EUROGRAPHICS 2000 / M. Gross and F.R.A. Hopgood
(Guest Editors)

*Volume 19 (2000), Number 3*

# Color Distribution – a new approach to texture compression

D. Ivanov and Ye. Kuzmin

Department of Mathematics and Mechanics, Moscow State University, Moscow, Russia

**Abstract**
*Texture compression is recently one of the most important topics of 3D scene rendering techniques, because it allows rendering more complicated high-resolution scenes. However, because of some special requirements for these type of techniques, the commonly used block decomposition approach may introduce visual degradation of image details due to lack of colors. We present here a new approach to texture compression, which allows sharing of one color by several blocks providing a larger number of unique colors in each particular block and the best compression ratio. We also present an iterative algorithm for obtaining distributed colors on a texture, and discuss some advantages of our approach. The paper concludes with comparison of our technique with S3TC and other block decomposition methods.*

## 1. Introduction

Reproducing the visual complexity of the real world is a real 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. However, AGP bus and system memory are shared resources. AGP is also used for passing geometric data to the graphics accelerator, while the system memory services the operating system and other applications. Therefore, it is a mistake to assume that all bandwidth is available for transferring texture 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 texture compression is one of the hottest topics of GPU designs and graphics APIs.

Although textures are ordinary raster images used for 3D scene rendering, common techniques of image compression cannot be efficiently used for them. The major problem of ordinary techniques is dynamic encoding, such as RLE, LZW, Huffman[1, 2, 3], etc. Compression ratio provided by these techniques depends on the particular image characteristics (RLE encodes sequences of repeating elements into number-element pairs; LZW constructs a codebook and substitutes matching words by references in it; Huffman and other entropy coding techniques rely on elements frequency histogram); consequently, the length of compressed data varies from image to image. Therefore, arbitrary pixels cannot be decoded unless a significant amount of preceding data is retrieved and decoded as well. For example, to extract any arbitrary portion of an RLE-compressed data, the decoder has to process the whole data stream from the very beginning up to the desired part. Thus, these standard approaches are not useful for random access to individual texels. To eliminate this problem, special compression techniques were developed. Section 2 of this paper discusses recent technologies used for texture compression in existing graphics accelerators.

However, the major issue of all texture compression approaches is an adequate balance between visual quality and compression ratio. Trying to obtain high ratios with many existing techniques may produce images with significant degradation in visual quality.

This paper presents a new and efficient technique for texture compression, which has better quality in most cases while providing the best compression ratio among existing approaches. In Section 3 we present the general idea of our technique and discuss it's advantages over other approaches. Section 4 describes an algorithm that we used to compress raster images and some enhanced strategies that we considered in order to speed up the process. The paper concludes with a comparison of the proposed technique to other methods.

## 2. Existing texture compression techniques

Textures are typically used for rendering of 3D scenes at high frame rates; therefore, texture compression formats should comply with the following requirements:

- High compression ratios
- No visible image degradation
- Fast (real time) data decoding
- Efficient random access (texture lookup)

The last two requirements are of high importance for texture encoding techniques because texels are usually fetched randomly during rendering and they should be decoded as fast as possible to provide appropriate fill-rates.

All existing texture compression approaches may be divided into two major groups: (1) vector quantization (VQ) and palletizing[2]; (2) Block decomposition and block transforms[3].

VQ is based on the principle of look-up tables. For every texture, the VQ-coder constructs a codebook, which is a table of blocks (typically $2 \times 2$ pixels) that appear most frequently in the corresponding image. Each blocks is then 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, then 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 the 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 procedure requires two serially dependant 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 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 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.

The other group of techniques, which is called block decomposition, solves the problem of two separate memory calls per one texel. This approach uses the idea of dividing an image into equal blocks (typically, 4x4 pixels) and storing each one 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 a block containing the required texel, and simply extract its color.

This basic idea is used widely in most existing texture compression techniques. Recently, the following schemes of block decomposition have appeared.

- Texture and Rendering Engine Compression (TREC)[4]
- S3 Texture Compression (S3TC)[5]
- 3dfx's texture compression (FXT1)[6]

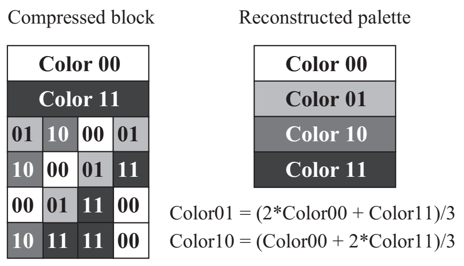


**Figure 1:** *S3TC Scheme.*

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.

The other technique, originally developed by S3 (S3TC) and then used by Microsoft in their DirectX texture compression (DXT)[7], is very efficient, and therefore is implemented in S3's new graphics solution Savage2000. It operates with blocks of 4x4 pixels. Each texel is represented by a 2 bit index of color from a local palette, which is generated for each

| FXT1 scheme | Block size | Bits/ texel | Colors stored | Palette size |
|---|---|---|---|---|
| CC_MIXED | $4 \times 4$ | 2 | 2 | 4 |
| CC_HI | $4 \times 8$ | 3 | 2 | 8 |
| CC_CHROMA | $4 \times 8$ | 2 | 4 | 4 |
| CC_ALPHA | $4 \times 8$ | 2 | 3 | 4 |

**Table 1:** *FXT1 compression schemes*

block. The palette has 4 entries, which are interpolated from 2 RGB565 colors stored in the block (Figure 1).

Because the colors are stored in RGB565 format, the palette information requires 32 bits per each block. Indices take another 32 bits per block. Thus, the compressed block requires 8 bytes, while the original takes 48 considering RGB888. Consequently, the S3TC scheme provides 6:1 compression ratio.

DirectX texture compression (DXT) may be considered as an extension of S3TC. DXT has five various schemes that allow to code not only opaque textures, but also transparency and alpha channels.

FXT1 technique was developed by 3dfx. It uses the same principle and has 4 modifications (Table 1). The first scheme CC_MIXED is the same as S3TC. The others compress blocks of 4x8 pixels. Local palettes can consist of 4 or 8 entries, which are interpolated or stored directly in a block as it is done for the CC_CHROMA scheme.

Generally, FXT1 is more flexible than DXT; therefore, it provides slightly better results in terms of visual quality. The average compression ratio is 6:1 for 24-bit textures and 8:1 for 32-bit textures. The decoder for FXT1 as well as for DXT is implemented in hardware by 3dfx in their new generation of Voodoo accelerators.

## 3. Color Distribution Basics

The major problem of S3TC and FXT1 is lack of colors on each separate block. For example, S3's texture compression format provides two unique colors for texels of a $4 \times 4$ pixels block. The other two colors are linearly interpolated from them; however, that is not enough for many types of images.

Let us consider an image having three colors that do not belong to one line in the RGB cube. For instance, it can be red, green and blue. If pixels of these colors appear in one block simultaneously, S3's compression scheme will not be able to represent them without visual degradation. It will either discard the less frequent color or construct a line (in the RGB cube) representing minimum distance from all existing points. In some cases disregarding less frequent colors may not result in visible distortion of the original image; however, it often introduces perceptible effects as shown on Figure 2.

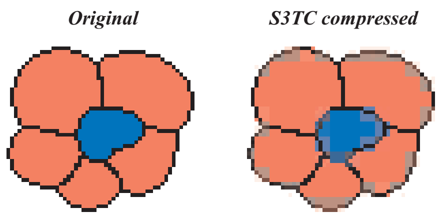


**Figure 2:** *Visual degradation of S3TC. (Both images are zoomed from 64x64 pixels size).*

### 3.1. Our proposal in brief

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

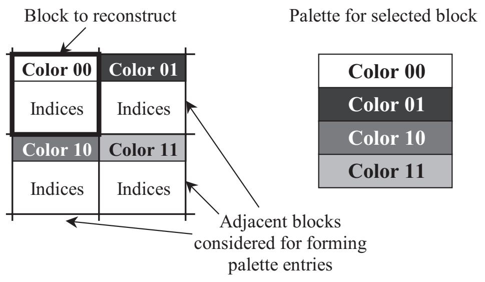


**Figure 3:** *Forming palette from adjacent blocks.*

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, one to represent the image shown on Figure 2 with no color degradation at all, because this image has only 4 unique colors.

### 3.2. Formal definition of our approach

Let us introduce some notations in order to specify the proposed approach more formally.

**Local Palette (LP)** is a 4 or 8-entry palette, which is constructed for each block and used to substitute indices by real colors.

**Local Surrounding** of a block B is a square of blocks (typically, $2 \times 2$ or $3 \times 3$), which includes B. Blocks from Local Surrounding are used for generating local palettes.

**Local Palette (LP) Pattern** is a matrix (a pattern), which has the same size as Local Surrounding. Non-zero elements of this matrix indicate that corresponding neighbor blocks should be used for generating a Local Palette.

Figure 4 shows some examples of LP patterns that may be used for block reconstruction. The pattern (a) corresponds to a scheme presented in Figure 3.

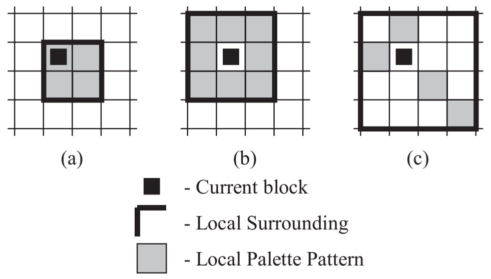


**Figure 4:** *Examples of Local Palette Patterns.*

It should be noted, that LP Pattern may be the same for all blocks of an image; however, it also may differ from block to block and, in this case, it should be stored for each particular block.

Regarding all aforementioned notations, we can specify a data format for our approach to texture compression. Each block of texels is represented by the following elements:

- Color (RGB565 for TrueColor)
- Local Palette Pattern (only if required)
- Indices (2 or 3 bits per texel)

In general, a color may be stored in any format, which is required for a particular task. The Local Palette Pattern is required in each block only if its structure is not the same for the whole texture. Indices are 2 or 3 bits per texel depending on the palette size.

The texture reconstruction procedure is the same as for other block decomposition methods. The Local Palette should be generated for a block. In order to do it, the decoder could simply get colors from the Local Surrounding judging from the LP Pattern. When a palette is obtained, decoder can reconstruct color of each texel.

### 3.3. Color Distribution (CD) Scheme

Storing the Local Palette Pattern gives us more flexibility in providing better matching colors on a block. However, retrieving colors from a Local Surrounding is a task very similar to extracting a block of colors from a codebook during VQ compression. Required colors may lie in random order, and generating palette may take several memory calls.

In order to simplify the decoding procedure and avoid the problem of several memory calls for generating the Local Palette, 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 details by the following.

Let us subdivide a texture by a uniform grid having cells of 4x4 texels size, and let us consider that every node of this grid contains one color. Each cell is represented by 4 nodes, which supply decoder with 4 colors (Figure 5). Each texel, therefore, may be indexed by 2-bit value which indicates from which corner we should take the color.

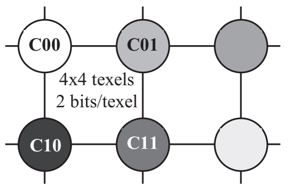


**Figure 5:** *Color Distribution Scheme.*

As shown on Figure 5, each block has 4 corners each of which holds one color for decompression. On the other hand, each node corresponds to 4 adjacent blocks, so the same color will be included 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

$$16\text{bits (RGB565)} + 2 \times 16 \text{ bits (indices)} = 6 \text{ bytes},$$

which is 8:1 of uncompressed data. Thus, providing compression ratios better than S3TC and FXT1, the Color Distribution Technique allows operation with a larger number of unique colors on one block.

### 3.4. Decompression Algorithm

This section present an algorithm for extracting one texel from a texture compressed with CD Technique. We assume here, that colors and block indices are stored as separate 2-dimentional arrays; however, interlaced storages may also be considered.

```
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++; //index is 1 or 3
  if (index&2) block_y++; //index is 2 or 4
  return GetColor(block_x, block_y);
}
```

The above pseudo-code uses GetBlockIndex() and GetColor() functions to extract the index of the corresponding block and required color from arrays, respectively. Since decoding typically takes place during 3D scene rendering, 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 Algorithm

If we consider Color Distribution Scheme from color quantization (palletizing) point of view, we may well suppose that our goal is to generate a palette of $N/16$ entries ($N$ denotes the number of texels in a texture). However, in addition to general quantizing approaches[8, 9, 10], we should apply special restrictions to this palette. These restrictions correspond to node-texel relationship, which makes only 4 entries of a palette available for substituting texels of each block.

In this section we present an iterative algorithm, which explicitly preserves aforementioned restrictions, and on the other hand, inherits some ideas of Iterative Conditional Mode (ICM) algorithms[11, 12, 13, 14].

We consider here the simple model of human vision, and assume that color, perceived in each texel, only depends on the value of this texel; however, more accurate and complicated models may be considered[13, 14] with some modifications.

In order to simplify the description of the algorithm we present it for the CD Scheme; however, it may be generalized for any scheme in terms of Section 3.2 without significant modifications.

### 4.1. Iterative algorithm of Color Distribution

The idea of the algorithm is based on an attempt to minimize the average degradation of each texel, i.e. put color in each node so that the distance from each texel (in color metric space) to the nearest surrounding node will be minimal. Although, solving this problem accurately seems to be very sophisticated, we propose very simple algorithm, which gives sufficient approximation.

#### 4.1.1. Color Metric Space

We have chosen 2 color spaces for our experiments. RGB space with color distance being weighted sum of component differences is very simple, but it is not perceptually uniform. For this reason, we also considered CIE $L^*u^*v^*$[15], which is more complicated for computations, but more accurate from human vision point of view. However, our tests showed that use of $L^*u^*v^*$ makes conversion process slower, while no significant color degradation is introduced.

#### 4.1.2. Data structures

The algorithm uses the following structures, which stand for individual texel and node, respectively.

```
struct TEXEL {
   COLOR Color; // CIE or RGB
   float Error; // Approximation error
};
struct NODE {
   COLOR Color;    // CIE or RGB
   float Priority; // Node priority
   BOOL SetUp;     // Set up flag
};
```

#### 4.1.3. Brief description of the algorithm

Before the algorithm begins, all nodes are marked as not se tup, and all texel errors are set to the maximum possible. For each node, all adjacent blocks are inspected. For these blocks, the algorithm finds the color which would maximize the average texel error decrease if this color were placed in a node. This color is then placed in a node and the priority is set to the average error decrease.

The node priority indicates how the average texel error would decrease if the corresponding color is considered to be the final one for the compressed data. It is obvious that the algorithm should decrease the average error as much as possible, so it finds a node with maximum priority (considering non se tup nodes only) and marks it as set up This color will be used for the local palette of adjacent blocks.

Figure 6(c) illutrates priority map for weii-known Mandrill image. Highest priorities indicates which node will maximally decrease an average error if it set up. Colors obtained on the last stage of iterative procedure are shown at Figure 6(d)

Because we set up a node, the texel errors in adjacent blocks should be recalculated to reflect the changes. This procedure leads to possible changing of colors and priorities in the adjacent nodes, and the algorithm should recalculate them, too.

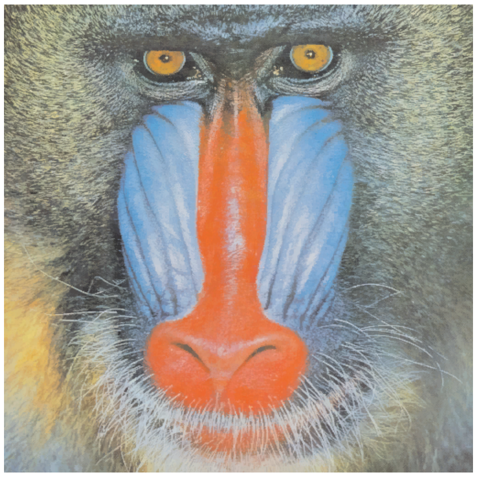
The procedure of setting up a node with maximum priority, decreasing texel errors, and recalculating the influenced nodes continues until not all nodes are set up. When every node has a proper color in it, we just need to substitute each texel by the index of the best matching color in the Local Palette.

#### 4.1.4. Texel error calculation

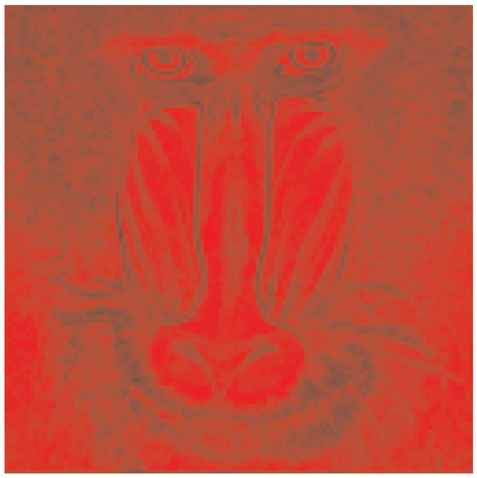
The error of each texel indicates the accuracy of the approximation currently available in the Local Palette. In the very beginning, no colors are set up in nodes, so the error is the maximum possible (we assume it is 256.0 for RGB).

```
Texel.Error=256.0;
for (each Node from AdjacentNodes(Texel)){
    if (Not Node.SetUp) continue;
    Dist=ColorDist(Texel.Color, Node.Color);
    if (Dist < Texel.Error) Texel.Error=Dist;
}
```

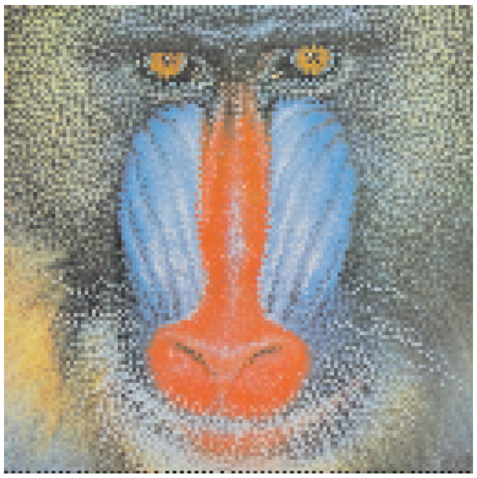
a) Original image (512x512 pixels)



b) Compressed with Color Distribution (8:1)



c) Starting priorities of nodes (higher intensity corresponds to higher priority). Node of highest intensity will be set-up first.



d) Node colors obtained at the end of the compression (4x4 blocks are filled with top-left node colors).

**Figure 6:** *Well-known mandrill image compressed 8:1, map of priorities and distributed colors.*

Thus, the minimal distances to the colors available in adjacent setup blocks are stored as texel error.

### 4.1.5. Node priority and color calculation

The priority indicates how the average error would decrease if we marked the node as "set up". It is calculated by the following procedure.

```
Node.Priority=0.0; // Minimal value
for (each Texel from AdjacentBlocks(Node)){
  ErrorDecrease=0.0; // Decrease accumulator
  for(each Texel1 from AdjacentBlocks(Node)){
    Dist=ColorDist(Texel.Color, Texel1.Color);
    if (Dist < Texel1.Error)
      ErrorDecrease+=Texel1.Error - Dist;
  }
  if (ErrorDecrease>Node.Priority){
    Node.Color = Texel.Color;
    Node.Priority = ErrorDecrease;
  }
}
```

Thus, all the colors available in adjacent blocks are considered, the one providing maximum error decrease is chosen and placed in the node. The priority is then set to the error decrease.

#### 4.1.6. Nodes set up algorithm

As was described in Section 4.1.3, the algorithm sets up one node at a time judging from its priority. More formally,

```
// Initialization
for (all Texels) Texel.Error = 256.0;
for (all Nodes) CalculatePriority(Node);
// main loop
while (Not all Nodes are set up) {
  Node = NodeWithMaxPriority();
  Node.SetUp = True;
  for (each Texel from AdjacentBlocks(Node))
       CalculateError(Texel);
  for (each Node1 sharing blocks with Node)
       if (Not Node1.SetUp)
          CalculatePriority(Node1);
}
```

When all the blocks are set up, the local palettes for each block are available, and we can substitute the real colors by indices. This operation concludes the whole process of texture compression.

### 4.2. Texel clustering

The above algorithm inspects all texels of a block every time it needs to recalculate errors or node priorities. Moreover, during the calculation of node priority, distances for all existing texel pairs are required. This large number of operations may be significantly decreased by clustering of the block texels in advance. If we define the maximum cluster size so that there would be no visual difference between points of one cluster, we can consider generated clusters instead of single texels for the iterative algorithm of CD compression.

In the general case, proper clustering is a very complicated task. However, 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 M. The task is to find clusters not exceeding D in diameter and holding all points from M. The number of clusters, obviously, should be as little as possible. At each step, our algorithm finds the diameter [XY] of M. Then, it forms a cluster around X with diameter D and deletes the obtained points from M. The same procedure is also applied to Y unless distance from X to 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, the algorithm of color setup works with clusters as if they were texels, except it should know how many texels belong to one cluster. This information is required for correct calculation of cluster errors and node priorities.

In practice, substituting texels with clusters makes the speed of the compression algorithm 2-5 times faster introducing no visual difference into the resulting image.

### 4.3. Preprocessing

Obviously, in some cases the decision as to what color should be placed in a node may be made without calculating priorities. Let us consider 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 represents, in fact, equal or less than 4 colors, than all nodes of this area may be set with the obtained colors in a chess-board order.

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

## 5. S3TC vs. Color Distribution

Let us summarize some advantages of our technique, called Color Distribution, in comparison to S3TC, which is the most widely used texture compression standard.

- **Compression Ratio**. CD provides 8:1 compression ratio for RGB textures while images compressed using any scheme mentioned in Section 2, including S3TC, takes 6:1 of the original size.
- **Visual Quality**. CD provides 4 unique colors for a local palette of each block, which gives it more power on multi-colored images. S3TC and other standards generally interpolate palette entries from 2 original colors stored in a block.
- **Fast decompression**. In the general case, if we need to extract just one texel from a separate block, we have to make 3 memory calls, one for indices and two others for extracting colors (2 per a call). S3TC requires 2 calls per block. However, Color Distribution uses one color for 4 adjacent blocks, so the memory cache may be used very efficiently, and we expect that bus traffic will be near the same for both schemes.

## 6. Conclusion

In conclusion, we have presented a new approach to texture compression, which is, in fact, a powerful enhancement of block decomposition methods. The major advantage of our approach over ordinary ones is the sharing of one color by more than one neighboring block. This technique allows storage of less color data over the whole image, while providing larger number of unique color entries for each particular block. Therefore, a higher compression ratio is obtained and some image degradation problems are eliminated.

We have also presented a greedy algorithm for compressing RGB images into the proposed standard called Color Distribution. Clustering and preprocessing are considered as efficient techniques to speed up compression while introducing no visual difference in the final image.

Color Distribution is perfectly designed for graphics accelerators and may be easily implemented in hardware, because it has very simple decompression algorithm. Our research showed that it seems to be a very valuable feature for standard rendering pipelines.

## Acknowledgements

This work was performed by the Computer Graphics Group (Dept. of Math, Moscow State University) as a part of the research program conducted 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 careful reading and constructive criticism. We also acknowledge anonymous reviewers for their generous comments and suggestions.

## References

1. Khalid Sayood. *Introduction to Data Compression*. Morgan Kauffman Publishers, 1996. 1

2. Wolfgang Effelsberg, et al. *Video Compression Techniques*. Morgan Kauffman Publishers, 1998. 1, 2

3. Mark Nelson, Jean-Loup Gailly. *The Data Compression Book*. IDG Books Worldwide, 1995.

4. TREC: White paper. Microsoft Corporation. *http://www.microsoft.com/hwdev/devdes/whntrec.htm* 1, 2 2

5. S3TC: White paper. S3, Inc. *http://www.s3.com/savage3d/s3tc.pdf* 2

6. FXT1: White paper. 3dfx Interactive, Inc. *http://www-dev.3dfx.com/fxt1/fxt1whitepaper.pdf* 2

7. Compressed Texture Formats. Microsoft DirectX 7.0. Platform SDK, MSDN, October 1999. 2

8. P. Heckbert. “Color Image Quantization for frame buffer displays”. *Computer Graphics*, **16**(3):297–307, 1982. 5

9. M. Gervauz, W. Purgathofer. “A simple method for color quantization: Octree quantization”. *Graphic Gems*, pp. 287–293, Academic Press, New York, 1990. 5

10. A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1998. 5

11. 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.)*, **1913**, pp. 265–270, 1993. 5

12. F.Heitz, P. Perez, and P. Bouthemy. “Multiscaleminimization of global energy functions in some visual recovery problems”. *CVGIP: Image Understanding*, **59**(1):125–134, 1994. 5

13. 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. 5

14. J.M. Buhmann, D.W. Fellner, M. Held, J. Ketterer, and J. Puzicha. “Dithered Color Quantization”. Proceedings *Proceedings of EUROGRAPHICS*, **17**(3), 1998. 5

15. C.I. de L’Eclairage. *Colorimetry*. CIE Pub. 15.2 2nd ed., 1986. 5

**Authors**

Denis V. Ivanov, *Denis@forsythe-it.com*
Dr. Yevgeniy P. Kuzmin, *Yevgeniy@forsythe-it.com*
Computational Methods Lab
Mathematics and Mechanics Dept
Moscow State University,
Vorobyovy Gory, Moscow, Russia, 119899