications, it is desirable to display an image with as few different colors as possible with minimal loss in image quality. While good image quality is achievable using traditional Vector Quantization techniques, they are too slow for real-time video applications. An architectural design of a real-time, scalable color quantizer architecture is presented. It implements our fast Tree Structure Vector Quantization algorithm with a variable-size cubical prequantizer based on human perception proposed earlier. The design is scalable and uses different configurations of the processing and memory elements to process any 24-bit to 72- bit colors per input pixel to produce a 8-bit to 24- bit color palette.

1. Introduction

With continuing advances in multimedia, computer users are demanding better image quality than 24-bit resolution. The need to provide a true 24-bit color display is a strain on the cost and technological feasibility, particularly because of the large frame buffer needed to support the color resolution. For example, a 1280×1024 pixel, 24-bit frame buffer with a refresh rate of 72 Hz requires 3.9 Mbytes of memory, and a memory access time of 3.5 nsec/byte. On the other hand, a 8-bit frame buffer only requires 1.3 Mbytes of memory, and a memory access time of 10.6 nsec/byte. The cost of high-speed memory needed to support a full-color display on a high-resolution monitor makes true 24-bit to 72-bit color impractical for many applications and in fact unnecessary. With 24-bits there are a total of approximately sixteen million possible colors but only a small subset of these colors will appear in a particular image. This offers the opportunity for *color quantization*, which maps the colors to indices of a lookup table called the *color palette*. The mapped colors are decoded upon display using color palette by video palette DACs [1],[2]. As the image resolution and color resolution increases, even 24-bit color displays will be ill-equipped to display truly high quality images, such as those from the 48-bit Kodak photo CD system, which still has to be color quantized to be displayed on 24-bit color displays.

A number of approaches [4] based on vector quantization (*VQ*) have been suggested for the design of color palettes. The problem with these algorithms is that they are computationally intensive whose execution times are in the order of minutes to hour per image. and the performance is sensitive to the choice of initial color palette. Digital half-toning techniques such as ordered dither and error diffusion have been incorporated [5]. These techniques, however, sacrifice spatial resolution for tonal resolution, and have visible texture artifacts.

In this paper we use an algorithm which we had proposed earlier for color palette design that achieves both computational efficiency and good image quality with minimal artifacts [6], [7]. We perform the quantization in two stages. In the first stage we pass the image through a pre-quantizer which limits the number of initial colors. In the second stage we perform VQ to further reduce the number of colors obtained from pre-quantization to the desired number of colors in the palette. We emphasize the use of color palettes with a tree structure as it reduces the computation required both to design a palette and to quantize colors using a palette. This allows for efficient pixel mapping with search time proportional to the tree depth. A subjective distortion measure to reflect masking effect is used in both the pre-quantization stage and the VQ stage. We have found that in software the average CPU time of our color palette design algorithm [7] is 0.5 s for a 352 x 240 pixel image. The algorithm achieves an average PSNR of over 42 dB. for the different images.

While the software implementation of our algorithm is very fast, it is still not fast enough for real-time video applications, which requires a rate of 30 frames/sec or more. This paper describes the hardware mapping of our algorithm to meet this speed requirement. An important feature of this architecture is its *scalability*--different input and output resolutions in color quantization can be realized without re-designing the processors in the architecture.

This paper is organized as follows. Section 2 gives an overview of the algorithm for color palette design. Section 3 describes the architecture of the color quantizer. Section 4 discusses the architecture of the pre-quantizer processor. Section 5 presents the architecture of the TSVQ processors. Section 6 discusses the use of the color quantizer trainer as an encoder. Section 7 gives the computational requirements and simulation results. Section 8 gives the conclusion.

2. Color Palette Design Algorithm

In this section we briefly describe our color palette design algorithm which consist of two stages: a prequantization stage to efficiently reduce the number of colors to be quantized by a few orders of magnitude, then followed by a tree structured vector quantizer (*TSVQ*) quantizer using these prequantized color vectors as its training set.

It has been known empirically that the best color palettes are not necessarily the ones that minimize the squared error because the subjective error resulting from quantizing a pixel's color value cannot be treated independently of the surrounding pixels. It has been also known [8] that the human observer is more sensitive to quantization errors in the smooth or low activity regions of an image as compared to quantization errors in the high activity regions. We use (sample standard deviation) as the activity function [7] which sums the absolute devi-

1058-6393/95 $4.00 © 1995 IEEE

ation of a color vector from the mean of the color vectors in an 8x8 block of pixels. The image is divided in 8×8 pixel blocks from which the activity is computed as:

$$\underline{a} = \frac{1}{64}\sum_{p=1}^{64}\left\|\underline{x}_p - \bar{\underline{x}}\right\| \quad , \qquad (1)$$

where $\underline{a}$ is activity vector (a_1, a_2, a_3) of color components, and $\underline{x}_p$ is color of pixel p in a block, $\bar{\underline{x}}$ is the mean color of that block. The activity function of each block, A, is a *weighted* L_2 *norm* of the elements in activity vector:

$$A = \sqrt{w_1a_1^2 + w_2a_2^2 + w_3a_3^2} \quad . \qquad (2)$$

The different weights are chosen to reflect the visual importance of each color component, e.g. (w_1, w_2, w_3)=(1,1,1) for RGB and (w_1, w_2, w_3)=(4,1,1) for YUV. We use a weighted squared error measure for the distortion measure which takes into account local masking effects [7]. The weighting function used is the inverse square of A.

We perform the color quantization in two stages. In the first stage we use a pre-quantizer that divides the tristimulus color space (can be either RGB, YUV, etc.), into voxels of three different sizes according to the *activity function*. The color vectors are categorized as belonging to one of the three regions, namely, the low activity region, the medium activity region and the high activity region by comparing their activity functions to experimentally determined thresholds for low (A_{low}) and high (A_{high}) activity regions. The centroid of each cell is used to represent all colors mapped into that cell. The prequantizer uses the activity function to partition the color space into different size cells or voxels by the following algorithm:

Algorithm 1: Variable-size prequantizer

Step 1: If ($A < A_{low}$) then color vector is assigned to a small voxel (Low activity).

Step 2: If ($A_{low} \le A \le A_{high}$) then color vector is assigned to a medium voxel (Medium activity).

Step 3: If ($A > A_{high}$) then color vector is assigned to a large voxel (High activity).

Step 4: Quantize all color vectors within a voxel to its centroid.

The prequantization stage drastically reduces the number of elements in the training set, without a perceptual loss in visual quality. In the second stage we use a greedy unbalanced tree structure vector quantization algorithm [7] and use it for designing the color palette. The color vectors obtained from the pre-quantization stage are used as the initial training set of vectors. Instead of using the mean square error as our distortion measure we use the weighted mean square error. The advantage of the weighting is that there will be more codewords available to code the high distortion regions which in our case correspond to the low-activity regions. It is desirable to have more codewords for low-activity regions as the distortion artifacts are easily visible in these regions compared to the high-activity regions.

Each stage of TSVQ is performed using the splitting method of the generalized Lloyd algorithm (GLA) [3]. A binary tree structure is implemented, where at each stage of growth, the node with the largest *weighted mean-square error* (*Wmse*) is split into two children nodes. The initial centroid of the left node is the same as the parent node, while the right's centroid is a small random offset from the left. Then the GLA iteration, consisting of finding the closer node for each child oct and then updating the centroids of each node based on the revised list of children octs, is performed until a tolerance criterion is reached. The tree is "grown" until the desired number of colors (corresponding to the number of leaf nodes) are obtained.

Special data structures are used for efficient mapping of the pixels from the image space to the codewords (Figure 1). In the prequantization stage, the pixels are mapped from image space into different voxels of the color space. The variable-size voxels are efficiently represented by an *octtree* structure. An octtree is made up of octs, each oct (representing a voxel) can be subdivided in 8 smaller children octs, each $^1/_8$ the size of its parent. Hence the top level oct (corresponding to a large voxel) can have up to 8 medium voxels or 64 small voxels or a mixture of medium and small voxels. Upon TSVQ, the octs are mapped onto the codewords, and these pointers are maintained after training. This structure supports a direct encoder--pixels are mapped directly from image space to codewords through their respective octs without performing distance comparison to locate the nearest codeword. The binary tree structure of the codewords has an added advantage of acting as a search tree for adaptive quantization of pixels not mapped by the initial prequantization, or for successive frames in a movie sequence [7].

3. Architecture of the Color Quantizer

Our algorithm is mapped onto three kinds of processors: prequantizer, TSVQ_A and TSVQ_B processors. There are three factors involved in a scalable color quantizer design: the resolution of the input pixels (number of input color bits per pixel), the size of the voxels (number of octs), and the size of the final color palette (number of output color bits). The weak relationship among these factors is addressed by offering three degrees of configuration freedom in the color quantizer architecture as shown in Figure 2. There is one or more prequantizer processors, one or more banks of oct memory and TSVQ_A processors, and one or more banks of codeword memory. The expected number of octs directly influence the number of banks of oct memory, as each bank can store up to 4K octs. The number of codeword memory needed is a function of the color palette size. The invariant components in the architecture are the single TSVQ_B processor and an overall controller.

Three kinds of data structures are needed--preoct, oct and color-word. A preoct is a preliminary version of an oct, and is produced by the prequantizer. It has three 24-bit integers representing the centroid color components of the voxel, and a 50-bit inverse distortion value. After prequantization, the preocts are immediately replaced by their floating-point counterparts--the octs. All floating-point words in the octs are 24-bits, divided into a 16-bit mantissa and an 8-bit exponent. Both the preocts and octs are stored in the oct memory. Each color-word is 32 bits wide, with a 8-bit index and 8-bit color components. A codeword can be made up of one or more color-words,

depending on the size of output palette, e.g., a 8-bit palette is one colorword wide, while a 24-bit palette requires three colorwords in parallel to represent one codeword.

Each pre-quantizer processor handles 24-bits (8 bits in each color component) of input color resolution. Its function is to prequantize pixels in each block based on its activity function, and to create octs. The number of prequantizer processors required depends on the input resolution: 24-bit pixels will require only one processor, while 72-bit pixels will require three processors. The prequantizer processors work in parallel, and communicate carry bits between the prequantizer at the LSB location and the prequantizer at the MSB location.

On the other hand, the number of TSVQ_A processors equals the number of banks of oct memory, as each processor is dedicated to its own memory bank. This parallelism is necessary to maintain a constant throughput in TSVQ for increasing number of octs. The GLA iteration in the TSVQ stage can be separated into two types of operations: oct operations and codeword operations. This separation is partitioned onto two different processors: TSVQ_A which performs the oct operations, and TSVQ_B which performs the codeword operations. Since there can be many octs per codeword, the number of TSVQ_A processors is much larger than TSVQ_B processors. Hence there is only one TSVQ_B processor in the architecture, but there is one TSVQ_A processor for every bank of oct memory.

The arithmetic operations on the prequantizer processor are integer/fixed point, while floating point operations are used for the TSVQ processors, in order to support the wide range of values needed on multiplication and accumulation operations. The floating point precision in the TSVQ stage also allows for easy conversion from any input to output resolution.

4. Prequantizer Processor

For high throughput, a pipeline design is adopted (Figure 3). Each color component of the activity function is processed by an activity generator block. As the pixels of each block are read in, the sum is accumulated and divided (by a right shift) to produce the mean. The pixels are also delayed by 64 samples through a FIFO, so pixels can be subtracted from the mean as it becomes ready to obtain the activity component. The delayed pixels are hashed to index into a lookup table to obtain the preoct. To conserve space, preocts are not created until a pixel in the image space maps into it. If the preoct is not found, then it is created and updated in the lookup table. If a smaller voxel is needed, the current preoct is split into children preocts corresponding to that voxel size.

For faster implementation, the activity function is not computed directly, as it involves computing a square root. We observed that the subjective distortion function d used in calculating the *Wmse* during TSVQ is just the inverse square of the activity function associated [7] with each oct. Hence the inverse distortion d^{-1} can be used instead in algorithm 1 by inverting the thresholds and changing the comparison signs. This operations saves a square root and then a square per distortion function. The minimum inverse distortion, corresponding to the color with the lowest activity function for that voxel, is stored with preocts.

5. TSVQ Processors

Various optimizations are used to speedup the hardware implementation of TSVQ with the following observations. The centroid $\underline{c}$ of a Voronoi region (corresponding to a codeword) is the sum of weighted vectors divided by its total weight:

$$\underline{c} = \frac{\sum d_j x_j}{\sum d_j} = \frac{\underline{\mu}}{\delta} \quad , \qquad (3)$$

where the vector $\underline{\mu} = (\mu_1, \mu_2, \mu_3) = \sum d_j \underline{x}_j$, scalar $\delta = \sum d_j$, j denotes the number of vectors and i will denote the three components of each vector (e.g. RGB, YUV etc.). Instead of storing color vector $\underline{x}_j$ in octs, the weighted vector $d_j \underline{x}_j$ is stored instead. Also stored in the oct are its distortion function d_j and the weighted vector square $d_j x_j^2$, which are used to determine the *Wmse* (weighted mean square error).

To figure out which children codeword an oct belongs to, distance from the centroid of the oct $\underline{x}$ to the centroid of each codeword $(\underline{c}_l, \underline{c}_r)$, where $\underline{c}_l$ and $\underline{c}_r$ are the right child and left child codewords respectively, is determined through weighted distance function $Dist(\underline{x}, \underline{c})$. If the following inequality:

$$Dist(\underline{x}, \underline{c}_l) \geq Dist(\underline{x}, \underline{c}_r) \qquad (4)$$

is satisfied, then the oct belongs to the right child, and vice versa. Expanding (5), we obtain:

$$(\underline{x} - \underline{c}_l)^T D(\underline{x} - \underline{c}_l) - (\underline{x} - \underline{c}_r)^T D(\underline{x} - \underline{c}_r) \geq 0, \qquad (5)$$

where T refers to the transpose operation and since the distortion matrix $D = dI$ for our algorithm, the distortion cancels out and is removed from consideration. Simplifying (6), we obtain the result as,

$$(\underline{c}_l^T \underline{c}_l - \underline{c}_r^T \underline{c}_r) - 2(\underline{c}_l - \underline{c}_r)^T \underline{x} \geq 0 = \alpha - \sum_{i=1}^{3} \beta_i x_i \geq 0 . \qquad (6)$$

The expression (7) are used by all TSVQ_A processors in GLA iterations. Since the scalar α and $\underline{\beta}$ are constants during each iteration, they are precomputed by the TSVQ_B processor upon updating left and right codewords.

Determining which codeword to split in the GLA iteration is based on the weighted mean-square error (*Wmse*) of a codeword. The *Wmse* is simply the square of the difference between octs and their corresponding centroid, weighted by the individual distortion of the octs:

$$Wmse = \sum d_j (x_j - c)^T (x_j - c) \quad . \qquad (7)$$

After cancellation of the common terms and simplifying, we get

$$Wmse = \sum d_j (x_j^T \cdot x_j) - (c^T \cdot \sum d_j x_j) = \lambda - (c^T \cdot \mu) \ (8)$$

Both λ and $\underline{\mu}$ are updated by the TSVQ_A processors, and the multiplication with the centroid and the final subtraction is handled by the TSVQ_B processor at the end of each iteration.

5.1 TSVQ_A Processor and Oct Memory

Each TSVQ_A processor performs the following three oct operations: converting preocts to octs, distance determination and accumulation of centroid weights of different octs. Preoct to oct conversion is performed immediately after prequantization of an image. It involves converting all the octs' centroids from integer to floating point, generates the distortion from the inverse distortion and the weighted centroid square from both as described in the previous section. In distance determination unit using the values α and β from TSVQ_B processor, the inequality (6) is checked for all octs mapped to the current codeword to partition them into left and right codewords. Once the octs are tagged as belonging to either left or right codewords, their weighted values λ, μ and δ are accumulated in the weight summation block, and the sums are sent to TSVQ_B for codeword generation.

One important feature of the oct memory/TSVQ_A processor configuration is that the octs belonging to a codeword are always written horizontally across the oct memory banks. This scheduling of the memory banks is handled by an overall controller which keeps a table of occupied horizontal lines in the oct memory.

5.2 TSVQ_B Processor and Codeword Memory

The TSVQ_B processor performs codeword operations: computes new centroid for codewords from centroid weights produced by the TSVQ_A processors, and determines whether to repeat the GLA iteration for codewords. Once the desired number of colors are obtained, the color palette is output from the codeword memory to the palette DAC.

The first part of the processor computes the centroid and *Wmse* for the left and right codewords using the equations (4) and (8) using the weight centroid block. Once they have been computed, the L-R centroid blocks compute α and β. The floating point centroid vector is converted to an integer centroid vector to the color palette precision. The codewords are stored in the codeword memory in colorwords according to format discussed in Section 3. The next part requires the selection of the codeword with the largest *Wmse*. To achieve this, a heap of all leaf codewords is kept by the TSVQ_B processor. Using the heap, the maximum codeword can be obtained in $O(1)$ time, and insertion of new codewords in $O(\log n)$ time, where n is the number of codewords in the heap.

6. The Color Quantizer Encoder

In this design, the pixels used in training are assumed to be the same pixels to be encoded later. This setup produces optimal results for color quantization, as all pixels in the image are mapped from the image space to the codewords through the octs. The octtree representation stored in the oct memory allows the color quantizer trainer to act directly as an encoder as well. Since each oct has a pointer directly to its codeword, the codeword is found immediately without having to search over the entire codebook. The encoder will proceed from the prequantizer processor, reading in pixels. The activity generators are by-passed and the pixels are hashed directly to locate their corresponding octs. Once the oct is found, its codeword pointer gives the location of the codeword in the codeword memory. Through TSVQ_B, the codeword is accessed and the centroid is output to the frame buffer. TSVQ_A processors are not used in the encoding process, and hence can be shut down to conserve power.

7. Computational Requirements

Simulations results based on a 24-bit to 8-bit mapping of a CIF format (352x240) image with 3,000 octs requires 500K fixed-point additions/subtractions and 4K fixed-point multiplications for prequantizer, 290K floating-point additions and 165K floating-point multiplications/divisions for TSVQ_A, and 7K floating-point additions/subtractions and 9K floating-point multiplications for TSVQ_B. This can be implemented with one prequantizer processor, two banks of oct memory, two TSVQ_A processors and one TSVQ_B processor. With a speed of an addition/subtraction in fixed point being 4 ns, addition/subtraction in floating point being 15 ns and a multiplication/division being 40 ns, the image takes 2.16 msec to prequantize and 11.42 msec to perform TSVQ, which is under the real-time constrain of 33.33 msec at 30 frames/sec. The average PSNR obtained using our algorithm is over 42 dB.

8. Conclusion

A hardware implementation of a fast TSVQ algorithm with prequantizer is presented. This design makes color quantization of real-time image sequences possible. This scalable design also ensures that advances in color resolution need not be matched with a proportional increase in frame buffer size, while fully exploiting the whole range of color gamut the increased resolution offers.

Acknowledgments

Navin Chaddha was supported by an IBM Graduate Fellowship. This work was supported in part by ARPA.

References

[1] D. Maliniak, "Color palette d-a converters paint a bright video picture," *Electronic Design*, Vol. 35(26), Nov. 1987, pp. 81-84, 87.

[2] L Letham, et al., "A high performance CMOS 70-MHz palette/DAC," *IEEE JSSC*, Vol. SC-22(6), Dec. 1987, pp. 1041-1047.

[3] A. Gersho and R. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, 1992.

[4] G. Braudaway, "A procedure for optimum choice of a small number of colors from a large color palette for color imaging," *Electronic Imaging*, 1987.

[5] C. Bouman and M. Orchard, "Color image display with a limited palette size," *Proc. SPIE Conf. Visual Comm. and Image Processing*, Nov. 8-10, 1989, pp. 522-533.

[6] N. Chaddha, W.-C. Tan, and T. Meng, "Color quantization of images based on human vision perception," *Proc. IEEE ICASSP 1994*.

[7] N. Chaddha, W.-C. Tan, and T. Meng, "Fast VQ algorithms for color palette design based on human vision perception," submitted to IEEE Tran. on IP.

[8] T. Cornsweet, *Visual Perception*, Academic Press, New York, 1970.

Figure 1: Organization of the data structures.

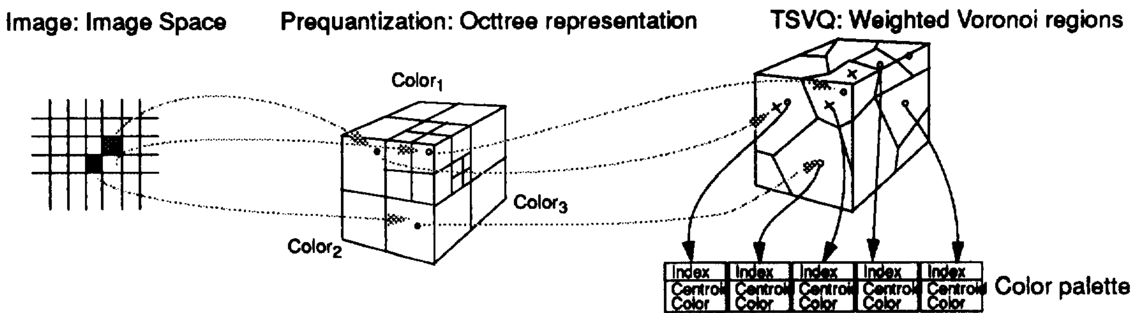

Figure 2: Block diagram of the color quantizer chip-set.

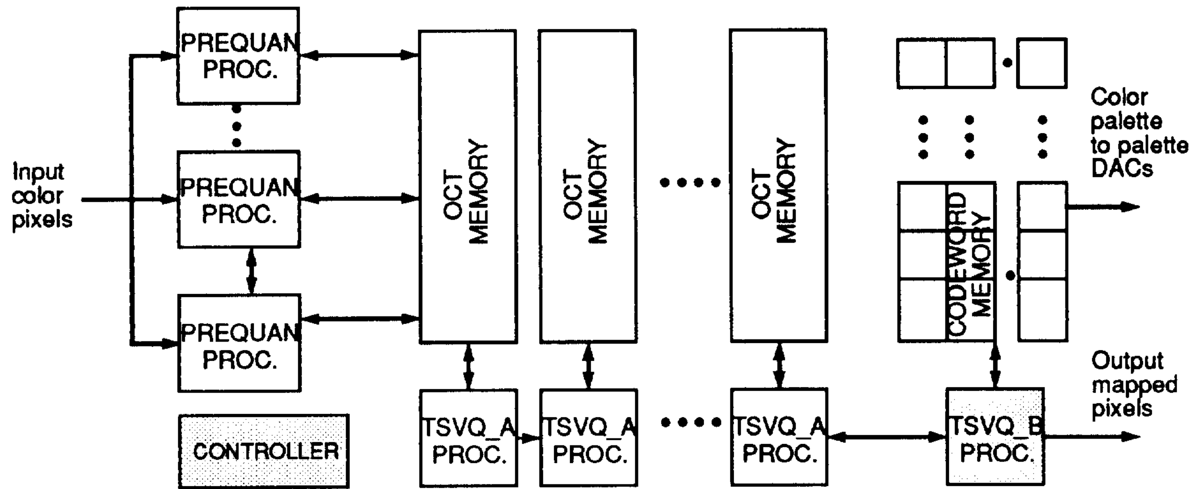

Figure 3: Block diagram of the prequantizer processor.