REIHE INFORMATIK
015/94

eXtended Color Cell Compression – A Runtime-efficient Compression Scheme for Software Video

Bernd Lamparter und Wolfgang Effelsberg
Universität Mannheim
Seminargebäude A5
D-68131 Mannheim

eXtended Color Cell Compression – A Runtime-efficient Compression Scheme for Software Video

Bernd Lamparter and Wolfgang Effelsberg

Praktische Informatik IV
University of Mannheim
68131 Mannheim, Germany
{lamparter, effelsberg}@pi4.informatik.uni-mannheim.de

Abstract. Multimedia applications require a compression and decompression scheme for digital video. The standardized and widely used techniques JPEG and MPEG provide very good compression ratios, but are computationally quite complex and demanding. We propose to use an extension to the much simpler Color Cell Compression scheme as an alternative. Our extension includes the use of variable block sizes, the reuse of color index values from previously encoded blocks, and Huffman encoding of the stream of blocks. We present experimental results showing that our scheme provides much better runtime performance than MPEG, at the cost of a slightly inferior compression ratio. It is thus especially suited for software videos in high-speed networks.

Keywords: multimedia, movie compression, block encoding, software video.

1 Introduction

The standardized compression techniques JPEG [12] for still images and MPEG [3] for motion pictures both include a Discrete Cosine Transform (DCT). This is a complex and computationally very demanding mathematical function. As a consequence, software motion pictures based on JPEG or MPEG are slow, even on the most powerful CPUs available today, and it is generally assumed that these compression schemes will only work well with special hardware. However, special hardware makes a movie system much less flexible and portable. It is thus desirable to develop alternative compression/decompression algorithms for movies which are optimized for computation in software on general purpose CPUs.

We propose an extension to the Color Cell Compression scheme for use in multimedia workstations. After a short introduction into Block Truncation Coding for monochrome images and Color Cell Compression for color images, we describe our eXtended Color Cell Compression (XCCC) scheme in detail. We have implemented XCCC and present experimental results on runtime performance and compression ratios.

2 Block Truncation Coding and simple Color Cell Compression

Our eXtended Color Cell Compression (XCCC) algorithm belongs to the family of block compression algorithms. Earlier examples from this family include Block Truncation Coding (BTC) and Color Cell Compression (CCC), brief descriptions of which preface the discussion of our algorithm.

2.1 Block Truncation Coding (BTC)

The Block Truncation Coding Algorithm [2] is used in the compression of monochrome images. When compressing color images, it can be applied separately to the three color channels.

The first step of the algorithm is the decomposition of the whole image into blocks of size $n \times m$ pixels. Usually these blocks are quadratic with $n = m = 4$. For each block P the mean value μ and the standard deviation σ is computed:

$$\mu = \frac{1}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} P_{i,j}$$

$$\sigma = \sqrt{\frac{1}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} P_{i,j}^2 - \mu^2}$$

where $P_{i,j}$ is the brightness of the pixel.

In addition a bit array of size $n \times m$ is calculated for each block. A one in this bit array indicates that the gray value of the corresponding pixel is greater than the mean value, a zero indicates that the value is smaller than the mean value:

$$B_{i,j} = \begin{cases} 1 & \text{if } P_{i,j} \geq \mu \\ 0 & \text{else} \end{cases}$$

The decompression algorithm knows out of the bit array whether the pixel is darker or brighter than the average. Last we need the two gray scale values for the darker and for the brighter pixels. These values (a und b) are calculated with the help of the mean value and the standard deviation, and are then stored together with the bit array:

$$a = \mu + \sigma\sqrt{p/q}$$
$$b = \mu - \sigma\sqrt{q/p}$$

Here p and q are the number of the pixels having a larger resp. smaller brightness than the mean value of the block.

During the decompression phase each block of pixels is calculated as follows:

$$P'_{i,j} = \begin{cases} a & \text{if } B_{i,j} = 1 \\ b & \text{else} \end{cases}$$

e. g. where the bit array shows a 1, the gray value a is used, where it shows a 0 the value b is used.

If the original image used one byte per pixel, we had a storage requirement of 128 bits for each 4×4 block. The compressed block can be stored with 16 bits for the bit array plus one byte for each of the values a und b. Hence we have a storage reduction from eight bits to two bits per pixel.

This basic version of BTC can be improved with a number of tricks [9]. Additionally, [10] describes a hierarchical version of BTC.

2.2 Color Cell Compression (CCC)

If BTC is to be used for color images rather than for gray scales, the components (red, green, and blue, resp. chrominance and luminance) may be compressed separately. However, the CCC method promises a much better compression rate [1].

Similar to BTC, the image is divided into blocks called "color cells". The two values a and b are now indices into a color lookup table (CLUT). The criterion for the bit array values is now the brightness of the corresponding pixel. The brightness of a pixel is computed in the following way, taking the human reception into account:

$$Y = 0.3 P_{\text{red}} + 0.59 P_{\text{green}} + 0.11 P_{\text{blue}}$$

The mean value of each block can now be computed out of these brightness values (analogous to the BTC method).

Let us define $P_{\text{red},i,j}$ as the red component of $P_{i,j}$, $P_{\text{green},i,j}$ the green component and $P_{\text{blue},i,j}$ the blue component. The next step is then to compute the color values a_{red}, a_{green}, a_{blue} as well as b_{red}, b_{green}, b_{blue}:

$$a_c = \frac{1}{q} \sum_{Y_{i,j} \geq \mu} P_{c,i,j} \quad \text{bzw.} \quad b_c = \frac{1}{p} \sum_{Y_{i,j} < \mu} P_{c,i,j} \quad \text{with } c = \text{red, green, blue}$$

Again p and q are the number of pixels with a brightness larger resp. smaller than the mean value. The bit array is computed as for BTC.

The color values $a = (a_{\text{red}}, a_{\text{green}}, a_{\text{blue}})$ und $b = (b_{\text{red}}, b_{\text{green}}, b_{\text{blue}})$ are now quantized onto a color lookup table. In this way we get the values a' und b'. These values are stored together with the bit array (Fig 2.2).

The decompression algorithm works analogous to the BTC method:

$$P'_{i,j} = \begin{cases} \text{CLUT}[a'] & \text{if } B_{i,j} = 1 \\ \text{CLUT}[b'] & \text{else} \end{cases}$$

(CLUT is the color lookup table.)

The two values a' und b' can each be stored in one byte if the CLUT has 256 entries. Hence the storage needed for one block of size 4×4 is two bits per pixel as with the BTC (to be more exact we would have to add the storage needed by the CLUT (256×3 Bytes for the full image)).

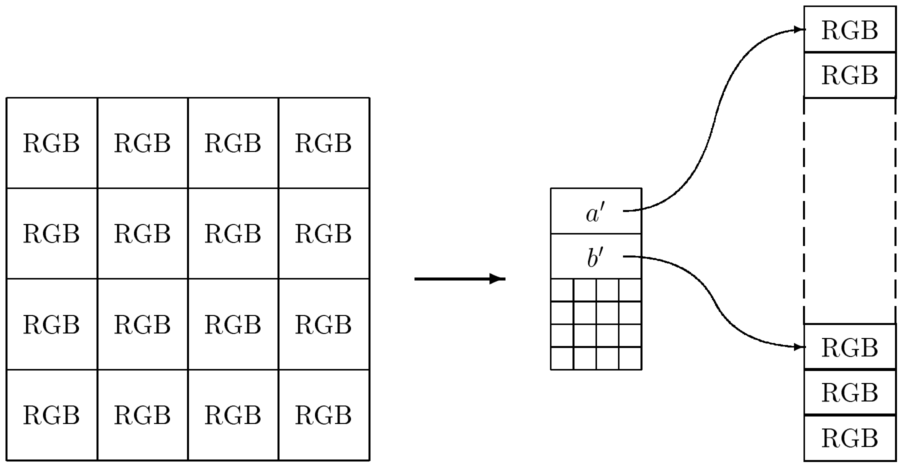

Fig. 1. Red-Green-Blue block and its CCC encoding

Color Cell Compression is not only one of the best compression algorithms, it is also one of the fastest [9]. All calculations can be done without floating point operations, and the asymptotic complexity is $O(N \cdot M \cdot (1 + \frac{\log k}{n \cdot m}))$ (image size $N \times M$, size of the block $n \times m$, size of the CLUT k). The decompression is also done without floating point operations with a complexity of $O(N \cdot M)$.

As in the case of the BTC algorithm, a number of possible improvements exist here as well [9]:

– If the two colors a and b are nearly equal, or one color dominates in frequency of occurrence, only one color is stored, and no bit array is needed.
– If an image contains large areas with only small differences in color, those areas may be encoded with larger blocks.
– For movies cuboids may be used, with time being the third dimension, if the changes from frame to frame are small enough.

Our algorithm "XCCC" is based on the second idea. It uses 4×4, 8×8, and 16×16 blocks. Larger blocks, tested in an earlier version, yielded no further improvement.

2.3 The DeltaCLUT-Technique

CCC (and XCCC) both use a color lookup table for storage-efficient color representation. The color lookup table typically has 256 entries, with the red, green, and blue components stored in one byte each. The color value of a pixel is then encoded as an index into the color lookup table, with one byte per pixel. Most of the color display adapters today use the color lookup table. The decompressed image already consists of index entries into the CLUT, hence the decoder does not have to map the 24 bit colors onto 256 CLUT entries (That is a very expensive step for MPEG decoders [8]). Moreover, we get a better compression ratio for a and b because of the reduction from 24 bits to 8 bits per color.

As our experiences shows, one color lookup table for the whole movie results in poor colors. Updating the CLUT during the replay of the movie may result in false colors because the old frame is shown with the color table entries of the new frame for a short time; that has a visually very disturbing effect. Loading the CLUT after loading the frame results in the reverse effect: the new frame is shown with the CLUT of the old. In [7] a method is presented preventing these problems: DeltaCLUT. The DeltaCLUT algorithm reserves a number of CLUT entries for color updates, and dynamically loads new colors into the CLUT while the movie is running. This combines the usage of the full color spectrum for the movie with the storage-efficient and widely deployed color lookup table technology.

3 XCCC: Extensions to CCC

Our XCCC scheme extends CCC in three steps in order to improve compression ratio and runtime performance.

3.1 First step: Adaptive block sizes

In many cases a image has large areas with small differences in colors (i.e. in the background). These areas can be coded with fewer bits.

We first investigate the optional use of large rectangles. If an image contains large areas with few colors, these areas can be compressed with larger rectangles [11].

In XCCC the images are first decomposed into large blocks (16×16) and, if necessary, these blocks are then subdivided. The algorithm for each block B is:

1. Calculate the CCC coding of the block
2. If the actual block has the minimal block size, then Done.
3. Calculate the mean difference Δe of the original pixel values and the values coded with CCC: $\Delta e = \sum_B |p - p'|_2$, where p is the pixel value, p' is the value of the same pixel after decompression.
4. If Δe is smaller than a given constant, then Done.
5. Divide the block into four subblocks and use the algorithm recursively for each of these blocks.

The data for simple CCC could be arranged in the data stream without any structuring information. But the output of the extended algorithm is a quadtree with color cells for each 16×16-block. Hence, we have to store a more complex data structure. This is done by adding a tag for each block. Figure 2 shows an example of the coding of an XCCC block. The tag is the logarithm of the length of the edge of the coded block. Blocks of minimal size need only one tag for four blocks because one minimal block is always followed by three more. After the tag we store the indices into the CLUT (a and b) and then the bit array. If the length or width of the image is not divisible by 16, we divide the residual rectangle into 4×4 blocks and encode some of these subblocks as rectangles.

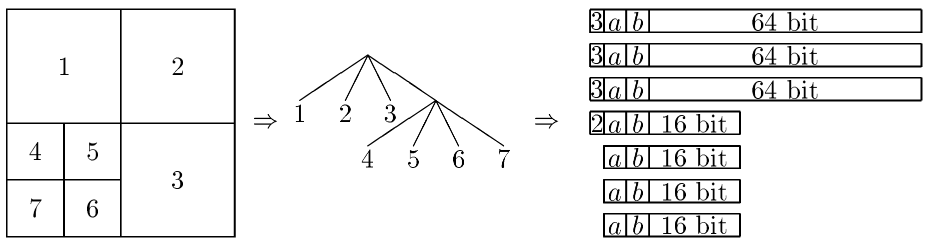

Fig. 2. Coding tree of the XCCC-algorithm

The use of adaptive block sizes introduces a small additional overhead for compression and results in much more efficient decompression for most images.

3.2 Second step: Single Color and Color Reuse

If an image has a large area with only one color, it is not necessary to store the bit array at all. XCCC does not store a bit array if the two color indices a and b are equal. There are two ways to let the decoder know that there is no bit array: First the encoder can store the two colors and the decoder will know from the equality that there is no bit array. Second, the encoder can use a special tag and store only one color. XCCC uses the second method.

If we implement bit array suppression for single color blocks, larger squares will not always improve the compression ratio. Instead of a 32×32 square, XCCC may use four 16×16 squares. But some of these squares will have no bit array and hence the total compression ratio may be better. Though this can also happen with smaller blocks, our experience shows that this is rarely the case.

Colors in the neighborhood are often equal in images. Because of this fact, we can sometimes reuse colors from the block coded previously. Color reuse is also stored in the tag.

For each tag we use one byte with the following bit encoding:

	7 6 5 4 3 2 1 0
4×4-block	- - - - - - 1 0
8×8-block	- - - - - - 1 1
16×16-block	- - - - - - 0 0
single color	- - 1 - - - - -
last dark	- 1 - - - - - -
last bright	1 - - - - - - -

The bit for "last dark" indicates that the color value b for the dark pixels should be reused from the previous block, "last bright" indicates the usage of value a from the previous block.

In the first step the encoder used only one tag for four of the smallest blocks. But now the tags of these four blocks are possibly different. If one of them has one of the bits 5, 6 or 7 set, the tag must be stored. The first tag is now the leader tag and the last bits are set to 01 instead of 10. the bits 2, 3 and 4 indicate which of the three tags are following.

Let us consider an example: 0101 1001 0010 0010 1000 0010. The first tag says, that the last dark color index should be reused, and tags number 3 and number 4 will follow. Tag 2 is implicit of simple type, 3 uses only one color and 4 reuses the last bright index.

3.3 Third step: Further improvements

Some blocks are encoded in only one ore two bytes, namely the blocks without bit array. If one of the four subblocks of a 16×16 block is a single color block, then it is cheaper in memory to store the four subblocks instead of the 16×16 block. Some are even stored in one byte only, namely those where a or b are taken from the previous block.

The remaining redundancy in the bit stream could be further compressed with Huffman codes [4]:

- The bit stream consist of three parts: The tags, the colors and the bit arrays. All three parts still have redundancy in the stream.
- Some of the tags are used very often, others never or very seldom. The usage of a Huffman code will reduce the total size of the tags to 50%.
- The colors can be compressed only by about 10% with a Huffman code because of the usage of a color lookup table. This table is chosen in a way so that all colors are used about equally often. Hence only a small redundancy of about 10% remains.
- A large redundancy is in the bit arrays, especially in the bit arrays of the 4×4-blocks. The redundancy can be as large as 60%.

Hence compression with three different Huffman codes would divide the size into half, but slow down the decompression.

From frame to frame colors are usually changed infrequently. So we could use a second "same color" flag to signal the reuse of the corresponding color in the last image.

A third improvement has been implemented: All blocks are test-wise subdivided into 4×4 blocks. Step 4 of the XCCC algorithms is then changed to the following:

4. If Δe is smaller than a given constant, and encoding of the actual block is better than encoding of the subdivided block, then Done.

4 Experiences: Compression ratio and decompression speed

The main goal of the XCCC scheme is to allow rapid decompression with software decoders. Table 1 shows the decompression speed in images per second. The measurements of MPEG [3] were done with the MPEG player of the University of California at Berkeley [8]. We used three movies: The first and second movie (butterfly) is a raytraced sequence of 350 frames sized 320×256 and 780×576 resp. The third movie was digitized from an analog video showing the University of Mannheim. It consists of scenes of buildings of the university (a palace) and other scenes depicting university life. Due to the analog origin, this movie consists of many different colors and color shadings. It is 320×256 in size and has 2000 frames.

The decompression was done on a DEC/alpha workstation with a 133Mhz CPU. The speeds are the real speeds viewed on the screen. The display adapter uses a color lookup table with 8 bits per pixel. XCCC uses the same technique internally thus requiring no conversion. In contrast, MPEG uses full color internally. Hence, the MPEG player has to dither in real-time. The player has several built in dithering methods. For the tests we used the fastest color dithering available ("ordered dithering").

Movie	Size	MPEG	XCCC
	(pixels^2)	(frames/s)	(frames/s)
Butterfly	320×240	10.5	42
Butterfly	780×576	2.1	6.8
Castle	320×240	7.8	24

Table 1. Decompression speed of software decoders (in frames/s)

Movie	Size	MPEG	JPEG	XCCC
Butterfly	320×240	0.80%$\hat{=}$0.19bpp	2.49%$\hat{=}$0.60bpp	3.0%$\hat{=}$0.72bpp
Butterfly	780×576	0.53%$\hat{=}$0.13bpp	1.54%$\hat{=}$0.37bpp	1.83%$\hat{=}$0.44bpp
Castle	320×240	1.5%$\hat{=}$0.36bpp	5.9%$\hat{=}$1.42bpp	6.3%$\hat{=}$1.51bpp

Table 2. Compression ratios (compressed size/full color size and bits per pixel)

The compression speed was measured on a DEC5000/133. For the small butterfly movie we got about 6.5 seconds per image with MPEG. Before XCCC can be started, the color lookup tables has to be computed with DeltaCLUT. This step needs about 7 seconds per image. XCCC then needs about 2 seconds per image. Together our compressor uses about 10 seconds per image.

Our experiments show that XCCC can decompress images very fast. The quality of the XCCC compressed images is comparable to the quality of MPEG compressed images. The great advantage of MPEG is the bit rate of the compressed movie. The size of XCCC compressed movies is about three to four times larger than the size of MPEG movies. Hence the domain of XCCC are local area networks with color workstations using the color lookup table technique. In this environment XCCC performs significantly better than MPEG.

5 Conclusions and Outlook

We have presented XCCC, an algorithm to decompress and play digital movies on standard color workstations at a reasonable speed without special hardware for the decompression. We have shown, that our algorithm is much faster in decompression than MPEG when implemented in software. On the other hand, MPEG gives a better compression ratio.

The next step will be experiments with Huffman tables for the three parts of the compressed data streams (tags, colors and bit arrays). We expect better compression, but we will have to pay the price of slower decompression.

The XCCC decompressor has been integrated into the XMovie system [6, 5], a test bed for the transmission and display of digital movies developed at the University of Mannheim. It is based entirely on standard hardware, and uses standard network technology and standard graphics adapters with color lookup tables.

References

1. G. Campbell, T. A. DeFanti, J. Frederikson, S. A. Joyce, A. L. Lawrence, J. A. Lindberg, and D. J. Sandin. Two Bit/Pixel Full Color Encoding. *Computer Graphics*, 1986.
2. E. J. Delp and O. R. Mitchell. Image Compression using Block Truncation Coding. *IEEE Transactions on Communications*, 1979.
3. D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, 1991.
4. D.A. Huffman. A method for the construction of minimum reduncancy codes. *Proceedings IRE*, 40:1098–1101, 1962.
5. R. Keller, W. Effelsberg, and B. Lamparter. Performance Bottlenecks in Digital Movie Systems. In D. Shepherd, editor, *4th International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, November 1993*, pages 163–174, 1993.
6. B. Lamparter and W. Effelsberg. X-MOVIE: Transmission and Presentation of Digital Movies under X. In R. G. Herrtwich, editor, *2nd International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, November 1991*, volume 614 of *Lecture Notes in Computer Science*, pages 328–339. Springer-Verlag Berlin Heidelberg, 1992.

7. B. Lamparter, W. Effelsberg, and N. Michl. MTP: A Movie Transmission Protocol for Multimedia Applications. In *Multimedia92, 4th IEEE ComSoc International Workshop on Multimedia Communications, Monterey, California*, pages 260–270, April 1992.
8. K. Patel, B. C. Smith, and L. A. Rowe. Performance of a Software MPEG Video Decoder. In P. Venkat Rangan, editor, *Proceedings of ACM Multimedia 93*, pages 75–82. Addison-Wesley, Aug 1993.
9. M. Pins. *Analysis and choice of algorithms for data compression with special remark on images and movies (In German).* PhD thesis, University of Karlsruhe, Germany, 1990.
10. J. U. Roy and N. M. Nasrabadi. Hierarchical Block Truncation Coding. *Optical Engineering*, 30(5):551–556, May 1991.
11. A. Urban. *ECCC - Implementation of extensions to the Color-Cell-Compression (In German)*, 1993.
12. G. K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):31–44, April 1991.