Michał Radziszewski[*], Witold Alda[**]

OPTIMIZATION OF FREQUENCY FILTERING IN RANDOM ACCESS JPEG LIBRARY

In the paper we present a method of direct access to single blocks of JPEG files which contain textures, with on-the-fly decompression. Anisotropic, adaptive filtering is applied in order to minimize visual defects appearing mainly on blocks borders. Main purpose of the method is to enable fast extraction of only these parts of an entire image which are currently needed and not to keep whole decompressed texture in the main memory. This approach enables effective usage of high quality textures with low memory consumption. It's benefits are mainly demonstrated in rendering complex 3D scenes using nondeterministic ray-tracing algorithm. The algorithms have been encapsulated into DLL and static library.

Keywords: *JPEG, adaptive filtering, textures, 3D scenes*

OPTYMALIZACJA FILTRACJI CZĘSTOTLIWOŚCIOWEJ W BIBLIOTECE JPEG O SWOBODNYM DOSTĘPIE

W artykule przedstawiono metodę swobodnego dostępu do pojedynczych bloków obrazów JPEG zawierających tekstury, z dekompresją wykonywaną na bieżąco. Zastosowane przy tym anizotropowe adaptacyjne filtry zostały dobrane pod kątem minimalizacji obserwowanych zniekształceń, pojawiających się głównie na granicach bloków. Głównym celem zaproponowanej metody jest umożliwienie szybkiego dostępu tylko do tych fragmentów obrazu, które aktualnie są wymagane, bez konieczności przechowywania całej zdekompresowanej tekstury w pamięci komputera. Takie podejście pozwala na efektywne użycie dużych tekstur o wysokiej rozdzielczości przy oszczędnym wykorzystaniu pamięci. Swoje zalety demonstruje głównie w renderowaniu scen 3D przy użyciu metody śledzenia promieni. Zaproponowane algorytmy zostały wbudowane w bibliotekę typu DLL i statyczną.

Słowa kluczowe: *JPEG, filtrowanie adaptacyjne, tekstury, sceny 3D*

1. Introduction

Texturing objects with high resolution image maps is common method which greatly increases rendering quality. Unfortunately these images in uncompressed formats usually consume huge amount of memory. If a scene contains many objects, which use

[*] PhD Student EAIiE, AGH-UST, mradzisz@student.agh.edu.pl

[**] Institute of Computer Science, AGH-UST, alda@agh.edu.pl

large quantity of different textures, loading all of them as arrays of pixels could easily cause an overflow of the computer memory capacity. This effect is much more painful than appears to be at first glance, because many objects often require more than one texture (i.e. working in *multi texturing mode*), when they want to display e.g. diffuse color texture, glossiness map, bump map or transparency map. With lack of memory, the only solution is to reduce textures resolution, which obviously leads to drastic decrease of rendered image quality. However, there exists a much better approach. It is possible to render directly from compressed textures and decompress only a tiny portion of it, which is exactly needed at a time. Easily achievable JPEG compression factor of, e.g. 16:1 is equivalent, in the sense of memory usage, to non-compressed image, downsampled four times. Yet the compressed image quality is much better than the downsampled one. The usage of compressed image data enables gaining benefits of full resolution textures for all scene objects, which may be very helpful while rendering, for example, magnifying curved mirrors.

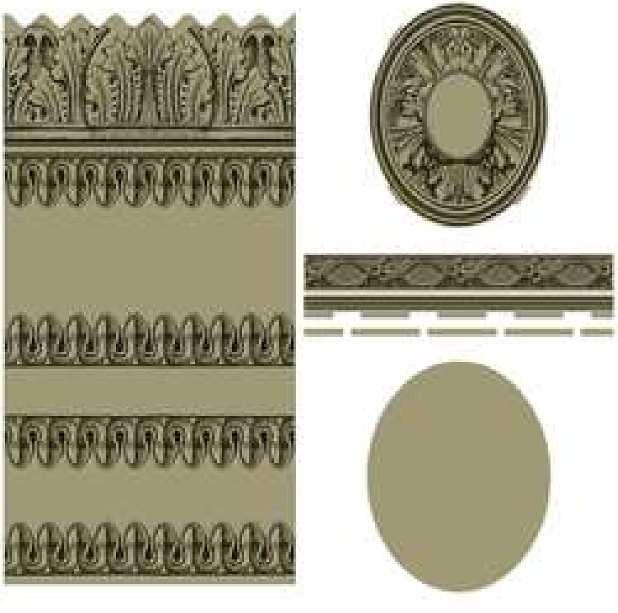

Fig. 1. Example of many textures placed in a single image

For the purpose of scene rendering usually many textures are put into a single image to avoid many files, as shown in Figure 1. Texture itself has to be rectangular, so in the case described, it contains some unused blank area, filled with constant color. The compression allows to store these blank parts with minimal memory usage. When thinking of the choice of compression method, we obviously have to use a lossy compression, because only this scheme allows achieving sufficiently high compression factor, up to 20:1 or sometimes even larger. Lossless compressed files (such as PNG) usually are much less compressed and also difficult for direct access to image fragments.

The JPEG standard is a good candidate for this purpose, since it is comparatively easy to convert it to direct access file format, and is very popular among variety of 3D models. The library functions allow to read JPEG files and keep them in memory in almost original state (from technical point of view, some parameters in conventional JPEG files stored as values relative to other parts of the image have to be converted to absolute values for a single fragment). To enable direct access to any part of data, JPEG blocks are initially indexed. The size of index array needs modest memory overhead, not decreasing overall JPEG compression factor much.

However, JPEG compression introduces artifacts, visible especially along borders of 8×8 pixel blocks as some kind of discontinuities. The filtering included in library offers solution to this flaw. It can be shown that these artifacts can be diminished without introducing excessive blur by careful usage of optimized filtering.

The adaptive filters used here, have their 'smoothing strength' adjusted to both image compression and sample location in a 8×8 block. It appears that heavily compressed images should be filtered stronger than good quality ones. Also filter should work with maximum smoothing only on borders of blocks. To achieve best possible performance, different image frequencies are treated separately, with strongly anisotropic filter. The compression error has been measured only for grayscale images, by comparing decoded image with a reference one using L_2 norm. Obtained results have been compared with standard decoder, in our case Windows Graphics Device Interface Plus (GDI+) used for this purpose, showing the advantage of our approach.

2. Related Work

There has been little work dedicated to rendering with compressed textures. In fact, we are not aware of any library dedicated to ray tracers. There are a few papers describing similar approach – for example [3] or [9], but they are designed for the needs of real time hardware rendering. That technique favors decompression speed over compression effectiveness, which is not a good choice for off-line ray tracing. The ray tracing is capable of rendering huge scenes, limited only by memory size, not necessarily in very short time, thus we believe for this purpose image quality is significantly more important than a speed. There have been much more work dedicated to image compression in general, not necessarily texturing. The most useful lossy compression scheme seems to be standard JPEG [1]. It uses a very efficient algorithm, and even though it is not new, no significantly better procedures have appeared since. Two examples of methods that could possibly replace JPEG are the wavelet transform based JPEG2000 [10], and fractal compression [8], which, according to the authors, can be used for texturing. However, they have not become as popular as the classic JPEG standard.

3. JPEG Compression

We describe here only main points of JPEG compression, relevant to our work. Detailed reference on JPEG file format can be found in [1]. To compress raw data into JPEG file, following steps have to be done:

1. For color input, all components of RGB model are separated and converted to YCrCb (luminance, red chrominance, blue chrominance) color space. This conversion allows to downsample and/or to quantize chrominance components with less visual loss of quality compared to simple operating on raw RGB data. In the case of grayscale input the image remains unchanged.
2. YCrCb components are split into 8 × 8 pixel blocks. Discrete Cosine Transform (DCT) is subsequently calculated on every block.
3. Frequency coefficients in each block are then quantized, resulting that a large part of them is rounded to zero. This is the main source of compression, but also of loss of information.
4. Blocks of components in interleaved order form so called Minimum Coded Units (MCU). Each MCU, which is differential and Huffman or arithmetic encoded, is placed sequentially in stream. For color images MCU in general is not equal to three blocks (one for each component), because sampling factors may be different for each component.

To decompress a JPEG file the above steps must be inverted and done in reverse order. Namely, they must follow: Huffman or arithmetic decoding, difference decoding, dequantization, inverse DCT, possibly upsampling and optional YCrCb to RGB color conversion. It is important to notice that JPEG standard has to be strictly preserved only until dequantization step, since operations performed here are based on bits and well defined integer values. Computing inverse DCT and upsampling leaves some freedom due to continuous values on which these procedures operate. As it appears later, slight modification of standard decoding, especially along block edges can improve performance.

4. Library Structure

The library is divided into two parts. In the first one, the input JPEG file is read with full error checking. The data is indexed to allow efficient direct access to single blocks of the image, and in this slightly converted, but still compressed form, is kept in memory. After that, the library is ready to perform texture sampling in several subsequent steps with optimal speed. The input single point address (u, v), normalized to unit square regardless of texture resolution, is converted by requested addressing mode (clamp, wrap or mirror, as available in DirectX or OpenGL). Next, the Minimum Coded Units necessary to filter point sample (up to four units, if sample happens to lie near the block's corner) are decoded. After that, the filtering is performed independently for each component in color images, and independently for frequency groups. Finally the conversion is done for color images, while in grayscale data single component is replicated onto all channels. In the current implementation due to optimization, the steps of the entire process are mixed, merged and executed in different sequence in order to improve the performance.

4.1. Image blocks indexation

The main difficulty in fast direct access to JPEG data fragments arises from differences in length of individual MCUs and differential encoding. Using information about sampling factor of each component and knowing that inverse DCT block is 8x8 pixels wide, we can easily transform the requested sample coordinates to MCU number, but we still cannot compute the address of sequence of bits defining the sample value. To cope with this we introduce an index array with pointers to each MCU, which solves the problem with different length of MCUs. To eliminate differential encoding we have to save differences between first inverse DCT coefficients (DC coefficients) at first bits of each block of new MCU. According to JPEG specification [1], difference must be coded on 11 bits. The index array uses 32 bit pointers, pointing individual bits, which gives a limitation on 512MB maximum compressed JPEG size which is far more than size of even largest textures. We also exploit the fact that Huffman decoding is typically faster than computing inverse DCT. What more, only even MCUs are indexed, which at costs of slight loss of speed reduces the memory overhead by half. Using the ideas mentioned above we managed to reduce the average index size to 2 bytes per MCU plus difference bits, which is acceptable memory overhead (discussed in 5.2).

4.2. Inverse DCT and Filtering

When using maps for ray tracing instead of standard displaying pixel by pixel, there is zero probability that a ray hits exactly the well defined integer sample location. That is, rays always fall between them, and requested color values must be somehow interpolated. The way of doing it has been described in many papers, e.g. in [5] or [6]. However, we present here an alternative approach for filtering, dedicated to JPEG inverse DCT compression scheme, which is more efficient than classic convolution with filter matrix. There exists a well known Fast Fourier Transform, capable of computing at once 64 pixel values from 64 coefficients (presented e.g. in [4]), excellent for usage in sequential approach. However, in direct access, only a single pixel value at a time is needed. Thus in the latter case naive sum of products seems to be the best option. What more, no filtering inside block is necessary, since discrete cosine coefficients can be, without any difficulties, converted to functions defined on real values from $[0, 8]$ range. Consequently, an extended iDCT algorithm can compute well defined value for any point from 8×8 square. Unfortunately, after such extension, values calculated in neighboring blocks mismatch, what causes visually distracting discontinuities seen as extra horizontal and vertical lines along the whole image. To cope with these artifacts, we have extended the iDCT domain further to whole $\mathbb{R}^2$ space, to enable filtering along borders of individual blocks. The extension of iDCT domain to all real values is done as follows. In the basic version of the algorithm, values outside the $[0, 8]$ range are clamped. However, introduction of sharp corners into this function decreases image quality. It can be shown, that smoothing them with 3^{rd} degree polynomial reduces the image error. The size of this smoothing depends on the extent of the filter. The dependence, however, is not very strong and due to this there can be found reasonably good smoothing coefficient for all image frequencies. This allows substantially optimize the computation of iDCT and therefore the overall performance of the library. The extension is presented in Figure 2. Dots show standard iDCT domain. The line which joins them marks iDCT domain as an extension from discrete points. The curved line is the final domain of the cosine transfer (in fact no more discrete) as all real values. Figure 3 shows the filtering effect. Solid lines present the mismatch of neighboring blocks when iDCT is extended to [0, 8] real interval. Extended cosine transform value is marked as a dashed line while the dotted line means the final smooth transition between blocks due to filtering. It shows that the filtering needs to be performed only on sides of blocks, while applied in the central part causes excessive blur, decreasing image quality.

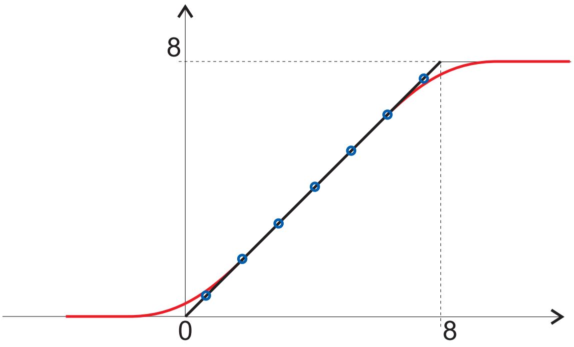

Fig. 2. Extension of 1D iDCT domain in 8 × 8 block

The filter extent has to be carefully adjusted. In case of low compressed files it should only mask small discontinuities, whereas in highly compressed ones – the filter, in addition, is designed to minimize quantization errors. What more, filter extent should depend on particular frequency component, which it has been applied to. However, filtering all 64 frequency coefficients separately leads to extremely slow algorithm, and splitting the domain into four 4×4 blocks seems to be enough for achieving good quality results in reasonable time. This way we use two frequency groups along each direction, which we call 'low' and 'high' frequencies. The actual value of filter extent for each frequency group is calculated as a value of 5^{th} degree polynomial approximating real data. Polynomial coefficients were obtained using a set of representative test images. For this purpose, several uncompressed photographs with different number of details were used (e.g. plain sky, forest, etc.). The photographs were compressed to JPEGs with quantization ranging from 1 to 255.

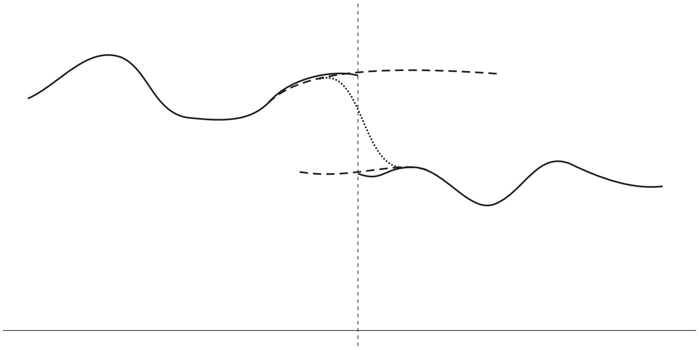

Fig. 3. Smoothed 1D iDCT value. Solid line comes from transform on [0,8] interval, dashed line – from transform extended to entire space; dotted line shows the result of filtering

For each test photograph and for each quantization the filter extent was manually set in order to minimize the error. The error estimation was guided by comparison of decoded image with reference bitmap, using L_2 norm as well as by the visually estimated image quality. We have set the parameters to minimize objective error and then increased the filter extents slightly to improve image smoothness without introducing too much blur. In Figure 4 and Figure 5 there are presented results of this procedure. Figure 4 presents the evaluation of filter extent along horizontal axis. Filter extent along vertical axis uses the same data, but the 2×2 matrix of frequency blocks is transposed (which results in swapping curve marked by '×' with '+'). Due to library architecture (output can be filtered from at most two blocks in horizontal direction and two blocks in vertical direction) the maximum plausible extent is 4 (half of the block size which is 8). The minimum extent is not specified in advance, but the value about 0.6 is just enough to ensure visual smoothness between blocks without introducing excessive blur.

5. Results

5.1. Error Reduction

The tests have been performed on variety of JPEG files, different from those previously used to adjust filter parameters. As reference images we have used monochromatic 640×480 pixel maps (the results for colour images are very similar and therefore we do not present them here). First, we introduced some controllable error by compressing these images using Corel Photo Paint with different compression coefficients. Next we decoded the compressed images using both our library and Windows GDI+ as a reference decoder. It seems that it strictly keeps the standard – the error metrics is almost identical with disabled filtering in our library. Finally we compared resulting bitmaps with initial reference images using L_2 norm for pixels on the edges (28 pixels out of 64 for each block). The remaining pixels are barely affected by this filtering

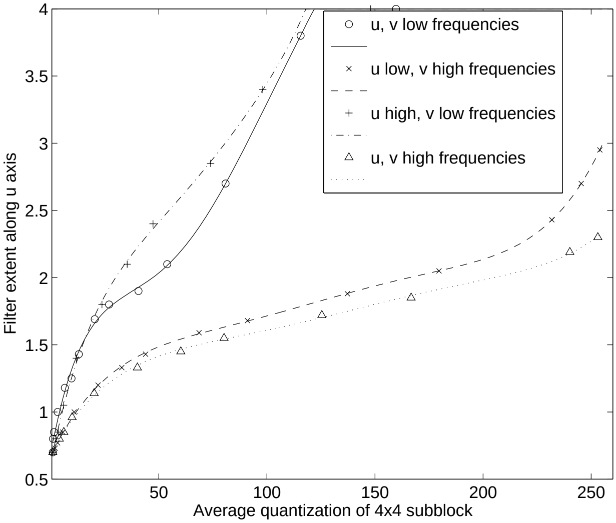

Fig. 4. Fitting filter extent to measured data

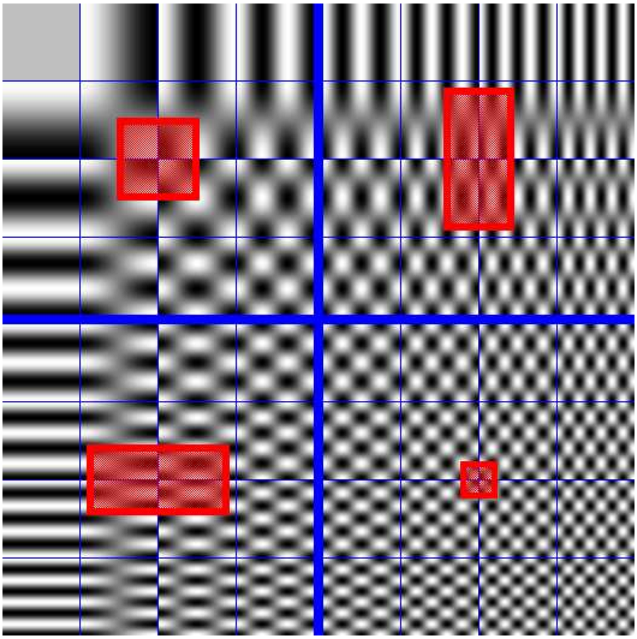

Fig. 5. Filter extents for different frequencies. The anisotropy of filters follows the average anisotropy of frequencies

algorithm. We have observed the tendency that our library is most effective with images with not too many details. The efficiency is slightly worse on files with lots of high frequency details. The measured decrease of error is similar for all test files, but the actual error value is much larger for detailed images, which results in less relative quality gain. Figures 6 and 7 present the most extreme cases of these tests.

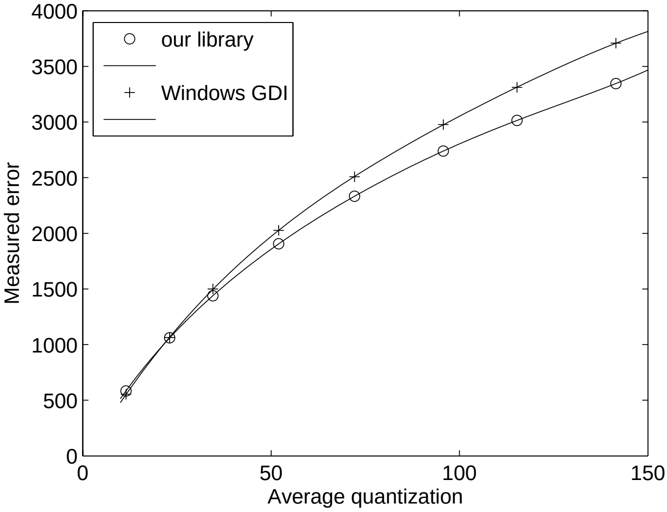

Fig. 6. Error comparison for images with high frequency content

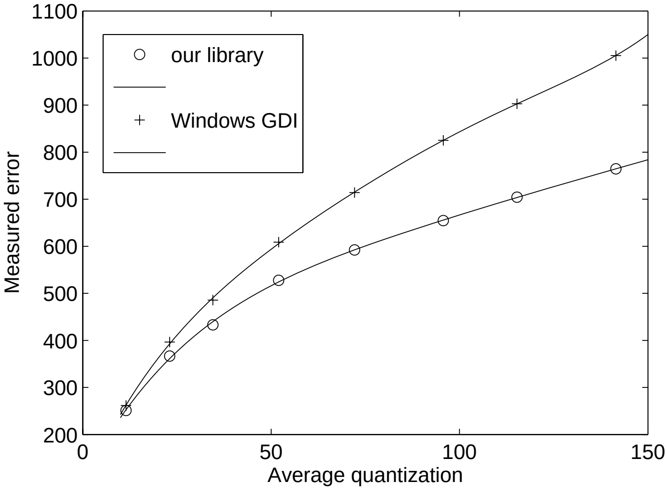

Fig. 7. Error comparison for images with low frequency content

5.2. Compression Efficiency

We have compared our method with uncompressed bitmaps. We present the results obtained by measuring performance for three different classes of JPEGs. All of them are decompressed by dedicated functions with different speed. These classes are:

- low compressed RGB data, MCU is 8×8;
- high compressed RGB data, MCU is 16×16;
- grayscale data, MCU is 8×8;

All tests except one are done by randomly accessing to 4 million pixel fragments from 3000×4000 map. Only the first test high compressed JPEG has been done on 2000×2000 map. Processing speed has been measured on 2.4 GHz PIV processor running under control of Windows XP. Library has been compiled with Intel C++ 8.0 compiler. The tests use typical textures prepared for rendering 3D scenes and models, which differ from 'ordinary' (i.e. with single photograph) JPEGs. These maps have some unused blank area filled with single color. It results in slightly better compression ratio compared to common JPEGs, however, due to our iDCT implementation has little impact on speed.

Table 1

Image type	Total time/ constructor time	Memory	Compression ratio converted file (original file)	Overhead
1	2	3	4	5
jpg lo	19s/1.1s	6.4 MB	1:5.4 (1:6)	8%
jpg hi	26s/0.09s	559 kB	1:21 (1:22)	8%
	24s/0.19s	696 kB	1:49 (1:60)	24%
	23s/0.17s	360 kB	1:95 (1:145)	52%
bmp	5.9s/-	34.3 MB	n/a	n/a
jpg gr	7.3s/0.58s	841 kB	1:13 (1:26)	98%
bmp gr	5.6s/-	11.4 MB	n/a	n/a

Results on compression efficiency are summarized in Table 1. Total processing time per pixel is measured with highest filtering quality enabled. Time taken by constructor is less then 6% in the worst case. Bitmaps constructor time was less than measuring error. Memory is the total memory required to store all data of decompressed file and overhead is computed as (mem – filesize) / filesize. Ratio is ratio of compression for converted to direct access JPEG data in memory, in brackets for the original file. It is worth to mention that indexing every even MCU increases the time only by about 10%, while reducing overhead by half.

5.3. Limitations

The library cannot use progressive version of JPEG format. We do not consider it as a major drawback because such files can be converted to sequential format by external tools without any loss of quality. We also limit the maximum number of components to 3 (JPEG standard supports 255). For further improvement of performance we have limited maximum sampling factors to 2 (JPEG specification requires 4). This allows making multiplication and division operations by shifting bits. We also do not allow multiscan files. It is unlikely to have one that satisfies all previous restrictions and has more than one scan.

6. Conclusions

We have plugged our JPEG library in raytracer (our implementation of bidirectional path tracer). The test was performed on scene that contains architectural model of 30k triangles, one spherical light source and camera inside. Every triangle has a diffuse material with color JPEG with 16x16 MCUs. About half of the triangles have an additional glossiness controlled by grayscale JPEG. Highest quality JPEG filtering was enabled. The total rendering time was divided in three parts:

1. kd tree traversal + ray-triangle intersection – about 50% of total time,
2. JPEG decompression – about 40%,
3. other operations, e.g. statistical calculations, memory management, frame buffer, etc. – 10%.

After these tests, we found that modern processors have enough computational power for accessing JPEG compressed data while rendering. The benefits from using our library are even greater during parallel rendering, when there is much more computational power, but no more memory (data have to be replicated on every machine).

References

[1] Information Technology – Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidelines. Recommendation T. 81. International Telecommunication Union, September 1992

[2] Lane T. G.: *Independent JPEG Group software.* From: `www.ijg.org`

[3] Beers A. C., Agrawala M., Chaddha N.: *Rendering from Compressed Textures.* [in:] Computer Graphics (SIGGRAPH '96 Proceedings), vol. 30, pp. 373–378, August 1996

[4] Loeffler C., Ligtenberg A., Moschytz G. S.: *Practical Fast 1-D DCT Algorithms with 11 Multiplications.* [in:] Proc. Int. Conf. on Acoustical and Speech, vol. 2, pp. 988–991, May 1989

[5] Smith A. R.: *A pixel is not a little square.* Microsoft technical report, 1995

[6] Mitchell D., Netravali A.: *Reconstruction Filters in Computer Graphics.* [in:] Computer Graphics (SIGGRAPH '88 Proceedings), vol. 24, pp. 221–228, August 1988

[7] Heckbert P.: *Fundamentals of Texture Mapping and Image Warping.* Master's Thesis, University of California, Berkeley 1989

[8] Stachera J., Nikiel S.: *Fractal Image Compression for Effective Texture Mapping.* WSCG '04 Proceedings

[9] van Varen J.: *Real-Time DXT Compression.* Id Software, inc. 2006

[10] Adams M. D.: *The JPEG-2000 Still Image Compression Standard.* 2002