Color quantization using octrees

Dan S. Bloomberg
Leptonica
http://www.leptonica.org/papers/colorquant.pdf

September 4, 2008

Abstract

We describe methods for performing color quantization on full color RGB images, using an octree data structure. The advantage of the octree is that it is simple to generate both a good partitioning of the color space and a fast inverse color table to find the color index for each pixel in the image. With only 256 colors, it is often necessary to error-diffusion dither the color for appearance, even though this increases the RMS pixel error. We dither using integers for efficiency without loss of color accuracy.

1 Introduction

The problem of color quantization is to represent full color RGB images, where each pixel is typically described by three 8-bit color samples, in an approximate fashion by a relatively small number of colors. We will assume that each color is represented by its 24-bit RGB value. Historically, the number of colors used has been determined by the depth of the display frame buffer, often 8 bits. Thus the full color space, consisting of about 16 million pixels (2^{24}), is divided into a small number of regions, and for each region, a single representative color is used for each pixel that falls into the region.

There are many algorithms for vector color quantization that can be found in the literature, some of which are quite good, and others quite poor. Furthermore, there is an excellent Open Source implementation in the jpeg jfif library [6]. Do we need another?

I believe the answer is ”yes.” The generic difficulties with vector quantization derive mostly from implementation inefficiency, both in the decision process for breaking up the color space and the assignment of samples to the correct partition. These difficulties have typically been overcome

by compromising the quality of the result, and the art is to find a way to minimize the loss of quality while using a fast quantizer. This report describes a particular approach to this problem.

There are in general two steps in the color quantization of a 24-bit RGB image. The first is to partition the color space into a small number of colors, and the second is to assign one of these colors to each pixel in the image. The second step requires a traversal through every pixel in the input image, using an *inverse color table* to map from the RGB value to the color table index. The visual result is usually improved with color *error diffusion dithering* (EDD). The first step requires some analysis of the image for best results, although one can also use a predetermined partition of the color space. The various approaches are described below.

1.1 One-pass quantization with fixed partitioning

The simplest color quantization takes a fixed, pre-determined color space partitioning, and performs a single pass over the input image to assign a color index to each pixel. The color index is always assumed to be an index into three 8-bit color tables (RGB). Each color table is typically 8 bits, to allow mapping of full RGB color onto an 8 bit display frame buffer. However, memory has recently become sufficiently inexpensive so that most displays today have either 16 or 24 bits of depth. Fixed, equal-volume partitioning suffers from serious contouring and loss of color fidelity without EDD. However, with EDD, visual color accuracy is reasonably good.

1.2 Two-pass quantization methods

There are several existing methods for adapting the partitioning the color space based on the pixels in a particular image. Of them, we briefly describe the *k-means, popularity, median cut,* and *octree* methods. The division of the color space can be represented either by a flat structure that describes the partitions, or by a tree. A tree can be generated either by starting with the entire color space and *splitting* it up, or by starting with a large number of small volumes and *merging* them. The division of colors along a color axis can be either flat or hierarchical. The *popularity* method uses a simple selection of colors from the color space, the *median cut* method uses a splitting algorithm with a flat division of the space, and the *octree* method is hierarchical and is naturally amenable to an algorithm that merges regions of color space.

The *k-means* method, and the related *Linde-Buxo-Gray* algorithm are iterative methods for clustering, with typical applications being pattern recognition and data compression. They are really multi-pass, not two-pass methods. In k-means, you choose the number of clusters and an initial set of cluster centers. Then, iteratively do the following until convergence or you run out of patience:

- scan the image, assigning each pixel to its nearest cluster center
- compute the centroid of each cluster, and use this as the new cluster center

The total error is guaranteed not to increase from one iteration to the next, so the method will converge to a locally optimal solution. However, this is not guaranteed to be globally optimal; the final centers will depend on the initial centers that are chosen. The LGB algorithm is similar. Unfortunately, these clustering methods are not practical for color quantization because, in addition to sensitivity to initial conditions, they require many iterations.

The *popularity* method was described by P. Heckbert [1] and an implementation was given by D. Clark [2]. The color representatives in the *color table* are selected as the colors that are most populated in the image. Generate a histogram of the colors, clipped to (e.g.) 5 bits in each sample of R, G, and B, and choose some number, typically not more than 256, of the most populated bins. Although very simple, this method has two serious drawbacks. First, if the image has many different colors, this will do badly on any colors far from the selected ones in the color table. Second, it will in general do poorly when using dithering, because dithering can only interpolate within the convex hull of the colors selected in the color table.

The *median cut* method partitions the color space to put roughly equal numbers of pixels in each color cell. It was originally described by P. Heckbert [1] and implementations are given by A. Kruger [3] and in the open source JFIF jpeg library [6]. Median cut is implemented by repeatedly dividing the space in planes perpendicular to one of the color axes. The region to be divided is chosen as the one with the most pixels, and the division is made along the largest axis. The division itself is chosen to put approximately half the pixels in each part. The method does well for pixels in a high density region of color space, where repeated divisions result in small cell volumes and color errors. However, the volume in a low density part of the color space can be very large, resulting in large color errors. In the open source jpeg library implementation, half the cuts are chosen from the highest population cells, and half are chosen from the highest volume cells. Making some of the cuts based on cell volume has two important effects. First, such cuts reduce the largest color errors that may occur. Second, by spreading the representative colors in the color table more evenly through the color space, they allow better results with dithering. More recently, I have implemented a modified version of median cut that, like the JFIF jpeg version, performs a composite partitioning. In the leptonica version, some of the partitions are based on population and others are based on the product (population * volume). There are some other tricks, and the result is quite good. A report on the leptonica median-cut method can be found at *http://www.leptonica.org/papers/mediancut.pdf* [7].

In general, *Octrees* are a good way to divide up the color space, while allowing fast pixel

indexing with an inverse color table. One method has been published, by M. Gervautz and W. Purgathofer, a summary of which can be found in A. Glassner's *Graphics Gems I* [5]. Unfortunately, their description is somewhat terse. They built an octree by taking pixels, one at a time, and either making a new color or merging the pixel with an existing color. After the prescribed number of colors in the color table have been established, the new pixel either makes a new color (causing merging of two existing colors), or it is merged with an existing color. Each color is represented by an octcube, or a set of octcubes, so the merging operation effectively prunes the octree. The method has the advantage that only the prescribed number of colors needs to be stored at any time. It has the disadvantage that the merging operations are complicated, and it is not easy to understand how to spread the colors through the full color space, which is necessary for dithering. An implementation of this approach is given by D. Clark [4].

We now proceed to describe various octree-based methods implemented in leptonica.

2 Octree color space partitioning

2.1 One-pass baseline version

We provide a one-pass implementation, `pixFixedOctcubeQuant256()`, using equal volumes of color space, for a baseline comparison with the two-pass methods. Each of the 256 color cells that fill the color space is composed of two third-level octcubes. We choose to divide the blue dimension down to only the second level, because of the relative insensity of the eye to blue. The color can be chosen either to be at the center of the octcube(s) or at the centroid of pixels in each region. We implemented it both ways and found it made little difference without using dithering and no difference when dithering. Thus, for simplicity, the present implementation uses the center of the octcube(s). Without dithering, the result is relatively poor on images with large regions of similar color. However, *because the color table covers the entire color space, with dithering the results are surprisingly good on all images*.

2.2 Simple two-pass population-based method

A fairly simple approach to building an octree is implemented in `pixOctreeQuantByPopulation()`. The colorspace is broken up into "level 4" octcubes, meaning that each color axis is divided into 16 parts, for a total of 16 x 16 x 16 = 4096 level 4 octcubes. In the first pass through the image, the number of pixels in each cube is summed, and the number of cubes with at least one pixel is found. For a typical image, with more than 256 such colors, 192 colors are reserved for the most

populated of these level 4 octcubes. This leaves 3904 level 4 octcubes that are not represented, so to get a colormap assignment for every possible octcube, the last 64 colors are taken to be all 64 level 2 octcubes. The actual color assigned to each of the 192 most popular colors is the average of all pixels that land within the level 4 octcube. Likewise, the actual color assigned to the remaining 64 colormap entries is the average color of the residual pixels (i.e., those pixels *not* in the 192 small octcubes) that fall in each of the level 2 octcubes. Generation of the colormap, in this way, is accompanied by generation of an "inverse" lookup table that maps from the level 4 octcube index to the colormap index. After the colormap and inverse lookup table are generated, the second pass is performed. Each source pixel gets mapped from its RGB value to level 4 octcube index, and thence to the colormap index, which is stored in the output image. This method is quite fast, gives reasonably good results for most images, and gives outstanding results for orthographically generated images (such as maps) where most of the pixels are in a relatively small number of colors, but, due to aliasing and compression artifacts, there are far more than 256 colors in total.

2.3 General octree quantization

We give below the details for generating an adaptive color octree, where the octcubes at various levels are selected based on the actual distribution of pixels within the color space, as well as the requested number of colors in the result. (The simpler population-based octree, described above, is pre-determined to have octcubes at levels 2 and 4 only.) The code in `colorquant1.c`, which is fairly heavily commented, supplements the description here.

2.4 Octree data representation and indexing

The two-pass octree method used here is simpler than the method of Gervautz and Purgathofer, both conceptually and in implementation. The basic data structure is a set of arrays that describe the leaves of an octree at each of the levels of interest, 0 through N, where N is the finest level of subdivision. Our implementation allows a choice of 4, 5 or 6 for N. At each level n, there are eight cubes that correspond to a single cube at level $n-1$. The cubes are indexed so that the cube i at level $n-1$ contains the eight cubes $8i+j$, $j = 0, ..., 7$ at level n. The cube index is computed from the RGB values by taking the most significant bits of the three color samples in the order:

```
r7 g7 b7 r6 g6 b6 r5 g5 b5 r4 g4 b4 r3 g3 b3 ...
```

down to the level of interest. For example, at level 5, the index consists of the 15 bits shown above. For any RGB value, it is easy to compute the cube indices at any level of the octree. This is one of the big advantages of the octree representation. It should be noted that the octree represented by

this set of arrays is *virtual*, in that we have no pointers going between the different levels. Instead, we have a *pyramid* of arrays of octcube leaves, one at each level, with fast indexing into each array.

2.5 Pruning the octree

The octree with the selected *color table entries* (CTEs) is built in the first pass. The image is scanned and the pixel counts are placed in the octcube (leaves) at a chosen level N, using the index mapping above. The deepest level N can be either 4, 5 or 6. For example, using $N = 5$, there are 2^{15} leaves. It is not necessary to use all the pixels in the image, so we take pixels from a subsampled version. Using only about seven percent of the pixels (subsampling by a linear factor of 4) gives a sufficient approximation to the pixel statistics. After the sampled pixels are placed in their octcubes at level N, the tree is pruned back. The goal is to label the octcube that corresponds to each CTE. For simplicity, a CTE is represented by either (a) a single octcube or (b) a set of octcubes, all at the same level, that belong to a single octcube at the next level up.

The tree is then pruned from level N, taking the octcubes in sets of eight, where the eight octcubes are those that compose the octcube at the next level up. For each set of eight octcubes at some level, one or two of the following conditions will be obtained:

1. One or more of the octcubes has already been selected as a CTE.
2. One or more of the octcubes that is not already a CTE has enough pixels to become a CTE. Make it into a new CTE.
3. None of the cubes is already a CTE or has enough pixels to become a CTE.

In both the first and second cases, the containing octcube at the next level up automatically becomes a CTE, which contains those subcubes that are not already a CTE. (The exception is in the special case where all 8 subcubes are already CTEs; the containing octcube is marked as taken but not assigned to a separate CTE.) In the third case, nothing is done at this level. When all cubes are processed at this level, the procedure is repeated at the next level up.

The decision for forming a new CTE is that the number of pixels in the cube exceeds a threshold that is proportional to the number of pixels yet to be assigned to a color divided by the number of colors left to be assigned. We actually hold back 64 colors in reserve, because when we get to the second level, we require that each of these 64 octcubes be a CTE by default. (We only prune back to level 2.) In this way *we constrain the maximum color error, while insuring that the entire color space is covered by the color table.* The value 1.0 for the proportionality constant at each level above 2 works well; the constant for level 2 is not used because the octcubes become CTEs automatically.

One problem with this approach is that the actual number of colors in the color table depends on the distribution of colors in the image. It can happen that 64 reserved colors is not enough. Suppose we prune back from level $N = 6$. Suppose an octcube at this level has enough pixels to become a CTE. This will force the containing octcubes at levels 5, 4 and 3 all to be CTEs. In a pathological case, where it is possible to run out of colors, three things can be done to reduce the number of colors that are actually used. You can start pruning from a lower level (5 or 4), you can specify a smaller total number of colors, or you can increase the proportionality constants for the CTE threshold values to reduce the number of CTEs at the deeper levels. In the leptonica algorithm, you specify the desired number of colors, but the algorithm is permitted to give more, up to a limit of 256. The initial octcube level and the threshold constants are compiled in.

The color associated with each CTE is the center of the controlling octcube. This is easier than maintaining a running centroid value, and has very little effect on the result. In fact, because the octcubes at each level are indexed as they would appear in the octree from left to right, the center of the octcube in which any pixel falls can be rapidly computed. This center value is also stored in the CTE, from which it can be rapidly extracted for dithering.

2.6 Assignment of color indices to image pixels

The pixels are assigned a CTE index on the second pass. This can be done very quickly in two different ways, of which we have implemented the second:

1. *Make an explicit inverse colormap.* Compute and store the index in the leaf array at the deepest level, in the place were we stored the pixel counts in the first pass. Then for each pixel, convert the RGB value of the pixel to the truncated octcube index at that level, and look up the colortable index from this array. No extra storage is required because we have already allocated the octcube array at this level.

2. *For each pixel, run down the tree from the root to find the CTE octcube that it belongs to.* This is done by converting the RGV value to an index into the array of octcubes at each successive level, stopping when you find an octcube that is marked as a CTE and the octcube at the next level down is *not* a CTE. If you reach the bottom, the octcube will be marked as a CTE and you take it.

For large images where the number of pixels is much larger than the number of octcubes at the deepest level, the first method will be faster. When dithering, the propagated error allows pixels, in principle, to have any color within the color space. This is the reason that the CTEs must cover the entire space.

3 Error diffusion dithering

Without dithering, a typical 256 color image will show severe contouring and color distortion, regardless of the method of choosing the colors. The color artifacts are particularly evident when the image has large regions of slowly-varying color. Consequently, if the goal of color quantizing is to preserve the appearance of the full color image, dithering is mandatory. Error-diffusion dithering is the most common and least objectionable type of dithering. There are four requirements:

- **Causality**. The error is diffused in raster or anti-raster order, to pixels that have not yet been treated.
- **Two-dimensional diffusion**. The error must be diffused to pixels that are located both in the current row and in one adjacent row.
- **Locality**. The error is diffused to nearby pixels, and the closer the pixel, the larger the fraction of error that is given to it.
- **Normalization**. The total error is diffused.

Two-dimensional diffusion, locality, and normalization are necessary to guarantee that the average color in a small region is close to the average color in the original. These constraints leave flexibility in the details of the diffusion. The original Floyd-Steinberg (F-S) method diffused the error to four adjacent pixels, with fractions 7/16, 3/16, 5/16 and 1/16.

We choose to diffuse the error to three pixels, with fractions 3/8 (to right), 3/8 (down) and 1/4 (down and to right), because this requires less computation and there is no significant penalty in appearance. Also, unlike F-S, because we do not diffuse to the left, we do not have a special case for the leftmost boundary pixels. To increase computation speed, we perform all computation in integers, and scale the pixel values up and down by a power of 2. We scale up to reduce roundoff error to a small fraction of a pixel level increment, and use power of 2 scaling for fast multiplication and division.

In more detail, we diffuse in raster order, and, for each color, keep line buffers for the current line and for the next line. The data in the current line buffer are copied to the next line buffer, and the the samples from the current line are multiplied by 64 and entered in the current line buffer. For each pixel, the error is the difference between the pixel value and the center of the cube in which the pixel is placed. We split the error into eighths. The error (multiplied by 64) is diffused and the sample value in each buffer is clipped to 0 and $2^{14} - 1$ to avoid overflow.

4 Discussion of the implementation and results

We consider a typical RGB image with 10^6 pixels. As mentioned above, the octree color table can be generated accurately, in the first pass, using about 70,000 pixels. The conversion speed for generating a colormapped image from an RGB image depends on the deepest level at which pixels are allowed to form CTE octcubes. On a million pixel RGB image, using a 1 GHz Pentium III, the total conversion time for levels 4, 5 and 6, which have 2^{12}, 2^{15} and 2^{18} leaves, is 0.40, 0.45 and 0.60 seconds, respectively.

It is difficult to show results with sufficiently good quality to compare, for example, the dithered versions of one-pass and two-pass color quantization. One must select an image that has many different colors, as well as regions of slow color sweep. Images with a variety of flesh colors are good tests because they are typically hard to represent accurately with a small number of colors. For display and print in tex, the color images must be converted to PostScript, and then rendered by the display engine (e.g., acroread) or the PostScript decomposer in the printer. The display is subject to the frame buffer color quantization, along with possible halftoning (e.g., gv, ghostview), and the print always loses resolution because a halftone screen is applied to the image. The image is best displayed in a browser, using one of the two native image formats (png or jpg), and we do that here. The figures referenced below can be seen in the color quantization section of:

http://www.leptonica.com/applications.html

As noted there, the actual rendering of these images will depend on the browser, the frame buffer depth (8, 16 or 24 bits), and the video display card.

The results for both one-pass and two-pass quantization are shown for an image that was chosen to highlight the difficulties in color quantization. This image has a large variety of flesh tones and a background with a very slow color sweep, both of which expose small color errors.

Figure 1 uses the baseline one-pass quantization with equal volumes spanning the color space, and no dithering. The contouring and color errors are very noticeable.

Figure 2 uses dithering on the one-pass quantization with equal volumes spanning the color space. The results are surprisingly good for such a crude color table.

Figure 3 uses the two-pass octree color quantization, with pruning from level $N = 5$, but without dithering. The contouring and color errors are much reduced from those using the one-pass color table (Figure 1), but the necessity for dithering is apparent from the evident contouring and color errors.

Finally, Figure 4 is made using the two-pass octree color quantization, again with pruning from

level $N = 5$, and with dithering. The result is comparable to the best median cut methods (both the JFIF jpeg implementation and the modified median cut in leptonica), and it is about as good as can be done with 256 colors. This should be compared with Figure 5, which shows the original full color RGB image.

References

[1] P. Heckbert, "Color image quantization for frame buffer display,", *Computer Graphics*, **16**(3), pp. 297-307 (1982).

[2] D. Clark, "The popularity algorithm," *Dr. Dobb's Journal*, pp. 121-127, July 1995.

[3] A. Kruger, "Median-cut color quantization," *Dr. Dobb's Journal*, pp. 46-54 and 91-92, Sept. 1994.

[4] D. Clark, "Color quantization using octrees," *Dr. Dobb's Journal*, pp. 54-57 and 102-104, Jan. 1996.

[5] M. Gervautz and W. Purgathofer, "A simple method for color quantization: octree quantization," in A. Glassner, ed, *Graphics Gems I*, Acad. Press, 1990, pp. 287-293.

[6] ftp://ftp.uu.net/graphics/jpeg, see jquant2.c in version 6b of JPEG library, 1998.

[7] D. Bloomberg, "Color quantization using modified median cut", *http://www.leptonica.org/papers/mediancut.pdf* , 2008.