Color Distribution for Compression of Textural Data

Denis V. Ivanov, Yevgeniy P. Kuzmin
Mathematics Department, Moscow State University
Moscow, Russia

Abstract

Texture compression is recently one of the most important topics of 3D scene rendering because it allows visualization of more complicated high-resolution scenes. As standard image compression methods are not applicable to textures, an overview of specific techniques that are currently used in real 3D accelerators is presented. Because their major weakness is eventual image quality degradation, we introduce an approach that makes it possible to achieve superior image quality in most cases while providing higher compression ratios. Being an essential refining upon block decomposition strategy, it allows sharing the same color by several blocks, which partially solves the problem of colors deficit. We also describe some methods that can be used for conversion of textures to the proposed representation. Future prospects conclude the paper.

Keywords: *Texture, Texel, Compression, Block Decomposition*

1. INTRODUCTION

Reproducing the visual complexity of the real world is a challenge for many Computer Graphics practitioners. Since every detail cannot be represented on the geometric level, texturing techniques were designed to introduce complexity to synthetic scenes. For instance, textures can be anything from wood grain or marble patterns to detailed pictures of people, trees, buildings, etc.

To simulate real life scenes and render them in real time, it is desirable to have fast access to a large number of high-quality detailed textures. However, this requirement introduces significant demand of system or graphics memory depending on which is used for texture storage. Memory limitations in turn forces application developers to use fewer and less detailed texture maps.

The Accelerated Graphics Port (AGP) has made it possible to access textures stored directly in system memory increasing overall available storage capacity. However, AGP bus and system memory are shared resources. AGP is also used for uploading geometric data to the graphics accelerator, while the system memory services operating system and other applications. Therefore, it is a mistake to assume that the whole bandwidth is available for transferring textural data. In many cases, AGP bandwidth can be a bottleneck of the whole graphics system that limits users in gaining as much as they can from texturing techniques.

Texture compression allows the use of less memory for textures, or, which is more important, render scenes with more detailed high-quality textures without the necessity of buying additional memory units; besides, it lowers bandwidth requirements because compressed data may be passed to the graphics accelerator and then decoded on the other side of the bus. For these two reasons (more economical use of memory and significant reduction of bus traffic) texture compression is one of the hottest topics of GPU designs and graphics APIs.

Because texture compression techniques should comply with some specific requirements that ensure their efficient use in 3D engines including those implemented in hardware, we circumstantially discuss these requirements in the next section. We also give there a detailed overview of currently most exceptional techniques and discuss their possible outcome of significant perceptual quality degradation.

In Section 3 we present a new approach that solves the problem of the local colors deficit, which may arise while representing textures. This approach, previously introduced in [9], is based on the idea of sharing the same color by several blocks. Some methods that can be used for generating proposed color data are described in Section 4. The paper concludes with performance analysis of the new technique and discussion of possible future work with respect to the proposed approach.

2. EXISTING TECHNIQUES

Textures are regular raster images that are used for rendering surface-based 3D models. Due to growing popularity of visualization technologies, real-time rendering solutions have a strong tendency to be optimized for using as much of hardware as possible. For this reason, textural data decompression, which is one of the stages performed during rendering, should be very simple and efficient. On the other hand, considering standard rasterization approaches (such as polygon scan-line rasterization), texels may be fetched in random order, which strongly depends on the viewpoint. For the reasons above, the following requirements for texture compression techniques are introduced.

- High Compression Ratios
- No visible image degradation
- Fast (real time) data decoding
- Efficient random texel access

Obviously, if any compression technique uses variable length data encoding (including RLE, LZW, Huffman, arithmetic, etc. [5,11,13]), it is not applicable to texture compression, because it violates the requirement of random texel access. For example, RLE (run-length encoding) strategy encodes sequences of repeating elements into counter-element pairs; therefore, decoder has to analyze data stream from the very beginning in order to retrieve the required texel. As most of standard image compression formats are based on mentioned above algorithms, they could not efficiently be used for textures.

The existing texture compression approaches may be divided into two major groups: (1) vector quantization (VQ) and palletizing; (2) Block decomposition and block transforms.

2.1 Vector Quantization

VQ is based on the principle of look-up tables. For every texture, the VQ-coder constructs a codebook, which is an array of blocks

(typically 2x2 pixels) that appear most frequently in the corresponding image. When a codebook is generated, the whole image is divided into rows of blocks of the same size, and each block is substituted by the index of the most acceptable entry of the codebook. The typical size of the codebook is 128 or 256 entries. If blocks of 1x1 pixels in size are considered, than the compression procedure is called palletizing and the codebook is called palette.

However, VQ techniques suffer from two major problems. The first problem is memory access. If a decoder needs to extract any individual texel, it must first retrieve the index of the corresponding block, and then get the block colors from the codebook. This decoding procedure requires two serially dependent memory references per one texel, unless the whole codebook is stored on chip. The latter solution in its turn requires the codebook be uploaded to graphics accelerator before any decoding begins. Thus, none of these approaches provide acceptable compression ratios in terms of bus traffic.

The second problem of VQ technique is visual quality. VQ has a potential possibility of representing sharp edges (if codebook is large enough); however, it cannot represent smooth variations of colors very well, because in this case almost all blocks will differ from one another giving no possibility to group them on visual similarity basis.

For these two reasons (small reduction of actual bus traffic and poor quality on smooth surfaces) VQ is rarely used for texture compression.

2.2 Block Decomposition

The other group of techniques, which is called block decomposition, solves the problem of two separate memory calls per one texel. This approach is based on the idea of dividing an image into equally sized blocks (typically, 4x4 pixels) and storing each of them in a uniform manner, so that each block takes the same amount of memory after compression. Thus, all blocks may be stored row by row, and an offset of a block containing any individual texel may be easily calculated. The decoder has to retrieve the data of the block containing the required texel, decompress it, and extract the corresponding color.

This basic idea is used widely in most existing texture compression approaches. Recently, the following techniques based on block decomposition have been developed.

- Texture and Rendering Engine Compression (TREC) [15].
- S3 Texture Compression (S3TC) [14].
- 3dfx's texture compression (FXT1) [6].

2.2.1 Texture and Rendering Engine Compression

Texture and Rendering Engine Compression (TREC) was developed by Microsoft. This technique is very similar to the JPEG standard since it is based on the two-dimensional discrete cosine transform (DCT) of 8x8 pixel blocks and further quantization of coefficients. This approach provides variable compression ratios with satisfactory visual quality; however, it is relatively expensive to put hardware implemented DCT decoder on a graphics accelerator board only for texture decoding purposes.

2.2.2 S3 Texture Compression

The other technique, which was originally developed by S3 and then licensed by Microsoft for DirectX texture compression (DXT) [3], is very efficient, and therefore is implemented in several graphics accelerators, such as Savage2000, Voodoo 5 and 6 series, ATI RagePro. The simplest scheme encodes blocks of 4x4 texels. Each texel is represented by a 2 bit index of color from a local palette, which is generated for each block. The palette has 4 entries that are linearly interpolated from 2 RGB565 colors stored in the block (Figure 1).

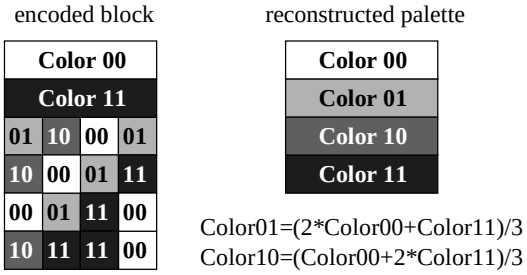

Figure 1: *S3TC block encoding scheme*

DirectX texture compression (DXT) has 5 variations of the S3TC scheme. These variations support transparency and alpha channel; however, none of them provides more then 2 original and 2 derived colors per each block.

2.2.3 3dfx's Texture Compression

The most recently developed texture compression technique was presented by 3dfx Interactive, Inc. and is called FXT1. This technique may be considered a powerful extension of S3 approach, since it separately encodes equally sized blocks of 4x8 texels by small local palettes (look-up tables). FXT1 has 4 modifications, each of which is specially designed to represent different color distributions within a block. However, the major idea remains the same: (1) some original colors (RGB555) are stored in a block; (2) local palette (or palettes) is generated by interpolation of provided colors; (3) each texel is represented by index of the most appropriate look-up table entry.

Table 1 briefly presents the FXT1 encoding parameters for different modifications. For *CC_HI* scheme, 2 original RGB555 colors, which are stored in a block, are used for interpolating 7 RGB8888 look-up table entries, while the 8th entry is defined transparent; thus, 3 bits per texel are required. *CC_CHROMA* refers to storing 4 colors, which are used with no change as the 4-entry palette. *CC_MIXED* is very similar to S3TC approach, since two 4-entry palettes are generated, each by interpolating between 2 colors (therefore, 4 colors are stored for a 4x8 block); then, two sub-blocks of 4x4 texels are encoded separately by their own palettes. For *CC_ALPHA*, two palettes of the same size are generated as well, but one of the colors out of 3, stored in a block, is shared during interpolation. This solution provides some extra space for storing alpha channel data in a block. While compressing, FXT1 encoder produces all representations of each 4x8 block of texels, chooses the one that introduces the least error, and stores the corresponding data in the resulting data stream.

FXT1 mode	Colors stored	Palette size	Bits/texel
CC_HI	2	8	3
CC_CHROMA	4	4	2
CC_MIXED	4	2x4	2
CC_ALPHA	3	2x4	2

Table 1: *FXT1 block encoding schemes*

Thus, S3TC represents each block of 4x4 texels by 8 bytes (4 for colors + 4 for indices), while FXT1 uses 16-byte data chunk for each 4x8 texel block. Compression ratio, provided by these techniques, is 6:1 for TrueColor 24bpp images, and 8:1 for 32bpp images, which is appropriate regarding texture compression requirements.

2.2.4 Quality Degradation on Compressed Textures

However, high compression ratio is archived by serious reduction of color data available for texels representation. For all described above schemes, excluding *CC_CHROMA*, generated local palette for each block consists of the colors belonging to a strait line in the RGB color space due to linear interpolation. For *CC_CHROMA*, 4 unique colors per 32 texels are used. Therefore, we can summarize that current techniques provide each block of 4x4 texels with 2 original colors, while others are linearly derived from them. This severe limitation may introduce perceptible quality degradation in some cases. For example, if a block comprises texels of at least 3 completely independent colors in RGB space (it could be pure red, green and blue), there is no way to reproduce them appropriately by interpolation of any two. Figure 2 validates this fact.

Figure 2: *Visual degradation of compressed images. Each flower is approximately 50x50 pixels. The original compression software was downloaded from S3 and 3dfx web sites, respectively.*

3. COLOR DISTRIBUTION

3.1 Color Distribution Basics

In order to provide more originally different colors on a block, we propose another approach to constructing a local palette it. Instead of interpolating colors from those stored in the reconstructing block, we propose to use colors stored in its neighbors. For example, we can put just one color in a block, and form a 4-entry palette by retrieving colors from the left, bottom and left-bottom adjacent blocks (Figure 3).

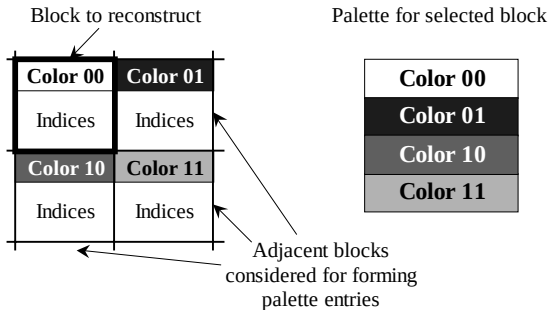

Figure 3: *Generating local palette from adjacent blocks*

Thus, the idea is to use colors from neighboring blocks instead of simple interpolation of colors stored in the currently reconstructing one. This approach allows, for example, encoding the image shown on Figure 2 with no color degradation at all, because it has only 4 unique colors.

Obviously, for constructing the local palette of a block we can choose any set of neighboring blocks, relative position of which may be defined by a pattern. Moreover, this pattern may even differ from block to block, and, in this case, should be stored for each of them. This strategy of color distribution is described in [9] with more details.

However, this new approach was originally developed as an efficient technique of texture compression. Therefore, if relatively distant blocks are involved to generate the palette, the problem of several memory calls, being similar to VQ, may arise. In order to simplify the decoding procedure and avoid the problem of several memory calls, we have chosen the simplest scheme, which provides 4 unique colors per a block. This scheme corresponds to Figure 3, and may be defined in more detail by the following.

3.2 Nodal scheme of Color Distribution

Let us subdivide a texture by a uniform grid having cells of 4x4 texels, and assign a color to every node of this grid. Thus, each block is defined by 4 corner nodes, which supply the decoder with 4 unique colors (Figure 4). As local palette has 4 entries, each texel can be indexed by 2-bit value which indicates the color of which corner should be taken.

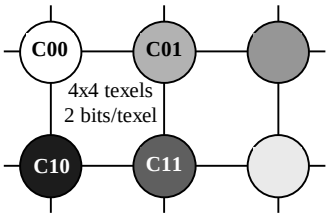

Figure 4: *Nodal scheme of color distribution*

As shown on Figure 4, each block has 4 corners providing 4 unique colors for decompression. On the other hand, each node is retrieved color from by 4 adjacent blocks, so the same color will be used in palettes of 4 blocks. This property significantly reduces the amount of memory required to store colors and indices. Indeed, the compressed texture has an equal number of blocks and nodes (we do not consider the last row and column here for simplicity), so each block takes

16 bits (RGB565) + 2*16 bits (indices) = 6 bytes,

which is 1/8 of uncompressed data considering TrueColor 24bpp images. Thus, providing compression ratios better than S3TC as well as FXT1, the color distribution approach allows reconstructing a larger number of unique colors on a block.

3.3 Decompression algorithm

Decompression of the data encoded by the described above technique can be implemented very efficiently. In fact, since color interpolation is not used, no arithmetic operations on colors are required. At this point, the proposed technique is even simpler than S3TC. However, color data is not as locally stored as it is done by standard block decomposition approaches, which may introduce a little more complex memory management. Pseudocode 1 presents the general strategy of decompressing images encoded by nodal scheme of color distribution. We assume here, that colors and block indices are stored as separate

2-dimentional arrays; however, interlaced storages may be considered as well.

Pseudocode 1: *Decompression algorithm for nodal scheme*

```
RGB565 GetTexel( int x, int y ) {
   block_x=x/4; block_y=y/4;
   index=GetBlockIndex(block_x,block_y);
   index=ExtractTexelIndex(index, x%4,y%4);
   if (index&1) block_x++; // shift left
   if (index&2) block_y++; // shift down
   return GetColor(block_x,block_y);
}
```

In Pseudocode 1, `GetBlockIndex()` and `GetColor()` functions are used for retrieving the index of the block and required color from the corresponding arrays, and `ExtractTexelIndex()` extracts 2-bit index of the corresponding texel from 32-bit index data of the whole block.

Since decoding typically takes place during 3D scene rendering, the data obtained by calling these functions is likely to be used for decompression of the next texel. For this reason, the memory cache (standard or specially designed) may be efficiently used for providing faster access to previously loaded indices and colors.

4. COMPRESSION ALGORITHMS

Compression of an image by proposed nodal scheme of color distribution appears to be much more complicated comparing to decompression. Obviously, the complexity of the compression algorithm and its time requirement are not critical parameters of entire technique; however, the visual quality of compressed images is of great importance. Since each node is involved in local palettes of 4 adjacent blocks, some variational methods are natural to be used for finding colors introducing minimal error.

In [9] we presented an algorithm that being relatively simple produces very good results on textures of various types. It does not reach the theoretical minimum of error, but the practical results proved to be quite sufficient. One of the advantages of the proposed algorithm is constant number of iterations that is equal to the number of blocks, since it sets up exactly one node by each iteration minimizing overall error as much as possible. On the other hand, nodes are assigned colors that are actually present in the original texture. This property is not required, but proved to result in compressed images of good visual quality.

In this paper we would like to formalize this algorithm, because it can be used not only for nodal scheme of color distributions, but also for any approach that looks up the color of a pixels in specially reconstructed local palette.

We also describe briefly the other solution, which is based on K-means clustering and is called Iterative Conditional Mode (ICM). Some aspects of its improvement and optimization are discussed.

4.1 Compression Algorithm

Let us introduce the following notations.

$X = \{x_i,\ i=1\ldots N\}$ denotes the original texture of N elements. We assume that perceptually uniform color space is used allowing us to measure distance between colors by Euclidean metric function. In practice, weighted RGB or $L^*u^*v^*$ ($L^*a^*b^*$) [4], which is more correct, can be used as adequate approximations of uniform color space.

$Y = \{y_i,\ i=1\ldots K\}$ denotes the global palette, i.e. the set of colors that are involved at least in one local palette. In case of nodal color distribution, Y would consist of all available nodes.

$R = \{r_{ij}\in\{0,1\},\ i=1\ldots N,\ j=1\ldots K\}$ is the set of rules that define which entries from the global palette Y should be considered as the local palette for a texel. Thus, for a texel x_i the local palette includes y_j if and only if $r_{ij}=1$. For the nodal scheme, each row of the matrix R would have exactly four units corresponding to block corners. In general, the property (4.1) should be fulfilled.

$$\forall i \sum_{j=1\ldots K} r_{ij} > 0\,. \tag{4.1}$$

$M = \{m_{ij}\in\{0,1\},\ i=1\ldots N,\ j=1\ldots K\}$ is the assignment matrix. It has exactly one unit in each row. Position of this unit specifies which color from a global palette is currently used for representing the corresponding texel. Because, in addition to the palette we defined a set of rules, the assignment matrix M should comply with (4.2), which is

$$\forall i \sum_{j=1\ldots K} r_{ij} m_{ij} = 1\,. \tag{4.2}$$

Thus, the formalized problem of texture compression is the following. Given an arbitrary texture X and a set of rules R, the objective is to find the global palette Y and the corresponding assignment matrix M, such that the error between the compressed image

$$X^* = RM \times Y, \tag{4.3}$$

and the original image X is minimal (the symbol × denotes here a standard matrix multiplication, while RM means 'per element' multiplication resulting in matrix of the same size).

It is well known that, due to physiological nature of human vision, the color perceived in each pixel depends not only on its original colors, but also on colors of neighboring pixels, as well. At this point, some complicated models based on Gaussian kernel as spatial support of each pixel can be considered. This strategy is extensively described in [2,8,10,7]. However, for the sake of simplicity we consider the straightforward Euclidean distance in approximately uniform color space as the error of compression. Thus, this error is defined by (4.4).

$$E(X, X^*) = \sum_{i=1\ldots N} \left\| x_i - \sum_{j=1\ldots K} r_{ij} m_{ij} y_j \right\|^2 \tag{4.4}$$

As was introduced previously, the proposed algorithm defines exactly one palette entry in each iteration; thus, let us assign a boolean value to each entry specifying whether it has been already set up or not. Denoting this values with

$S = \{s_j\in\{0,1\},\ j=1\ldots K\}$, we state that before the algorithm begins all s_j are set to 0, and in the end, they are all equal to 1.

$E = \{\varepsilon_i,\ i=1\ldots N\}$ is the set of errors calculated for each texel regarding only set-up palette entries. Each error is defined by (4.5), and in the beginning is set to the maximum possible value (we consider finite color space here).

$$\varepsilon_i = \begin{cases} \min\limits_{j:s_j r_{ij}=1} \left\| x_i - y_j \right\|^2, \text{if } \exists j : s_j r_{ij} = 1 \\ MAX\_VALUE, \text{otherwise} \end{cases} \tag{4.5}$$

For each palette entry that is not yet set up to the final value, we find the texel from the corresponding area, such that if the color of this texel were assigned to the palette entry, the overall error

would maximally decrease. The position of this texel for y_j is obtained by (4.6)-(4.7).

$$l = \max_{i:r_{ij}=1} \sum_{k=1\ldots N} r_{kj}\chi\left(\left\|x_i - x_k\right\|^2 - \varepsilon_k\right), \text{ where} \quad (4.6)$$

$$\chi(x) = \begin{cases} x, if\ \ x > 0 \\ 0, otherwise \end{cases}. \quad (4.7)$$

And, finally, the error decrease that would occur if the node y_j were set-up with the color x_l is defined by (4.8) in each palette entry.

$$d\varepsilon_j = \sum_{k=1\ldots N} r_{kj}\chi\left(\left\|x_l - x_k\right\|^2 - \varepsilon_k\right) \quad (4.8)$$

The general idea of the algorithm is to find on each step the node (palette entry) that will maximally decrease overall error E if set up. This entry is considered defined, and all others (only those that could have changes by this assignment) should be recalculated to reflect the changes.

This idea is formalized in Pseudocode 2.

Pseudocode 2: *Iterative Compression Algorithm*

```
M,Y CompressTexture( X,R ) {
   set s_j (j=1…K) to 0;
   set ε_i (i=1…N) according to (4.5);
   set y_j=x_l (j=1…K), where l complies (4.6);
   set dε_j (j=1…K) according to (4.8);
   while (not all s_i are 1) {
      m=arg max_{j=1…K}(s_j dε_j);
      s_m=1;
      dε_m=0;
      for (i:r_im=1) recalculate ε_i by (4.5);
      for (j:∃i:r_im r_ij=1) {
         if (dε_j>0) continue;
         set y_j=x_l, where l complies (4.6);
         set dε_j according to (4.8);
      }
   }
   // Y is defined, let us find M
   m_ij=1 if y_j minimizes ε_i, m_ij=0 otherwise
   return M,Y;
}
```

The strategy presented in Pseudocode 2 is relatively simple and produces very good results on different types of textures. In fact, recalculations of ε_i and $d\varepsilon_i$ are required for a small number of elements affected by changes made in a particular step; therefore, this algorithm is one of the fastest among other iterative algorithms of this class.

4.1.1 Preliminary Clustering

However, considering proposed nodal scheme of color distribution, the speed of this algorithm can be further improved by preliminary clustering of the colors within each block. Because proper clustering is a complicated task in general case, we propose to use a very simple algorithm, which produces sufficient results and works very fast considering that each time it needs to cluster 16 points. Description of this algorithm follows.

Let us consider a set of points in a metric space, and denote it with X. The task is to find clusters not exceeding *d* in diameter and comprising all points from X. The number of clusters, obviously, should be as little as possible. At each step, the algorithm finds the diameter [*xy*] of X. Then, it forms a cluster around *x* with diameter *d* and removes covered points from X. The same procedure is also applied to *y* unless distance between *x* and *y* is less then *d*. We should point out, that this procedure has some assumptions, and does not construct a minimal set of clusters; however, in practice it works very well, and, what is more important, it is relatively fast.

Thus, clusters can be treated in the same way as texels in Pseudocode 2. The only difference would be storing the number of texels belonging to each cluster, since this information is required for proper calculation of ε_i and $d\varepsilon_i$. In practice, this approach of preliminary clustering speeds up the iterative algorithm by 2-5 times, because the number of performed arithmetic operations is significantly reduced.

4.1.2 Preprocessing

To make algorithm even faster, some special cases may be determined before iterations start. Let us introduce two of these cases:

- If all blocks adjacent to a node has clusters representing one color (similar colors in term of visual difference), this color is the only choice for this node;
- If all clusters corresponding to a connected area of blocks represent, in fact, equal or less than 4 colors, than all nodes of this area may be set with these colors in a chess-board order.

The above-mentioned cases may be validated and processed in a proper manner after clustering, but before calculating errors. This procedure excludes some of the nodes from further consideration generally reducing the overall time of conversion.

4.2 Iterative Conditional Mode

The proposed above compression algorithm fills global palette with colors taken from the original image. This property generally does not introduce any perceived artifacts; moreover, in some cases this approach produces better images from human vision point of view (for example, on sharp edges). However, we can tune standard approaches to serve our needs. One of these approaches is so called ICM (Iterative Conditional Mode), which can be derived from [1,2]. ICM is, in fact, conceptually similar to K-means clustering, which iterates through color space converging to the minimum of error function $E(X,X^*)$.

In our notations, if the global palette Y is defined, the assignment matrix M should comply with (4.9) to minimize overall error for a given palette.

$$m_{ij} = \begin{cases} 1, if\ \ j = \arg\min_{k:r_{ik}=1} \left\|x_i - y_k\right\|^2 \\ 0, otherwise \end{cases} \quad (4.9)$$

On the other hand, if the assignment matrix M is fixed, palette entries may be obtained by (4.10).

$$y_j = \frac{\sum_{i=1\ldots N} m_{ij} x_i}{\sum_{i=1\ldots N} m_{ij}} \quad (4.10)$$

Thus, the ICM-like algorithm alternately finds the assignment matrix for the fixed palette, and calculates palette colors as the median of the set of pixels that are indexed by this palette entry. This strategy is expressed by Pseudocode 3.

Pseudocode 3: *ICM – Iterative Conditional Mode*

```
M,Y CompressTexture( X,R ) {
   set y_j (j=1…K) to arbitrary colors;
   do {
      set m_ij (i=1…N,j=1…K) by (4.9);
      set y_j (j=1…K) by (4.10);
   } until (converged);
   return M,Y;
}
```

The ICM algorithm is quite efficient; however, it is well known that it gets frequently stuck in local minima. Therefore, some special steps should be taken to solve this problem. For example, multiscale (deterministic) annealing [10,12] can be considered.

The complexity algorithm can be significantly improved by multiscale optimization, described in [7].

5. CONCLUSION

In this paper we introduced a number of requirements that should be fulfilled by texture compression technique for its efficient use in real-time systems of 3D scene rendering. We analyzed the most exceptional approaches, currently invented, including S3TC and FXT1. These approaches proved to be very efficient; however, they may introduce perceivable quality degradation due to the lack of colors available for decompressing each block of texels.

In order to provide more originally different colors to each block, we proposed a general technique of color distribution, and, in particular, the nodal scheme being the simplest one in terms of memory management, which is critical for real-time hardware-implemented algorithms.

The proposed technique perfectly complies with all introduced requirements. Thus, compression ratio is superior to the one obtained by S3TC and FXT1, and is equal to 8:1 for TrueColor 24bpp textures. Visual quality of compressed images is better in most cases, because more independent colors are available for each block. Decompression algorithm is very simple and can be efficiently implemented in hardware. Finally, because color distribution is an improvement upon block decomposition approach, random access to the texture elements is naturally supported.

Because, according to the proposed approach, color data are shared by several blocks of texture, finding optimal palette that minimizes overall error appears to be a relatively complicated task. We introduced an algorithm that being simple and efficient produces quite sufficient approximations to the optimal solution. Some optimizations, such as preliminary clustering and preprocessing, are discussed.

We also described in brief the modification of standard ICM approach for obtaining distributed colors, and specified some possible improvements for preventing its getting stuck in local minima.

In conclusion, the proposed approach of color distribution can be efficiently used for compressing images of any types, while nodal scheme, being specially designed for texture compression, is a valuable technique that allows real-time rendering of 3D scenes with much higher level of details.

6. FUTURE WORK

We would propose two directions of further investigations in the scope of color distribution approach to image compression.

As was mentioned above, this approach may be used not only for compressing textural data, but also for regular image compression. In addition to nodal scheme, introduced in this paper, there could exist some other strategies that may serve different application. Careful investigation of these strategies seems to be very interesting and, possibly, worth developing.

For the other topic of research we would propose careful analysis of compression algorithms. More correct models of human vision, such as Gaussian kernel in a pixel [2], can be implemented. Some properties of the proposed iterative algorithm may be investigated. ICM-like approach may also be carefully studied and improved by multiscale annealing and optimization. All these steps are likely to improve obtained visual quality of compressed images.

7. ACKNOWLEDGEMENTS

The presented here techniques were developed by the Computer Graphics Group (Dept. of Math, Moscow State University) in accordance with the Research Agreement between Department of Mathematics and Mechanics of Moscow State University and Intel Technologies, Inc. We would like to thank Jim Hurley and Alexander Reshetov (Intel Technologies, Inc.) for their constant attention to this project as well as their constructive criticism.

8. REFERENCES

1. J. Besag. On the statistical analysis of dirty pictures. Journal of the Royal Statistical Society, Series B, vol. 48, 1986.
2. J.M. Buhmann, D.W. Fellner, M. Held, J. Ketterer, and J. Puzicha. Dithered Color Quantization. Proceedings of EUROGRAPHICS, vol.17, no.3, 1998.
3. Compressed Texture Formats. Microsoft DirectX 7.0, Platform SDK, MSDN. – Microsoft, 1999.
4. C.I. de L'Eclairage. Colorimetry. CIE Pub. 15.2 2nd ed., 1986.
5. Wolfgang Effelsberg, et al. Video Compression Techniques. Morgan Kauffman Publishers, 1998.
6. FXT1: White Paper. – 3dfx Interactive, Inc., 1999.
7. F.Heitz, P.Perez, and P. Bouthemy. *Multiscale minimization of global energy functions in some visual recovery problems.* CVGIP: Image Understanding, vol. 59, no.1, 1994.
8. T. Flohr, B. Kolpatzik, R. Balasubramanian, D. Carrara, C. Bouman, and J. Allebach. Model based color image quantization. Proceedings of the SPIE: Human Vision, Visual Processing, and Digital Display IV (J. Allebach and B. Rogowitz, eds.), vol. 1913, 1993.
9. Denis Ivanov, Yevgeniy Kuzmin. Color Distribution – a new approach to texture compression. Proceedings of EuroGraphics' 2000.
10. J. Ketterer, J. Puzicha, M. Held, M. Fischer, J.M. Buhmann, and D. Fellner. On spatial quantization of color images. Proceedings of the European Conference on Computer Vision, 1998.

11. Mark Nelson, Jean-Loup Gailly. The Data Compression Book. IDG Books Worldwide, 1995.
12. J. Puzicha, M. Held, J. Ketterer, J. Buhmann, D. Fellner. On Spatial Quantization of Color Images. Technical report IAI-TR-98-1, University of Bonn, 1998.
13. Khalid Sayood. Introduction to Data Compression. Morgan Kauffman Publishers, 1996.
14. S3TC: White Paper. – S3, Inc., 1998.
15. TREC: White paper. – Microsoft Corporation, 1998.

About the author

Denis V. Ivanov, Ph.D. student – Denis@fit.com.ru
Dr. Yevgeniy P. Kuzmin, Senior Scientist – Yevgeniy@fit.com.ru

Computational Methods Lab.
Mathematics and Mechanics Dept.
Moscow State University,
Vorobyovy Gory, Moscow, Russia, 119899