# Real-Time Texture Streaming & Decompression

**November 11th 2006**
**J.M.P. van Waveren**



## Abstract

In this article several different lossy compression formats and streaming solutions are evaluated for rendering textures from very large texture databases. Furthermore a compression format similar to JPEG and an SIMD optimized threaded pipeline is introduced to achieve high speed streaming of textures.

# 1. Introduction

Textures are digitized images drawn onto geometric shapes to add visual detail. In today's computer graphics a tremendous amount of detail is mapped onto geometric shapes during rasterization. Especially uniquely textured environments require huge amounts of texture data. Not only textures with colors are used but also textures specifying surface properties like specular reflection or fine surface details in the form of normal or bump maps. All these textures can consume large amounts of storage space and bandwidth. Fortunately compression can be used to reduce the storage and bandwidth requirements.

There are compressed texture formats like DXT or S3TC that can be decompressed in hardware during rasterization on current graphics cards. However, these formats are optimized for decompression in hardware and as such typically do not result in the best possible compression ratios. Graphics applications may use vast amounts of texture data that is not displayed all at once but streamed from disk as the view point moves or the rendered scene changes. Strong compression may be required to deal with such vast amounts of texture data to keep storage and bandwidth requirements within acceptable limits. As these textures are streamed from disk they have to be decompressed on the fly before they can be used for rendering on current graphics cards.

There are several formats like GIF, PNG and JPEG-LS for lossless compression of images. Lossless (reversible) image compression techniques preserve the information so that exact reconstruction of the image is possible from the compressed data. In other words there is no loss in quality when an image is compressed to one of these formats. However, these compression formats typically also do not result in compression ratios that are high enough to store vast amounts of texture data.

In this article several different lossy compression formats and streaming solutions are evaluated for rendering textures from very large texture databases. Furthermore a compression format similar to JPEG and an SIMD optimized threaded pipeline is introduced to achieve high speed streaming of textures.

# 2. Lossy Color Image Compression Formats

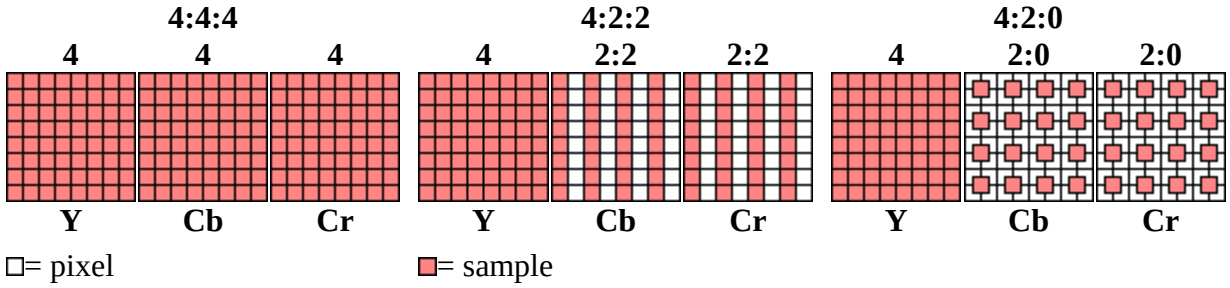
There are several standardized compression formats available for lossy compression of color images. Some well known standards are JPEG, JPEG 2000 and HD Photo.

## 2.1 JPEG

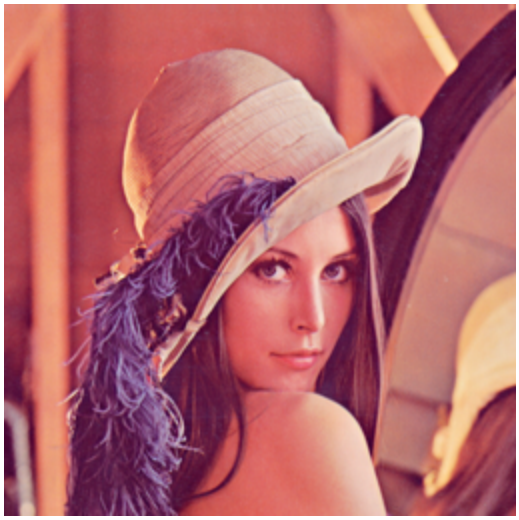
JPEG [1] is a lossy compression format which allows for a wide range of compression ratios at the cost of quality. Compression ratios well beyond 20:1 are possible but there may be a noticeable loss in quality. In particular JPEG compression may produce significant blocking artifacts at higher compression ratios. However, at a 10:1 compression ratio an image can usually not be distinguished by eye from the original.

The name JPEG stands for Joint Photographic Experts Group, the name of the joint ISO/CCITT committee which created the standard. JPEG was designed specifically to discard information that the human eye cannot easily see. Slight changes in color are generally not noticeable, while the human eye is much more sensitive to slight changes in intensity (light and dark). Furthermore high frequency changes are usually less noticeable to the human eye than low frequency changes.

A JPEG compressor first transforms the color data from the RGB color space to an appropriate color space to separate the intensity (luma) from the color information (chroma). JPEG uses the YCbCr color space which is the same as the color space used by PAL, MAC and Digital color television transmission. The Y component represents the luma of a pixel and the components Cb and Cr represent the blue and red chroma respectively. Chroma subsampling is often used to improve the compression ratio with little loss in quality because the human eye is less sensitive to high frequency chroma changes. Typical sampling ratios are 4:4:4 (no downsampling), 4:2:2 (reduce by factor of 2 in horizontal direction), and most commonly 4:2:0 (reduce by factor of 2 in horizontal and vertical directions).




The images below show the 256x256 "Lena" image converted to YCbCr with a 4:2:0 sub-sampling ratio.



Original 256 x 256 "Lena" image.

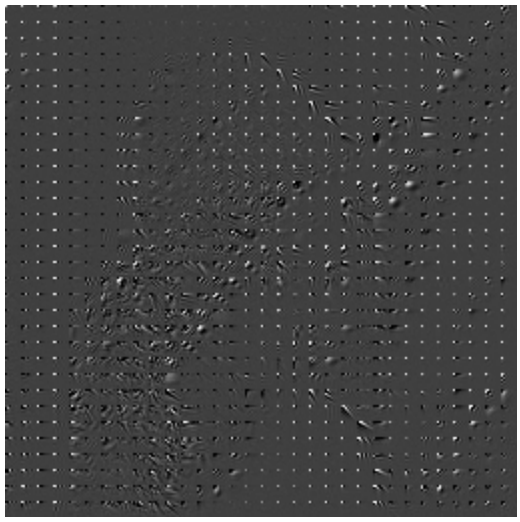
4:2:0 luma (Y) drawn to an image.

4:2:0 blue chroma (Cb) drawn to an image.

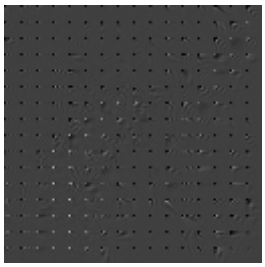
4:2:0 red chroma (Cr) drawn to an image.

Each of the three channels, luma (Y), red chroma (Cr) and blue chroma (Cb), is processed individually. A Discrete Cosine Transform (DCT) is used on each 8x8 block of data from one of the channels to transform the spatial image data into a frequency map. The frequencies represent the average value and successively higher-frequency changes within a block. The images below show the frequency data for the "Lena" image after conversion to YCbCr with a 4:2:0 sub-sampling ratio.

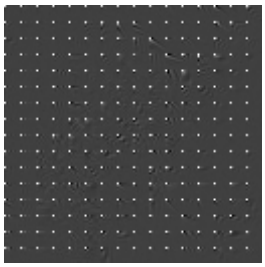
Original 256 x 256 "Lena" image.



4:2:0 luma (Y) frequencies drawn to an image.



4:2:0 blue chroma (Cb) frequencies drawn to an image.



4:2:0 red chroma (Cr) frequencies drawn to an image.

The frequency data is then quantized to remove image information that is less noticeable to the human eye. Typically high frequency data is diminished while low frequency data is maintained. A lot of the high frequency color data can usually be removed before it becomes noticeable to the human eye.

The quantized frequencies are then rearranged to zig-zag order to maximize the length of runs of zeros. The least visible coefficients, the ones most likely to be zero-ed, are grouped at the end of the sequence. The rearranged and quantized frequencies are compressed with an entropy encoder. Typically a simple Huffman encoder combined with run-length compression is used for this purpose. The JPEG standard also allows the use of Arithmetic coding which is mathematically superior to Huffman coding. However, Arithmetic coding is rarely used as it is much slower to encode and decode compared to Huffman coding.

## 2.2 JPEG 2000

JPEG 2000 [3, 4] is a Discrete Wavelet Transform (DWT) based image compression standard created by the Joint Photographic Experts Group committee with the intention of superseding the Discrete Cosine Transform (DCT) based JPEG standard. JPEG 2000 has superior compression performance in comparison to the JPEG standard. The compression gains over JPEG are attributed to the use of the DWT and a more sophisticated entropy encoding scheme.

A JPEG 2000 compressor first splits the image into tiles, rectangular regions of the image that are transformed and encoded separately. The purpose of these tiles is to cope with memory limitations. The compressor then transforms the color data in the tiles from the RGB color space to the YCbCr color space or uses a Reversible Component Transform (RCT) leading to three components. These components are then wavelet transformed individually to an arbitrary depth.

The result of the wavelet transform is a collection of subbands that represent several approximation scales. A subband is a set of real coefficients that represent aspects of the image associated with a certain frequency range as well as a spatial area of the image. These coefficients are subjected to uniform scalar quantization, giving a set of integer numbers. These quantized subbands are split further into precincts which are regular non-overlapping rectangular regions in the wavelet domain. Precincts are split further into code-blocks. Except those located at the edges of the image, code-blocks are located in a single subband and have equal sizes.

The encoder has to encode the bits of all quantized coefficients of a code-block, starting with the most significant bits and progressing to less significant bits by a process called the Embedded Block Coding with Optimal Truncation (EBCOT). In this encoding process, each bit-plane of the code-block gets encoded in three so called coding passes, first encoding bits of insignificant coefficients with significant neighbors, then refinement bits of significant coefficients and finally bits of coefficients without significant neighbors. The three passes are called significance propagation, magnitude refinement

and cleanup pass, respectively. The bits selected by these coding passes then get encoded by a context-driven binary arithmetic coder. The context of a coefficient is formed by the state of its nine neighbors in the codeblock. The result is a bit-stream that is split into packets where a packet groups selected passes of all code-blocks from a precinct into one indivisible unit. Packets are the key to scalability where packets containing less significant bits can be discarded to achieve lower bit-rates at the cost of quality.

Although JPEG 2000 produces superior quality compared to JPEG, the gains are modest at medium compression ratios (10:1). The improvement is typically much larger at higher compression ratios [5] but there may also be a noticeable loss in quality. JPEG 2000 eliminates some of the compression artifacts introduced by JPEG at higher compression ratios, such as blocking artifacts. However, JPEG 2000 can introduce quite prominent blurring and ringing artifacts. Furthermore JPEG 2000 decompression is significantly more computationally expensive than JPEG decompression and requires more memory during decoding.

## 2.3 HD Photo

HD Photo [12] (formerly known as Windows Media Photo) employs a compression algorithm optimized for the digital photography market. HD Photo offers image quality comparable to JPEG-2000 [13] with computational complexity and memory requirements closer to JPEG.

Images are processed in 16x16 macro blocks, allowing a minimal memory footprint for embedded implementations. HD Photo uses the reversible YCoCg-R color space and a reversible lapped biorthogonal transform (LBT) based on the Hadamard transform and rotation. The transform coefficients are quantized and coefficient prediction and adaptive scanning is used before entropy encoding significant bits in several passes.

The compression algorithm is computationally efficient, and is designed for high performance encoding and decoding while minimizing system resource requirements. However, even though HD Photo is not as computationally expensive as JPEG-2000, it is still a factor slower than JPEG.

# 3. Hardware Accelerated Decompression

Current graphics cards allow several forms of hardware decompression of color images by exploiting a GPU, video decoding units or texture units.

## 3.1 Inverse DCT on a GPU

Part of the JPEG decompression algorithm can be implemented in one or more fragment programs which can be executed on current GPUs. For instance the inverse Discrete Cosine Transform (DCT) can be implemented in fragment programs as shown by nVidia [15].

On a GeForce 6800 doing no other work, the performance of the inverse DCT implemented in fragment programs by nVidia is about 134 Mega Pixels per second (MP/s). However, the fragment programs only performs the inverse DCT for a grayscale image. 1.5 times the amount of work is required to decompress a 4:2:0 JPEG color image, which brings the process down to 89 MP/s. Combining the results and converting the color space back to RGB further reduces the performance.

Entropy decoding can typically not be implemented in a fragment program because it is not possible to read from a variable length bit stream on current graphics cards. As such the entropy decoding has to be done on a CPU and the quantized frequencies have to be uploaded to the graphics card. Unfortunately on today's CPUs the most expensive part of JPEG decompression is the entropy decoding (mostly due to branching). Even if a fast Huffman decoder is used with a lookup table this is typically more expensive than the inverse DCT or the color conversion. Furthermore pushing the quantized frequencies to the graphics card increases the upload. There is typically more upload bandwidth required than used to upload the uncompressed image because the frequencies cannot be uploaded as 8-bit texture components.

The nVidia inverse DCT implementation requires multiple fragment programs and multiple rendering passes. This means multiple draw calls which typically does not improve the overall performance. Obviously there is a trade between using the CPU and GPU. With a lot of Hyper Threaded CPUs out there and a growing base of multi-core CPUs, decompressing images on a CPU is usually faster than running a fragment program on the GPU because the GPU is typically already taxed with 3D rendering.

## 3.2 Hardware Accelerated MPEG Decoding

Most of today's graphics cards support hardware accelerated MPEG-1 or MPEG-2 decoding. MPEG-1 and MPEG-2 can be setup to store only I-frames which are close to compressed JPEG images. The MPEG decoder on the graphics card can write an MPEG-1 or MPEG-2 image directly to a texture in memory on the graphics card which can then be used for rendering. As such the hardware decoder can be used for general hardware accelerated texture decompression.

Images can be uploaded to the graphics card as MPEG-1 or MPEG-2 files with a single or multiple I-frames and they can be decompressed in hardware. During decoding these MPEG-1 or MPEG-2 files will also take up memory on the graphics card. Unfortunately current drivers are not mature enough to really benefit from hardware accelerated MPEG decoding for real-time texture decompression.

## 3.3. DXT Compression

The DXT format, also known as S3TC [16, 17], is designed for real-time decompression in hardware on the graphics card during rendering. DXT is a lossy compression format with a fixed compression ratio of 4:1 or 6:1. DXT compression is a form of Block Truncation Coding (BTC) where an image is divided into non-overlapping blocks and the pixels in each block are quantized to a limited number of values. The color values of pixels in a 4x4 pixel block are approximated with equidistant points on a line through color space. Such a line is defined by two end points and for each pixel in the 4x4 block an index is stored to one of the equidistant points on the line. The end points of the line through color space are quantized to 16-bit 5:6:5 RGB format and either one or two intermediate points are generated through interpolation.

Most of today's graphics cards support the DXT format in hardware. Unfortunately the compression ratio of DXT is only 6:1 for three channel color images and 4:1 for channel color images with alpha channel. These compression ratios are generally not good enough to store vast amounts of texture data. However, a DXT compressed images can be compressed down further by exploiting specific knowledge about the structure of the DXT format. Half the data in the DXT1 format is used for RGB colors in 16-bit 5:6:5 format. For each 4x4 block of pixels there are two such RGB colors that define the end points of the line through color space which is used to approximate the colors in the block. The other half of the data in the DXT1 format is used to store indices to equidistant points on the lines through color space.

The colors from all the 4x4 blocks can be placed in one or two textures and such textures can be compressed with a regular texture compressor like JPEG or HD Photo. The indices, however, are much harder to compress. The indices need to be compressed with a lossless compressor because noticeable artifacts may occur even if the indices are only off by one. A good DXT compressor will typically try to use all points on a line through color space to preserve as much detail as possible. As a result compressing the indices with an entropy encoder does not work well because the different indices occur at similar

frequencies. Furthermore the lines through color space can have any orientation in the original texture. As a result run-length or LZ-based compression does not work very well either because of the randomized nature of the sequences of indices. Only if the original texture has a lot of flat areas, or areas with smooth axis aligned color ramps the individual indices or sequences of indices will be similar and they will compress well.

256 x 256 "Lena" image compressed to DXT1 with the ATI Compressonator.

The indices from the DXT1 format drawn to an image. The indices have been remapped to natural order on the lines through color space. There are some patterns but there is also a lot of noise.

First color image.

Second color image.

The compression of the indices can be improved by rotating and/or mirroring the indices in each 4x4 block in order to line them up. However, even then the compression ratios are still not very impressive (typically below 4:3). In other words it is hard to compress the indices for high detail images which also means it is hard to compress a DXT1 compressed image down to half its original size. Even if the color data would compress down to nothing, the indices remain and they consume half the data in the DXT1 format.

To compress a RGBA image in DXT5 format, the two textures from the DXT1 format can be extended with an alpha channel. In the DXT5 format half the data is used for colors plus an alpha channel and the other half is used to store indices either to points on lines through color space or to points on lines through alpha space. Just like the color indices, compressing the indices to points on lines through alpha space does not work well for images with a high detail alpha channel.

Tests using high detail images show that the two color (+alpha) textures can be compressed down with a JPEG compressor to about 50 to 40 percent of their DXT compressed size with little loss in quality. The indices can be compressed down to about 90 to 80 percent of their original size. In the best case this results in a compression ratio of about 10:1 for three channel color images and 7:1 for four channel color images with an alpha channel. Although these compression ratios are reasonable, it is hard to achieve these ratios for general images and it is even harder to go up to higher compression ratios.

At the cost of quality the DXT color data can be compressed down further but any loss in quality and compression artifacts are typically magnified by the DXT compression.

Another issue is that mip maps which are typically used to avoid aliasing artifacts during 3D rendering either have to be stored and compressed with a full resolution image which requires more storage space, or a DXT compressed image has to be decompressed, then mip maps have to be generated from the uncompressed DXT and they have to be re-compressed to DXT format. Creating mip maps from DXT compressed data typically produces noticeable artifacts.

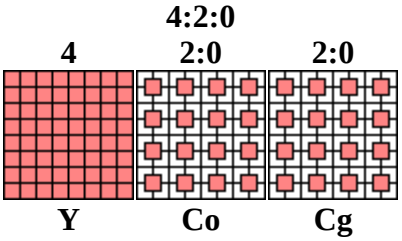
# 4. DCT Based Compression Format

A compression format for real-time texture streaming must achieve good compression ratios and has to allow the implementation of a fast decompression algorithm on today's computers. Evaluation of the above compression formats and decompression solutions leads to a compression format very similar to JPEG. Regular JPEG is a free standard and very fast implementations of the different sub-routines used for JPEG decompression are readily available.

The compression format presented here is different from JPEG in that it uses the YCoCg color space instead of the YCbCr color space. The YCoCg color space was first introduced for H.264 video compression [18, 19]. The RGB to YCoCg transform has been shown to be capable of achieving a decorrelation that is much better than that obtained by various RGB to YCbCr transforms and is very close to that of the KL transform when measured for a representative set of high-quality RGB test images [19]. Furthermore the transformation from RGB to YCoCg is very simple and requires only integer additions and shifts. The following matrix transformation shows the conversion from RGB to YCoCg.

$$\begin{bmatrix} \mathbf{Y} \\ \mathbf{Co} \\ \mathbf{Cg} \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{G} \\ \mathbf{B} \end{bmatrix}$$

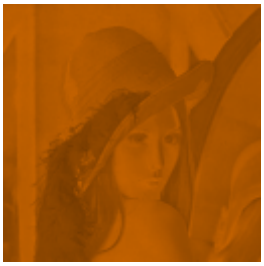
The compression format presented here uses a sub-sampling ratio of 4:2:0 to achieve high compression ratios at a minimal loss in quality.



The images below show the "Lena" image converted to YCoCg with a 4:2:0 sub-sampling ratio.

Original 256 x 256 "Lena" image.

4:2:0 luma (Y) drawn to an image.

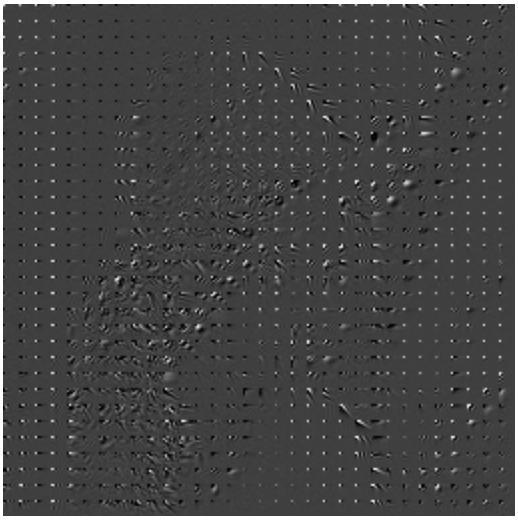


4:2:0 orange chroma (Co) drawn to an image.

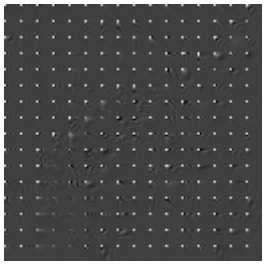
4:2:0 green chroma (Cg) drawn to an image.

Just like JPEG each of the three channels, luma (Y), orange chroma (Co) and green chroma (Cg), is processed individually. Also just like JPEG a Discrete Cosine Transform (DCT) is used on each 8x8 block of data from one of the channels to transform the spatial image data into a frequency map. The images below show the frequency data for the "Lena" image after conversion to YCoCg with a 4:2:0 sub-sampling ratio.

Original 256 x 256 "Lena" image.



4:2:0 luma (Y) frequencies drawn to an image.



4:2:0 orange chroma (Co) frequencies drawn to an image.

4:2:0 green chroma (Cg) frequencies drawn to an image.

The frequency data is then quantized to remove image information that is less noticeable to the human eye. The quantized frequencies are rearranged to zig-zag order and then compressed with a simple run-length and Huffman encoder.

# 5. Fast Decompression

A fast decompressor is required for real-time streaming of compressed textures. Decompressors for the JPEG format are readily available [20, 21]. The decompressor described here for the compression format described in section 4 is very similar to a JPEG decompressor.

Because the color data is stored in 4:2:0 format the decompressor works on tiles of 16x16 pixels. Such a tile contains one 8x8 block for each of the two chroma components and four 8x8 blocks for the luma. There is only one intermediate buffer to which the DCT coefficients for one tile are run-length and Huffman decoded. The coefficients are then inverse transformed in place in this same buffer. The color space conversion transforms YCoCg data from this intermediate buffer to RGB data directly into the destination image to minimize the memory footprint during decompression.

An implementation in C of the decompression of one tile can be found in appendix A. This particular implementation decompresses tiles with three color channels and an alpha channel. The decompression of the alpha channel can be trivially removed for the decompression of images with only three color channels.

## 5.1 Run-Length & Huffman Decoding

An implementation in C of the run-length and Huffman decoding can be found in appendix B. The decoder reads run-length and Huffman compressed data from a bit stream. SIMD optimized routines are readily available to fetch bits from a bit stream [22]. However, the run-length and Huffman decoder presented here does not use SIMD code. The decoder presented here is optimized to reduce the number of conditional branches and the remaining branches are setup to be more predictable. Conditional branches that are hard to predict typically result in numerous mispredictions and significant penalties on today's CPUs that implement a deep pipeline [23,24]. When a branch is mispredicted, the misprediction penalty is typically equal to the depth of the pipeline.

Bits are read from the bit stream in the routines 'GetBits' and 'PeekBits'. Both routines read bits from an intermediate buffer which may need to be re-filled regularly. This intermediate buffer is filled in 'FillBitBuffer'. Filling the bit buffer in 'FillBitBuffer' has been made branchless where the last byte of the bit stream is repeated if the decoder tries to read beyond the end of the input bit stream.

Some key observations can be made in the Huffman decoder that allow the removal of many conditional branches. First of all the Huffman codes used here are never longer than 16 bits. Furthermore by definition Huffman codes for the more frequently occurring symbols use fewer bits. In particular most symbols are encoded with 8 or less bits. As such two lookup tables are implemented for codes with 8 or less bits. The first lookup table 'look_nbits' stores the number of bits for Huffman codes with 8 or less bits. The second lookup table 'look_sym' stores the actual symbol for a given bit pattern of 8 bits that represents a Huffman code of 8 or less bits. Both lookup tables are indexed with 8

bits from the input stream. These 8 bits represent either the first 8 bits of a longer Huffman code, or a Huffman code of 8 or less bits and possibly additional bits that are not part of the Huffman code. For all bit patterns of 8 bits that represent the first 8 bits of longer Huffman codes the 'look_nbits' table stores a zero. By testing for a zero in the 'look_nbits' table the decoder can tell whether or not the first 8 bits read from the input stream contain a full Huffman code or a partial Huffman code. For Huffman codes of 8 or less bits the bits are removed from the intermediate bit buffer and the actual symbol is looked up in the 'look_sym' table.

Huffman codes with more than 8 bits are decoded in 'DecodeLong'. This routine first reads 16 bits for the longest possible Huffman code. Then a fast test is used to determine the actual number of bits for the long code. The 16 bits represent a long code of 9 up to 16 bits which is left justified to 16 bits. This left justified Huffman code is compared to a set of constants specific to the given Huffman table, where the actual code has 'n' or more bits if the left justified code is larger than or equal to 'test_nbits[n]'. The actual number of bits is determined by comparing the left justified code with all constants and accumulating the results of the comparisons. Once the actual number of bits is known the bits are removed from the intermediate bit buffer. Furthermore a lookup table is used to retrieve the symbol for the Huffman code.

After decoding the category the 'HuffmanDecode' routine reads an offset and calls the routine 'ValueFromCategory' to calculate the actual coefficient value from the category and offset. This routine has been made branchless without using a lookup table.

The end result is a run-length and Huffman decoder with small lookup tables and very few conditional branches. There are two conditional branches to fill the bit buffer, one in 'GetBits' and one in 'PeekBits'. There is one conditional branch to switch between decoding short and long Huffman codes in 'GetCategory'. There is a loop in 'DecodeLong' but this loop always executes a fixed number of iterations and can be trivially unrolled by the compiler. Furthermore there is one conditional branch for the run-length decoding in 'HuffmanDecode'.

## 5.2 Inverse DCT

There are several fast SIMD optimized implementations of the Inverse Discrete Cosine Transform (iDCT) available [26, 27, 28]. The iDCT algorithm used here is based on the Intel AP922 algorithm [27, 28]. This algorithm is specifically designed to exploit integer SIMD architectures while satisfying the precision requirements of the IEEE standard 1180-1900 [25].

The AP922 algorithm uses several different rounding and correction techniques to counter loss in precision. The AP922 algorithm first operates on rows using 32-bit precision, and then on columns using 16-bit precision. The row iDCT uses the MMX / SSE2 instruction **'pmaddwd'** which calculates the 2D dot product of 16-bit integers and stores the result as a 32-bit integer. The results of the dot products are added and right-shifted to less precision with SHIFT_ROUND_ROW(). A rounder RND_INV_ROW (0.5

fixed point) is added before shifting down with SHIFT_ROUND_ROW() for proper rounding.

The column iDCT uses the MMX / SSE2 instruction '**pmulhw**' which computes

**( a * b ) >> 16**.

This instruction rounds down and as such introduces a bias of -0.5. The AP922 algorithm adds corrections in order to counter this bias. At two points a value of one is added and at another two points a value of one is subtracted. Furthermore, in two places an 'or' instruction is used to set the least significant bit which is statistically equivalent to adding 0.5. When using a column with only zeros and assuming infinite precision, these corrections in the AP922 algorithm accumulate to the following bias values before using SHIFT_ROUND_COL().

| row | bias |
|---|---|
| 0 | + **1** |
| 1 | + **0.5** + **sqrt**( **0.5** ) |
| 2 | - **0.5** - **sqrt**( **0.5** ) |
| 3 | - **1** |
| 4 | - **1** |
| 5 | - **1.5** + **sqrt**( **0.5** ) |
| 6 | - **0.5** - **sqrt**( **0.5** ) |
| 7 | - **1** |

Instead of adding and or-ing values in the column iDCT, these bias values can be propagated back through the column iDCT after removing the corrections, and added to the same rounders that are used for proper rounding of the row iDCT. Through back propagation the following rounders can be derived for BITS_INV_ACC = 4. These rounders can be divided by two for the case BITS_INV_ACC = 5.

| row | rounder |
|---|---|
| 0 | - **2048** |
| 1 | + **3755** |
| 2 | + **2472** |
| 3 | + **1361** |
| 4 | + **0** |
| 5 | - **1139** |
| 6 | - **1024** |
| 7 | - **1301** |

Before doing the shift-right with SHIFT_ROUND_COL() the AP922 algorithm adds RND_INV_COL (0.5 fixed point) for proper rounding. This addition can be avoided by

adding 65536 to the above rounder for the first row. Furthermore the above rounders are added in the row iDCT before doing the SHIFT_ROUND_ROW() where the AP922 algorithm adds RND_INV_ROW for proper rounding of the result of the row iDCT. In other words not only the above rounders are added before the SHIFT_ROUND_ROW(), but also the original RND_INV_ROW.

| row | rounder |
|---|---|
| 0 | RND_INV_ROW - **2048** + **65536** |
| 1 | RND_INV_ROW + **3755** |
| 2 | RND_INV_ROW + **2472** |
| 3 | RND_INV_ROW + **1361** |
| 4 | RND_INV_ROW + **0** |
| 5 | RND_INV_ROW - **1139** |
| 6 | RND_INV_ROW - **1024** |
| 7 | RND_INV_ROW - **1301** |

The end result is the above set of rounders that add 0.5 (fixed point) before right-shifting for proper rounding at the end of the row iDCT, add a bias to each row to take care of the rounding in the column iDCT, and also add 0.5 (fixed point) before right-shifting for proper rounding at the end of the column iDCT. When these rounders are pushed forward through SHIFT_ROUND_ROW() and the column iDCT without corrections, this results in rounding that is the same as or superior to the rounding of the AP922 algorithm.

Adding all the rounders in one place does not only improve the precision but also simplifies the algorithm by removing several instructions throughout the column iDCT. In particular this saves 12 **'paddsw'** instructions and 4 **'por'** instructions in the MMX implementation and also saves 6 **'paddsw'** instructions and 2 **'por'** instructions in the SSE2 implementation. The AP922 algorithm has been modified further to perform dequantization right before the iDCT without having to temporarily spill the dequantized values to memory.

An implementation in C of the modified AP922 algorithm can be found in appendix C. MMX and SSE2 implementations can be found in appendix D and E respectively.

## 5.3 Color Space Conversion

SIMD optimized routines for the conversion from YCbCr to RGB as used by JPEG are readily available [29]. The compression format described in section 4, however, does not use the YCbCr color space. Instead the YCoCg color space is used which significantly reduces the computational complexity. Unlike the conversion from YCbCr to RGB, the conversion from YCoCg to RGB uses only addition and subtraction. The following matrix transformation shows the conversion from YCoCg to RGB.

$$\begin{aligned} \mathbf{R} &= [\ 1 \quad 1 \quad -1] \ [\mathbf{Y}] \\ \mathbf{G} &= [\ 1 \quad 0 \quad 1] \ [\mathbf{Co}] \\ \mathbf{B} &= [\ 1 \quad -1 \quad -1] \ [\mathbf{Cg}] \end{aligned}$$

This transform can be implemented with as few as two additions and two subtractions as shown below.

$$\begin{aligned} \mathbf{t} &= \mathbf{Y} - \mathbf{Cg} \\ \mathbf{R} &= \mathbf{t} + \mathbf{Co} \\ \mathbf{G} &= \mathbf{Y} + \mathbf{Cg} \\ \mathbf{B} &= \mathbf{t} - \mathbf{Co} \end{aligned}$$

However, the YCoCg color data is stored in 4:2:0 format. In other words there is an unique luma value for each pixel and there is one pair of chroma values for each 2x2 block of pixels. As such the conversion from 4:2:0 YCoCg to RGB is implemented differently where for each 2x2 block of pixels a pair of chroma values is converted to three values that can be added to each unique luma value. For each block of 2x2 pixels the following three variables are calculated.

$$\begin{aligned} \mathbf{r} &= \mathbf{Co} - \mathbf{Cg} \\ \mathbf{s} &= \mathbf{Cg} \\ \mathbf{t} &= \mathbf{Co} + \mathbf{Cg} \end{aligned}$$

Then for each pixel the RGB values are calculated as follows.

$$\begin{aligned} \mathbf{R} &= \mathbf{Y} + \mathbf{r} \\ \mathbf{G} &= \mathbf{Y} + \mathbf{s} \\ \mathbf{B} &= \mathbf{Y} - \mathbf{t} \end{aligned}$$

An implementation in C of the color conversion can be found in appendix F. This routine works on one 8x8 block of a 16x16 tile at a time. For each 8x8 block the routine works on two rows at a time. In other words the routine works on rows of 2x2 blocks of pixels. The routine also writes out a decompressed alpha channel for RGBA textures. For RGB only textures this alpha channel can be trivially removed and a constant value of 255 can

be written to the destination texture for the alpha channel. The RGBA values are written to the destination texture with clamping because the quantization and DCT transform may have distorted the YCoCg and alpha values such that the conversion back to RGBA results in values that are outside the [0,255] range. MMX and SSE2 implementations can be found in appendix G and H respectively.

# 6 Mip Mapping & DXT Compression

Mip maps are pre-filtered collections of downsampled textures that accompany a full resolution texture intended to reduce aliasing artifacts during rendering. When streaming textures from disk these mip maps can be stored compressed with the full resolution textures. Both the full resolution texture and the mip maps can then be streamed from disk and decompressed. However, instead of storing and streaming compressed mips maps it is usually faster to generate the mip maps from the decompressed full resolution texture. Once the full resolution texture is decompressed a simple box filter can used to create the mip maps.

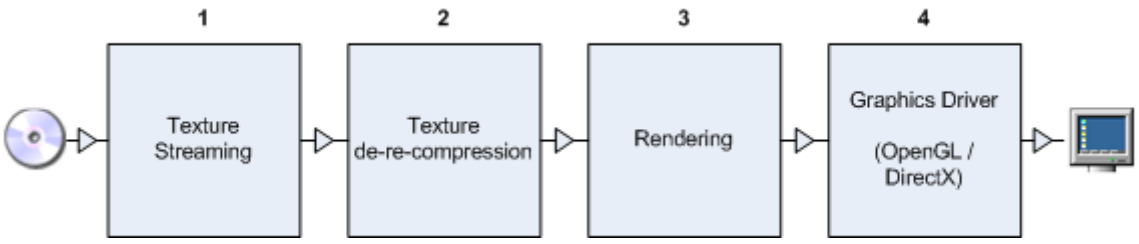
Most of today's graphics cards allow textures to be stored in a variety of compressed formats that are decompressed in hardware during rasterization. As previously described one such format which is supported by most graphics cards is S3TC also known as DXT compression [16, 17]. DXT compressed textures do not only require significantly less memory on the graphics card, they generally also render faster than uncompressed textures because of reduced bandwidth requirements. Some quality may be lost due to the DXT compression. However, when the same amount of memory is used on the graphics card there is generally a significant gain in quality.

The texture streaming solution presented here stores textures in a compression format very similar to JPEG. This format cannot be decompressed in hardware during rasterization on current graphics cards. However, it may still be desirable to save memory on the graphics card and improve the rendering performance by using textures that are stored in a compressed format that can be decompressed in hardware during rasterization. After streaming and decompressing a texture from disk and generating mip maps, the full resolution texture with mip maps can be compressed to DXT format in real-time as shown in [30]. On high end systems with more video memory available the high quality DXT compression as described in [30] can be used. On systems where video memory and performance are of no concern at all, the textures can be used without compression for the best visual quality.

# 7. Threaded Pipeline

The texture streaming solution presented here reads data from disk which is then decompressed and possibly re-compressed. If this process is serialized all steps in the pipeline add up and the throughput is limited by the time it takes to complete all the steps in the pipeline. Even with a very fast decompressor things add up and the throughput may not be sufficient to stream in detail at a rate which is high enough for high fidelity rendering.

A large texture database typically stores many smaller textures or the textures are broken up into many smaller tiles. As such it is possible to run different steps from the texture streaming pipeline in parallel as long as each step works on different data. The following image shows the full pipeline broken up in four threads going from streaming data from disk all the way to up to the graphics driver.




Current graphics drivers can either not be orchestrated from multiple threads or need to be synchronized first. As such there is only one thread talking to the graphics driver which is the renderer.

Threading the pipeline does not improve the latency for streaming individual textures. However, breaking the pipeline up in threads does significantly improve the throughput when continuously streaming texture data. The streaming, decompression and possibly re-compression of textures is broken up into two threads. With multi-threading the de-re-compression time is typically completely hidden by the time it takes to stream compressed data from disk because the de-re-compression is usually faster than reading data from disk, especially when streaming from a slow DVD drive. While new data is being read from disk the de-re-compression thread can decompress and recompress data that has already been read. Furthermore the streaming thread does not do a whole lot of work and mostly waits for the hard drive or DVD drive. In other words a lot of CPU time is available for the decompression thread while the streaming thread is waiting.

# 8. Results

The following images show the original 256x256 "Lena" image and the same image compressed with a 10:1 ratio (to 10% of the original size) and a 20:1 ratio (to 5% of the original size) using the compression format described in section 4.


original 256x256 "Lena" image


10:1 compressed 256x256 "Lena" image


20:1 compressed 256x256 "Lena" image

The streaming throughput of the decompressor described in this article is tested and compared with several decompressors for JPEG 2000, HD Photo and regular JPEG. The "JPEG 2000 JasPer" is version 1.701.0 of the JasPer JPEG 2000 decompressor [6]. This decompressor does not use any SIMD optimizations. The "JPEG 2000 OpenJPEG" is version 1.0 of the OpenJPEG [7] JPEG 2000 decompressor. Just like JasPer this decompressor does not use any SIMD optimizations. The "JPEG 2000 RV-Media" is the decompressor from the RV-Media Jpeg2000 SDK 1.0 Beta [8]. The "JPEG 2000 LeadTools" is the decompressor from the LeadTools Raster Imaging SDK Pro 14.5 [9]. The "JPEG 2000 J2K-Codec" is version 1.9 of the J2K-Codec [10] which is an SIMD optimized JPEG-2000 decompressor. The "JPEG 2000 Kakadu" is version 5.2.2 of the Kakadu JPEG 2000 decompressor [11] which is also SIMD optimized. The "HDPhoto .Net" is the HD Photo decompressor that comes with the Microsoft .Net Framework 3.0 [14]. The "HDPhoto Reference" is the reference implementation for embedded devices from the Microsoft DPK 1.0 [12]. The "JPEG IJG" is version 6b of the JPEG decompressor from the Independent JPEG Group [20]. This decompressor by default uses an integer LL&M iDCT and does not use any SIMD optimizations. The "JPEG IJG x86 SIMD" is the IJG JPEG library with x86 SIMD extensions by Miyasaka Masaru [21]. The "fast DCT" is the decompressor described in this article.

The different decompressors are tested using the 256x256 "Lena" image compressed at a 10:1 and a 20:1 ratio and a typical chroma sub-sampling ratio of 4:2:0. For the RGBA decompression the blue channel from the "Lena" image is replicated to the alpha channel. At a 10:1 compression ratio with 4:2:0 chroma sub-sampling all compressed images exhibit very good or comparable quality. At a 20:1 compression ratio all compressed

images show some visible loss in quality. The images compressed to JPEG 2000 or HD Photo typically show less noticeable loss in quality than the images compressed to JPEG or the compression format described here. The following table shows the RGB peak signal-to-noise ratio (PSNR) for the 10:1 and 20:1 compressed 256x256 "Lena" image using the different decompressors (higher = better).

| RGB PSNR | | |
|---|---|---|
| **decompressor** | **10:1 ratio** | **20:1 ratio** |
| JPEG 2000 JasPer | 41.6 | 38.5 |
| JPEG 2000 OpenJPEG | 41.6 | 38.5 |
| JPEG 2000 RV-Media | 41.6 | 38.5 |
| JPEG 2000 LeadTools | 41.6 | 38.5 |
| JPEG 2000 J2K-Codec | 41.4 | 38.4 |
| JPEG 2000 Kakadu | 41.5 | 38.5 |
| HDPhoto .Net | 41.4 | 39.0 |
| HDPhoto Reference | 41.4 | 39.0 |
| JPEG IJG | 39.9 | 36.6 |
| JPEG IJG x86 SIMD | 39.9 | 36.6 |
| fast DCT (C) | 40.1 | 37.0 |
| fast DCT (MMX optimized) | 40.1 | 37.0 |
| fast DCT (SSE2 optimized) | 40.1 | 37.0 |

The following tables show the decompression performance of several decompressors in MegaPixels per second (MP/s) on an Intel 2.8 GHz dual-core Xeon and an Intel 2.9 GHz Core 2 Extreme. The compressed images are decompressed as fast as possible from memory without generating mip maps or compression to DXT. The decompression from memory is done with hot cache to make the tests reproducible using the same CPUs with as little dependencies on the memory subsystem as possible.

| Throughput in Mega Pixels per second decompressing RGB from memory | | | | |
|---|---|---|---|---|
| | 10:1 ratio | | 20:1 ratio | |
| **decompressor** | **MP/s [1]** | **MP/s [2]** | **MP/s [1]** | **MP/s [2]** |
| JPEG 2000 JasPer | 0.55 | 1.72 | 0.61 | 1.92 |
| JPEG 2000 OpenJPEG | 0.64 | 1.34 | 0.72 | 1.55 |
| JPEG 2000 RV-Media | 1.05 | 2.89 | 1.28 | 3.56 |
| JPEG 2000 LeadTools | 2.65 | 5.96 | 3.40 | 7.65 |
| JPEG 2000 J2K-Codec | 3.58 | 8.01 | 4.88 | 11.05 |
| JPEG 2000 Kakadu | 3.42 | 10.53 | 4.37 | 15.67 |
| HDPhoto .Net | 8.60 | 17.07 | 10.01 | 20.24 |
| HDPhoto Reference | 10.69 | 18.70 | 12.52 | 22.54 |
| JPEG IJG | 25.37 | 44.08 | 30.42 | 53.14 |
| JPEG IJG x86 SIMD | 54.45 | 103.21 | 67.62 | 129.12 |
| fast DCT (C) | 30.74 | 48.72 | 34.40 | 55.21 |
| fast DCT (MMX optimized) | 70.49 | 125.24 | 88.82 | 170.71 |
| fast DCT (SSE2 optimized) | 83.44 | 131.47 | 117.43 | 190.14 |

| Throughput in Mega Pixels per second decompressing RGBA from memory | | | | |
|---|---|---|---|---|
| | 10:1 ratio | | 20:1 ratio | |
| **decompressor** | **MP/s [1]** | **MP/s [2]** | **MP/s [1]** | **MP/s [2]** |
| JPEG 2000 JasPer | 0.46 | 1.35 | 0.50 | 1.53 |
| JPEG 2000 OpenJPEG | 0.50 | 1.02 | 0.57 | 1.20 |
| JPEG 2000 RV-Media | 0.82 | 2.25 | 0.94 | 2.67 |
| JPEG 2000 LeadTools | 2.02 | 3.75 | 2.62 | 4.59 |
| JPEG 2000 J2K-Codec | 2.66 | 5.88 | 3.67 | 8.27 |
| JPEG 2000 Kakadu | 2.29 | 7.83 | 3.04 | 11.76 |
| HDPhoto .Net | 6.55 | 11.95 | 7.72 | 14.21 |
| HDPhoto Reference | 7.13 | 12.75 | 8.40 | 15.33 |
| JPEG IJG | NA | NA | NA | NA |
| JPEG IJG x86 SIMD | NA | NA | NA | NA |
| fast DCT (C) | 21.20 | 33.79 | 23.95 | 38.01 |
| fast DCT (MMX optimized) | 46.81 | 84.19 | 60.70 | 117.31 |
| fast DCT (SSE2 optimized) | 57.12 | 90.95 | 78.75 | 132.80 |

[1] Intel 2.8 GHz Dual-Core Xeon ("Paxville" 90nm NetBurst microarchitecture)

[2] Intel 2.9 GHz Core 2 Extreme ("Conroe" 65nm Core 2 microarchitecture)

The complete streaming solution is tested with a 12× DVD drive, with a peak outer edge throughput close to 15 MB/s. Ignoring the seek times, the peak outer edge throughput is equivalent to 5.2 RGB MP/s or 3.9 RGBA MP/s when streaming uncompressed texture data. The throughput increases significantly when using a 10:1 or 20:1 compression ratio and the SSE2 implementation of the decompressor described here. The following tables show the peak throughput when streaming and decompressing from the 12× DVD drive without using a multi-threaded pipeline.

| Throughput in Mega Pixels per second streaming & decompressing RGB from a 12× DVD | | | | |
|---|---|---|---|---|
| | 10:1 ratio | | 20:1 ratio | |
| **decompressor** | **MP/s [1]** | **MP/s [2]** | **MP/s [1]** | **MP/s [2]** |
| JPEG 2000 JasPer | 0.54 | 1.67 | 0.61 | 1.89 |
| JPEG 2000 OpenJPEG | 0.63 | 1.31 | 0.72 | 1.53 |
| JPEG 2000 RV-Media | 1.03 | 2.74 | 1.26 | 3.44 |
| JPEG 2000 LeadTools | 2.52 | 5.35 | 3.29 | 7.13 |
| JPEG 2000 J2K-Codec | 3.35 | 6.95 | 4.66 | 10.00 |
| JPEG 2000 Kakadu | 3.21 | 8.77 | 4.20 | 13.63 |
| HDPhoto .Net | 7.39 | 12.88 | 9.14 | 16.97 |
| HDPhoto Reference | 8.88 | 13.78 | 11.18 | 18.55 |
| JPEG IJG | 17.10 | 23.95 | 23.58 | 35.27 |
| JPEG IJG x86 SIMD | 26.71 | 34.77 | 41.11 | 57.87 |
| fast DCT (C) | 19.38 | 25.25 | 25.90 | 36.17 |
| fast DCT (MMX optimized) | 30.07 | 36.96 | 48.09 | 64.96 |
| fast DCT (SSE2 optimized) | 32.20 | 37.48 | 55.39 | 67.59 |

| Throughput in Mega Pixels per second streaming & decompressing RGBA from a 12× DVD | | | | |
|---|---|---|---|---|
| | 10:1 ratio | | 20:1 ratio | |
| **decompressor** | **MP/s [1]** | **MP/s [2]** | **MP/s [1]** | **MP/s [2]** |
| JPEG 2000 JasPer | 0.45 | 1.31 | 0.50 | 1.50 |
| JPEG 2000 OpenJPEG | 0.49 | 0.99 | 0.57 | 1.81 |
| JPEG 2000 RV-Media | 0.80 | 2.13 | 0.93 | 2.58 |
| JPEG 2000 LeadTools | 1.92 | 3.42 | 2.54 | 4.34 |
| JPEG 2000 J2K-Codec | 2.49 | 5.12 | 3.51 | 7.48 |
| JPEG 2000 Kakadu | 2.16 | 6.53 | 2.93 | 10.23 |
| HDPhoto .Net | 5.61 | 9.16 | 7.03 | 12.04 |
| HDPhoto Reference | 6.04 | 9.63 | 7.59 | 12.83 |
| JPEG IJG | NA | NA | NA | NA |
| JPEG IJG x86 SIMD | NA | NA | NA | NA |
| fast DCT (C) | 13.77 | 18.17 | 18.36 | 25.62 |
| fast DCT (MMX optimized) | 21.37 | 26.80 | 34.26 | 47.08 |
| fast DCT (SSE2 optimized) | 23.29 | 27.45 | 39.35 | 49.39 |

[1] Intel 2.8 GHz Dual-Core Xeon ("Paxville" 90nm NetBurst microarchitecture)

[2] Intel 2.9 GHz Core 2 Extreme ("Conroe" 65nm Core 2 microarchitecture)

With a threaded pipeline the throughput increases significantly and the streaming solution presented here is typically limited by the DVD throughput as shown in the following tables.

Throughput in Mega Pixels per second
streaming & decompressing RGB from a 12× DVD
using a threaded pipeline

| | 10:1 ratio | | 20:1 ratio | |
|---|---|---|---|---|
| **decompressor** | **MP/s [1]** | **MP/s [2]** | **MP/s [1]** | **MP/s [2]** |
| JPEG 2000 JasPer | 0.55 | 1.72 | 0.61 | 1.92 |
| JPEG 2000 OpenJPEG | 0.64 | 1.34 | 0.72 | 1.55 |
| JPEG 2000 RV-Media | 1.05 | 2.89 | 1.28 | 3.56 |
| JPEG 2000 LeadTools | 2.65 | 5.96 | 3.40 | 7.65 |
| JPEG 2000 J2K-Codec | 3.58 | 8.01 | 4.88 | 11.05 |
| JPEG 2000 Kakadu | 3.42 | 10.53 | 4.37 | 15.67 |
| HDPhoto .Net | 8.60 | 17.07 | 10.01 | 20.24 |
| HDPhoto Reference | 10.69 | 18.70 | 12.52 | 22.54 |
| JPEG IJG | 25.37 | 44.08 | 30.42 | 53.14 |
| JPEG IJG x86 SIMD | 52.43 | 52.43 | 67.62 | 104.86 |
| fast DCT (C) | 30.74 | 48.72 | 34.40 | 55.21 |
| fast DCT (MMX optimized) | 52.43 | 52.43 | 88.82 | 104.86 |
| fast DCT (SSE2 optimized) | 52.43 | 52.43 | 104.86 | 104.86 |

Throughput in Mega Pixels per second
streaming & decompressing RGBA from a 12× DVD
using a threaded pipeline

| | 10:1 ratio | | 20:1 ratio | |
|---|---|---|---|---|
| **decompressor** | **MP/s [1]** | **MP/s [2]** | **MP/s [1]** | **MP/s [2]** |
| JPEG 2000 JasPer | 0.46 | 1.35 | 0.50 | 1.53 |
| JPEG 2000 OpenJPEG | 0.50 | 1.02 | 0.57 | 1.20 |
| JPEG 2000 RV-Media | 0.82 | 2.25 | 0.94 | 2.67 |
| JPEG 2000 LeadTools | 2.02 | 3.75 | 2.62 | 4.59 |
| JPEG 2000 J2K-Codec | 2.66 | 5.88 | 3.67 | 8.27 |
| JPEG 2000 Kakadu | 2.29 | 7.83 | 3.04 | 11.76 |
| HDPhoto .Net | 6.55 | 11.95 | 7.72 | 14.21 |
| HDPhoto Reference | 7.13 | 12.75 | 8.40 | 15.33 |
| JPEG IJG | NA | NA | NA | NA |
| JPEG IJG x86 SIMD | NA | NA | NA | NA |
| fast DCT (C) | 21.20 | 33.79 | 23.95 | 38.01 |
| fast DCT (MMX optimized) | 39.32 | 39.32 | 60.70 | 78.64 |
| fast DCT (SSE2 optimized) | 39.32 | 39.32 | 78.64 | 78.64 |

[1] Intel 2.8 GHz Dual-Core Xeon ("Paxville" 90nm NetBurst microarchitecture)

[2] Intel 2.9 GHz Core 2 Extreme ("Conroe" 65nm Core 2 microarchitecture)

The above tables show that without SIMD optimizations the streaming of texture data with a 10:1 or 20:1 compression ratio is limited not by the DVD throughput but by the decompressor throughput. However, with SIMD optimizations the throughput is only limited by the throughput of the DVD drive and the decompression time is completely hidden. At higher compression ratios the streaming performance typically improves. Not only are the bandwidth requirements reduced, the decompression becomes faster as well because at higher compression ratios an entropy decoder has to decode fewer bits. All tests shown here are with the 256x256 "Lena" image. For other images the RGB-PSNR may be different when using to the various compression formats. However, the performance of the different decompressors is mostly dependent on the compression ratio and typically varies very little with different images.

It is interesting to relate the streaming performance to realistic rendering of a walk/run/drive-through over uniquely textured terrain. To simplify the calculations the view point is assumed to be moving over a flat terrain and the texture detail is displayed in square layers around the view point. Each layer around the view point displays the same number of pixels but a lower detail layer is twice the size in world coordinates than the higher detail layer directly above it. To render the flat terrain at a decent resolution of 1024 x 768 or higher, a typical resolution of 2048 x 2048 pixels is used for each square layer. If there are five square layers of 2048 x 2048 pixels there are a total of 5 x 2048 x 2048 pixels (about 21 Mega Pixels) available for rendering at any time. As the view point

moves the layers have to be updated and essentially rows and columns of pixels have to be updated at the side(s) of the square layers in the direction in which the view point moves. In the worst case the view point moves along a diagonal and both a row and column of pixels need to be updated. Assuming there are four pixels per square inch on the highest detail layer, this amounts to 11209 pixels that have to be updated per inch of movement along the diagonal.

The average walking speed for a human is about 3 miles per hour which equals about 53 inches per second. At this speed about 0.59 mega pixels have to be streamed per second to update the texture detail around the view point. A very fast human can run at a speed of up to 20 miles per hour which equals about 352 inches per second. At this speed about 3.95 mega pixels have to be streamed per second. When driving a car at 80 miles per hour about 15.78 mega pixels have to be streamed per second. These numbers are for rendering a terrain which is completely flat without any elevation. For a terrain with hills and mountains the amount of texture detail that needs to be updated can be up to 1.5 times higher.

Although the streaming pipeline is threaded, none of the decompressors tested above is using multiple threads. During the streaming tests the CPU is doing no other work than decompressing data. On single CPU/core systems there is usually only a small percentage of the CPU available for texture streaming and decompression because the CPU is typically already taxed with other things like rendering. Furthermore mip maps may have to be generated for the streamed textures and the textures including mip maps may need to be compressed to DXT format which also consumes CPU time. In other words on systems with few CPUs/cores it is even more important to use a very fast SIMD optimized decompressor in order to stream in texture data at a rate high enough for high fidelity rendering.

# 9. Conclusion

On today's computers, especially computers with multiple CPUs or cores, real-time streaming of vast amounts of texture data can be achieved by using compression and a high performance SIMD optimized decompressor. Furthermore the streaming throughput can be improved significantly by using multi-threading to break up the texture streaming pipeline into multiple steps that can run in parallel.

# 10. Future Work

As faster CPUs and systems with more CPUs or cores become available it will become advantageous to use compression formats that achieve better quality and higher compression ratios at the cost of more expensive decompression. As more CPU time becomes available compression formats like JPEG 2000 and HD Photo typically achieve acceptable quality at higher compression ratios and as such improve the streaming throughput as long as the throughput is not limited by the time required for decompression.

# 11. References


1. JPEG: Still Image Data Compression Standard
William B., Pennebaker, Joan L., Mitchell
Van Nostrand Reinhold, New York 1993
Available Online: http://www.amazon.com/JPEG-Compression-Standard-Multimedia-Standards/dp/0442012721

2. The JPEG Still Image Data Compression Standard
Gregory K. Wallace
Special issue on digital multimedia systems, Volume 34, Issue 4, Pages: 30 - 44, 1991
Available Online: http://portal.acm.org/citation.cfm?id=103089

3. JPEG-2000
Joint Photographic Experts Group
ISO/IEC 15444-1:2004, March 2000
Available Online: http://www.jpeg.org/jpeg2000/

4. An Overview of JPEG-2000
Michael W. Marcellin, Michael J. Gormish, Ali Bilgin, Martin P. Boliek
Proc. of IEEE Data Compression Conference, pp. 523-541, 2000
Available Online: http://rii.ricoh.com/~gormish/pdf/dcc2000_jpeg2000_note.pdf

5. JPEG 2000 Image Codecs Comparison
Dmitriy Vatolin, Alexey Moskvin, Oleg Petrov, Artem Titarenko
CS MSU Graphics & Media Lab Video Group, September 2005
Available Online: http://www.compression.ru/video

6. JasPer
Michael David Adams
The JasPer Project version 1.701.0, February 2004
Available Online: http://www.ece.uvic.ca/~mdadams/jasper/

7. OpenJPEG
David Janssens, Yannick Verschueren, Francois Devaux, Antonin Descampe, HervZ Drolon, FreeImage Team
Communications and remote sensing Laboratory, Universite catholique de Louvain, Belgium, version 1.0, December 2005
Available Online:http:/www.tele.ucl.ac.be/PROJECTS/OPENJPEG/

8. RV-Media Jpeg2000 SDK
RV-Media Ltd.
Jpeg2000 SDK version 1.0 Beta, July 2006
Available Online: http://www.rv-media.net/content/blogcategory/15/36/

9. LeadTools Raster Imaging
LeadTools
Raster Imaging SDK Pro 14.5, June 2006
Available Online: http://www.leadtools.com/SDK/Raster/Raster-Imaging.htm

10. J2K-Codec
Alex Saveliev
J2K-Codec version 1.9, May 2006
Available Online: http://j2k-codec.com/

11. Kakadu JPEG 2000
David Taubman
Kakadu, version 5.2.2, July 2006
Available Online: http://www.kakadusoftware.com/

12. HD Photo
Microsoft
Microsoft, May 2006
Available Online:http:/www.microsoft.com/windows/windowsmedia/forpros/wmphoto/default.aspx

13. Windows Media Photo and JPEG 2000 Codecs Comparison
Dmitriy Vatolin, Alexey Moskvin, Oleg Petrov
CS MSU Graphics & Media Lab Video Group, August 2006
Available Online: http://www.compression.ru/video

14. HD Photo Codec
Microsoft
.Net Framework 3.0, October 2006
Available Online: http://www.microsoft.com/downloads/details.aspx?FamilyID=a6e324c4-ae01-4725-a92e-2b38beaf34c0&DisplayLang=en

15. Discrete Cosine Transform Shader
nVidia
Featured Code Samples at developer.nvidia.com, September 23rd 2005
Available Online:
http://download.developer.nvidia.com/developer/SDK/Individual_Samples/featured_samples.html#gpgpu_dct

16. S3 Texture Compression
Pat Brown
NVIDIA Corporation, November 2001
Available Online: http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt

17. Compressed Texture Resources
Microsoft Developer Network
DirectX SDK, April 2006
Available Online: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Compressed_Texture_Formats.asp

18. Transform, Scaling & Color Space Impact of Professional Extensions
H. S. Malvar, G. J. Sullivan
ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.6 Document JVT-H031, Geneva, May 2003
Available Online: http://ftp3.itu.int/av-arch/jvt-site/2003_05_Geneva/JVT-H031.doc

19. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range
H. S. Malvar, G. J. Sullivan
Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, Document No. JVT-I014r3, July 2003
Available Online: http://research.microsoft.com/~malvar/papers/JVT-I014r3.pdf

20. Free Library for JPEG Image Compression
Independent JPEG Group
IJG, March 27th 1998
Available Online: http://www.ijg.org/

21. IJG JPEG library with x86 SIMD extensions
Miyasaka Masaru
Softlab, version 1.02, February 2006
Available Online: http://cetus.sakura.ne.jp/softlab/jpeg-x86simd/jpegsimd.html

22. Using MMX™ Instructions to Get Bits From a Data Stream
Intel
Intel® Developer Services, March 1996
Available Online:

23. Avoiding the Cost of Branch Misprediction
Rajiv Kapoor
Intel, December 2002
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/19952.htm

24. Branch and Loop Reorganization to Prevent Mispredicts
Jeff Andrews
Intel, January 2004
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/ia32/pentium4/optimization/66779.htm

25. IEEE Standard Specification for the Implementation of 8x8 Inverse Discrete Cosine Transform
IEEE Std. 1180-1990
Institute of Electrical and Electronics Engineers, Inc, 1990
Available Online: http://ieeexplore.ieee.org/iel1/2259/4127/00159237.pdf?arnumber=159237

26. Using MMX™ Instructions in a Fast iDCT Algorithm for MPEG Decoding
Intel
Intel® Developer Services, March 1996
Available Online:

27. A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX™ Instructions
Intel
Intel Application Note, AP-922, Version 1.0, June 4th 1999
Available Online: http://citeseer.ist.psu.edu/697580.html

28. Using Streaming SIMD Extensions 2 (SSE2) to Implement an Inverse Discrete Cosine Transform
Intel
Intel Application Note, AP-945, Version 2.0, September 21st 2000
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/code/optimization/193014.htm

29. Color Conversion from YUV12 to RGB Using Intel MMX™ Technology
Intel
Intel® Developer Services, March 1996
Available Online:

30. Real-Time DXT Compression
J.M.P. van Waveren
Intel Software Network, October 2006
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/324337.htm

# Appendix A

```
/*
   Decompression Of One Tile
   Copyright (C) 2006 Id Software, Inc.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

struct HuffmanTable {
   ...
};

HuffmanTable huffTableYDC;
HuffmanTable huffTableYAC;
HuffmanTable huffTableCoCgDC;
HuffmanTable huffTableCoCgAC;
HuffmanTable huffTableADC;
HuffmanTable huffTableAAC;

unsigned short quantTableY[]       = { ... };
unsigned short quantTableCoCg[]    = { ... };
unsigned short quantTableA[]       = { ... };

int dcY;
int dcCo;
int dcCg;
int dcA;

void DecompressTileRGBA( byte *rgba, int stride ) {
   ALIGN16( short YCoCgA[10*64] );

   // Y: 4 blocks, Co: 1 block, Cg: 1 block
   HuffmanDecode( YCoCgA + 0*64, huffTableYDC, huffTableYAC, &dcY );
   HuffmanDecode( YCoCgA + 1*64, huffTableYDC, huffTableYAC, &dcY );
   HuffmanDecode( YCoCgA + 2*64, huffTableYDC, huffTableYAC, &dcY );
   HuffmanDecode( YCoCgA + 3*64, huffTableYDC, huffTableYAC, &dcY );
   HuffmanDecode( YCoCgA + 4*64, huffTableCoCgDC, huffTableCoCgAC, &dcCo );
   HuffmanDecode( YCoCgA + 5*64, huffTableCoCgDC, huffTableCoCgAC, &dcCg );

   // Inverse DCT of YCoCg channels
   IDCT( YCoCgA + 0*64, quantTableY, YCoCgA + 0*64 );
   IDCT( YCoCgA + 1*64, quantTableY, YCoCgA + 1*64 );
   IDCT( YCoCgA + 2*64, quantTableY, YCoCgA + 2*64 );
   IDCT( YCoCgA + 3*64, quantTableY, YCoCgA + 3*64 );
   IDCT( YCoCgA + 4*64, quantTableCoCg, YCoCgA + 4*64 );
   IDCT( YCoCgA + 5*64, quantTableCoCg, YCoCgA + 5*64 );

   // Alpha: 4 blocks
   HuffmanDecode( YCoCgA + 6*64, huffTableADC, huffTableAAC, &dcA );
   HuffmanDecode( YCoCgA + 7*64, huffTableADC, huffTableAAC, &dcA );
   HuffmanDecode( YCoCgA + 8*64, huffTableADC, huffTableAAC, &dcA );
   HuffmanDecode( YCoCgA + 9*64, huffTableADC, huffTableAAC, &dcA );
```

```
    // Inverse DCT of Alpha channel
    IDCT( YCoCgA + 6*64, quantTableA, YCoCgA + 6*64 );
    IDCT( YCoCgA + 7*64, quantTableA, YCoCgA + 7*64 );
    IDCT( YCoCgA + 8*64, quantTableA, YCoCgA + 8*64 );
    IDCT( YCoCgA + 9*64, quantTableA, YCoCgA + 9*64 );

    // Color conversion
    YCoCgAToRGBA( YCoCgA, rgba, stride );
}
```

# Appendix B

```
/*
   Run-Length and Huffman Decoding of DCT Coefficients
   Copyright (C) 2006 Id Software, Inc.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

int jpeg_natural_order[64+16]      = { ... };

const int HUFF_WORD_SIZE    = 8;    // symbol size in bits
const int HUFF_MAXBITS      = 16;   // maximum number of bits in any code
const int HUFF_LOOKUPBITS   = 8;    // lookup table for codes with less than this number of bits

struct HuffmanTable {
   int            minCode[HUFF_MAXBITS+1];                  // minCode[k] is smallest code of length k
   int            symOffset[HUFF_MAXBITS+1];                // symOffset[k] is index into symbols[] of 1st symbol of length k
   unsigned char  symbols[1<<HUFF_WORD_SIZE];               // symbols in order of increasing code length
   unsigned char  look_nbits[1<<HUFF_LOOKUPBITS];           // number bits for codes with no more than HUFF_LOOKUPBITS bits
   unsigned char  look_sym[1<<HUFF_LOOKUPBITS];             // symbol for codes with no more than HUFF_LOOKUPBITS bits
   int            test_nbits[HUFF_MAXBITS];                 // codes left justified to 16 bits larger equal test_nbits[k] have k or more bits
};

int getBits;
int getBuff;
int dataBytes;
const byte * data;

void HuffmanDecode( short *coef, const HuffmanTable &dctbl, const HuffmanTable &actbl, int *lastDC ) {
   int s, k, r, t;

   memset( coef, 0, 64 * sizeof( short ) );

   s = GetCategory( dctbl );          // get DC category number

   if ( s != 0 ) {
      r = GetBits( s );               // get offset in this DC category
      s = ValueFromCategory( s, r );  // get DC difference value
   }

   s += *lastDC;
   *lastDC = s;

   coef[0] = (short) s;

   for ( k = 1; k < 64; k++ ) {
      s = GetCategory( actbl );       // s: (run, category)
      t = s & 15;                     // t: category for this non-zero AC
      r = s >> 4;                     // r: run length for zero AC, 0 <= r < 16
```

```
      k += r;

      if ( t != 0 ) {
         r = GetBits( t );                  // get offset in this AC category
         s = ValueFromCategory( t, r );     // get AC value
         coef[ jpeg_natural_order[ k ] ] = (short) s;
      } else {
         if ( r != 15 ) {
            break;                          // all the remaining AC values are zero
         }
      }
   }
}

void FillBitBuffer( void ) {
   assert( getBits <= 15 );
   dataBytes -= 2;
   int s = ( ~( ( unsigned int ) dataBytes ) ) >> 31;
   getBuff = ( getBuff << 16 ) | ((int) data[0] << 8) | ((int) data[s]);
   getBits += 16;
   data += 2*s;    // repeat last byte if at the end
}

inline int GetBits( int bits ) {
   assert( bits <= 16 );
   if( getBits < bits ) {
      FillBitBuffer();
   }
   getBits -= bits;
   return ( getBuff >> getBits ) & ( ( 1 << bits ) - 1 );
}

inline int PeekBits( int bits ) {
   assert( bits <= 16 );
   if( getBits < bits ) {
      FillBitBuffer();
   }
   return ( getBuff >> ( getBits - bits ) ) & ( ( 1 << bits ) - 1 );
}

inline int GetCategory( const HuffmanTable &htbl ) {
   // Peek the first HUFF_LOOKUPBITS bits.
   // FillBitBuffer will repeat the last byte when trying to read beyond the end of the stream.
   // However, the first bits we peek here are still valid and the lookup table will work as intended.
   int look = PeekBits( HUFF_LOOKUPBITS );

   // Lookup the number of bits for this Huffman code.
   int nb = htbl.look_nbits[look];

   // If this is a huffman code of HUFF_LOOKUPBITS or less bits.
   if ( nb != 0 ) {
      getBits -= nb;
      return htbl.look_sym[look];
   } else {
      // Decode long codes with length >= HUFF_LOOKUPBITS.
      return DecodeLong( htbl );
   }
}

int DecodeLong( const HuffmanTable &htbl ) {
   // Peek the maximum number of bits for a code.
```

```
    int look = PeekBits( HUFF_MAXBITS );

    // Find out how many bits are actually used.
    int nb = HUFF_LOOKUPBITS + 1;
    for ( int i = HUFF_LOOKUPBITS + 1; i < HUFF_MAXBITS; i++ ) {
        int b = ( look >= htbl.test_nbits[i] );
        nb += b;
    }

    getBits -= nb;
    look >>= HUFF_MAXBITS - nb;

    return htbl.symbols[ htbl.symOffset[nb] + look - htbl.minCode[nb] ];
}

inline int ValueFromCategory( int category, int offset ) {
    // return ((offset) < (1<<((category)-1)) ? (offset) + (((-1)<<(category)) + 1) : (offset))
    int m = 1 << category;
    return offset + ( ( ( offset - ( m >> 1 ) ) >> 31 ) & ( 1 - m ) );
}
```

# Appendix C

```
/*
   Integer Inverse Discrete Cosine Transform
   Copyright (C) 2006 Id Software, Inc.
   Original AP922 algorithm is Copyright (C) 1999 - 2000 Intel Corporation.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

#define BITS_INV_ACC          5                     // 4 or 5 for IEEE
#define SHIFT_INV_ROW         ( 16 - BITS_INV_ACC )
#define SHIFT_INV_COL         ( 1 + BITS_INV_ACC )

#define RND_INV_ROW           1024 * (6 - BITS_INV_ACC)   // 1 << (SHIFT_INV_ROW-1)
#define RND_INV_COL           16 * (BITS_INV_ACC - 3)     // 1 << (SHIFT_INV_COL-1)

#define SHIFT_ROUND_ROW( x )    ( (x) >> (SHIFT_INV_ROW) )
#define SHIFT_ROUND_COL( x )    ( (x) >> (SHIFT_INV_COL) )

#define BIAS_SCALE( X )       ( X / ( BITS_INV_ACC - 3 ) )

#define f_tg_1_16             tan( 1.0 * M_PI / 16.0 )
#define f_tg_2_16             tan( 2.0 * M_PI / 16.0 )
#define f_tg_3_16             tan( 3.0 * M_PI / 16.0 )

#define f_cos_1_16            cos( 1.0 * M_PI / 16.0 )
#define f_cos_2_16            cos( 2.0 * M_PI / 16.0 )
#define f_cos_3_16            cos( 3.0 * M_PI / 16.0 )
#define f_cos_4_16            cos( 4.0 * M_PI / 16.0 )
#define f_cos_5_16            cos( 5.0 * M_PI / 16.0 )
#define f_cos_6_16            cos( 6.0 * M_PI / 16.0 )
#define f_cos_7_16            cos( 7.0 * M_PI / 16.0 )

#define FIX16( x )            (unsigned short) (x * (1<<16) + 0.5)
#define FIX15_COS_1_16( x )     (short) (x * f_cos_1_16 * (1<<15) + 0.5)
#define FIX15_COS_2_16( x )     (short) (x * f_cos_2_16 * (1<<15) + 0.5)
#define FIX15_COS_3_16( x )     (short) (x * f_cos_3_16 * (1<<15) + 0.5)
#define FIX15_COS_4_16( x )     (short) (x * f_cos_4_16 * (1<<15) + 0.5)

const unsigned short tg_1_16    = FIX16( f_tg_1_16 );
const unsigned short tg_2_16    = FIX16( f_tg_2_16 );
const unsigned short tg_3_16    = FIX16( f_tg_3_16 );
const unsigned short cos_4_16   = FIX16( f_cos_4_16 );

#define INIT_TABLE( FF )                                        \
   FF(f_cos_4_16),  FF(f_cos_2_16),  FF(f_cos_4_16),  FF(f_cos_6_16),  \
   FF(f_cos_4_16),  FF(f_cos_6_16), -FF(f_cos_4_16), -FF(f_cos_2_16),  \
   FF(f_cos_4_16), -FF(f_cos_6_16), -FF(f_cos_4_16),  FF(f_cos_2_16),  \
   FF(f_cos_4_16), -FF(f_cos_2_16),  FF(f_cos_4_16), -FF(f_cos_6_16),  \
```

```
                                                \
   FF(f_cos_1_16),  FF(f_cos_3_16),  FF(f_cos_5_16),  FF(f_cos_7_16),  \
   FF(f_cos_3_16), -FF(f_cos_7_16), -FF(f_cos_1_16), -FF(f_cos_5_16),  \
   FF(f_cos_5_16), -FF(f_cos_1_16),  FF(f_cos_7_16),  FF(f_cos_3_16),  \
   FF(f_cos_7_16), -FF(f_cos_5_16),  FF(f_cos_3_16), -FF(f_cos_1_16)

static const short tab_i_04[32] = {
   INIT_TABLE( FIX15_COS_4_16 )
};

static const short tab_i_17[32] = {
   INIT_TABLE( FIX15_COS_1_16 )
};

static const short tab_i_26[32] = {
   INIT_TABLE( FIX15_COS_2_16 )
};

static const short tab_i_35[32] = {
   INIT_TABLE( FIX15_COS_3_16 )
};

static const short *inverseRowTables[] = {
   tab_i_04, tab_i_17, tab_i_26, tab_i_35,
   tab_i_04, tab_i_35, tab_i_26, tab_i_17
};

static const unsigned int rounder[8] = {
                RND_INV_ROW - BIAS_SCALE( 2048 ) + 65536,
                RND_INV_ROW + BIAS_SCALE( 3755 ),
                RND_INV_ROW + BIAS_SCALE( 2472 ),
                RND_INV_ROW + BIAS_SCALE( 1361 ),
                RND_INV_ROW + BIAS_SCALE( 0 ),
                RND_INV_ROW - BIAS_SCALE( 1139 ),
                RND_INV_ROW - BIAS_SCALE( 1024 ),
                RND_INV_ROW - BIAS_SCALE( 1301 )
};

#define PMULHW( X, Y )          ((short)(((int)(X)*(Y))>>16 ))
#define DEQUANTIZE( X, Q )      ((X)*(Q))

void IDCT( const short *coeff, const unsigned short *quant, short *dest ) {

   for( int i = 0; i < 8; i++ ) {
      const short *x = &coeff[ i*8 ];
      short *y = &dest[ i*8 ];
      const short *w = inverseRowTables[i];

      short x0 = DEQUANTIZE( x[0], quant[i*8+0] );
      short x1 = DEQUANTIZE( x[1], quant[i*8+1] );
      short x2 = DEQUANTIZE( x[2], quant[i*8+2] );
      short x3 = DEQUANTIZE( x[3], quant[i*8+3] );

      short x4 = DEQUANTIZE( x[4], quant[i*8+4] );
      short x5 = DEQUANTIZE( x[5], quant[i*8+5] );
      short x6 = DEQUANTIZE( x[6], quant[i*8+6] );
      short x7 = DEQUANTIZE( x[7], quant[i*8+7] );

      int a0 = x0 * w[ 0] + x2 * w[ 1] + x4 * w[ 2] + x6 * w[ 3];
      int a1 = x0 * w[ 4] + x2 * w[ 5] + x4 * w[ 6] + x6 * w[ 7];
      int a2 = x0 * w[ 8] + x2 * w[ 9] + x4 * w[10] + x6 * w[11];
      int a3 = x0 * w[12] + x2 * w[13] + x4 * w[14] + x6 * w[15];
```

```
    int b0 = x1 * w[16] + x3 * w[17] + x5 * w[18] + x7 * w[19];
    int b1 = x1 * w[20] + x3 * w[21] + x5 * w[22] + x7 * w[23];
    int b2 = x1 * w[24] + x3 * w[25] + x5 * w[26] + x7 * w[27];
    int b3 = x1 * w[28] + x3 * w[29] + x5 * w[30] + x7 * w[31];

    a0 += rounder[i];                   /* + RND_INV_ROW; */
    a1 += rounder[i];                   /* + RND_INV_ROW; */
    a2 += rounder[i];                   /* + RND_INV_ROW; */
    a3 += rounder[i];                   /* + RND_INV_ROW; */

    y[0] = SHIFT_ROUND_ROW( a0 + b0 );
    y[1] = SHIFT_ROUND_ROW( a1 + b1 );
    y[2] = SHIFT_ROUND_ROW( a2 + b2 );
    y[3] = SHIFT_ROUND_ROW( a3 + b3 );

    y[4] = SHIFT_ROUND_ROW( a3 - b3 );
    y[5] = SHIFT_ROUND_ROW( a2 - b2 );
    y[6] = SHIFT_ROUND_ROW( a1 - b1 );
    y[7] = SHIFT_ROUND_ROW( a0 - b0 );
  }

  for( int i = 0; i < 8; i++ ) {
    short *x = &dest[ i ];
    short *y = &dest[ i ];

    short tp765 =   x[1*8] + PMULHW( x[7*8], tg_1_16 );
    short tp465 = - x[7*8] + PMULHW( x[1*8], tg_1_16 );
    short tm765 =   x[3*8] + PMULHW( x[5*8], tg_3_16 );
    short tm465 =   x[5*8] - PMULHW( x[3*8], tg_3_16 );

    short t7    = tp765 + tm765;                /* + 1; // correction +1.0 */
    short tp65  = tp765 - tm765;
    short t4    = tp465 + tm465;
    short tm65  = tp465 - tm465;                /* + 1; // correction +1.0 */

    short t6    = PMULHW( tp65 + tm65, cos_4_16 );  /* | 1; // correction +0.5 */
    short t5    = PMULHW( tp65 - tm65, cos_4_16 );  /* | 1; // correction +0.5 */

    short tp03  = x[0*8] + x[4*8];
    short tp12  = x[0*8] - x[4*8];

    short tm03  = PMULHW( x[6*8], tg_2_16 ) + x[2*8];
    short tm12  = PMULHW( x[2*8], tg_2_16 ) - x[6*8];

    short t0    = tp03 + tm03;                  /* + RND_INV_COL; */
    short t3    = tp03 - tm03;                  /* + RND_INV_COL - 1; // correction -1.0 */
    short t1    = tp12 + tm12;                  /* + RND_INV_COL; */
    short t2    = tp12 - tm12;                  /* + RND_INV_COL - 1; // correction -1.0 */

    y[0*8] = SHIFT_ROUND_COL( t0 + t7 );
    y[1*8] = SHIFT_ROUND_COL( t1 + t6 );
    y[2*8] = SHIFT_ROUND_COL( t2 + t5 );
    y[3*8] = SHIFT_ROUND_COL( t3 + t4 );

    y[4*8] = SHIFT_ROUND_COL( t3 - t4 );
    y[5*8] = SHIFT_ROUND_COL( t2 - t5 );
    y[6*8] = SHIFT_ROUND_COL( t1 - t6 );
    y[7*8] = SHIFT_ROUND_COL( t0 - t7 );
  }
}
```

# Appendix D

```
/*
   MMX Optimized Integer Inverse Discrete Cosine Transform
   Copyright (C) 2006 Id Software, Inc.
   Original AP922 algorithm is Copyright (C) 1999 - 2000 Intel Corporation.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

#define __ALIGN8                __declspec(align(8))

#define BITS_INV_ACC            5                       // 4 or 5 for IEEE
#define SHIFT_INV_ROW           16 - BITS_INV_ACC
#define SHIFT_INV_COL           1 + BITS_INV_ACC

#define RND_INV_ROW             1024 * (6 - BITS_INV_ACC)   // 1 << (SHIFT_INV_ROW-1)
#define RND_INV_COL             16 * (BITS_INV_ACC - 3)     // 1 << (SHIFT_INV_COL-1)

#define DUP2( X )               (X),(X)
#define DUP4( X )               (X),(X),(X),(X)
#define BIAS_SCALE( X )         ( X / ( BITS_INV_ACC - 3 ) )

__ALIGN8 static short tg_1_16[4]    = { DUP4(  13036 ) };       // tg * (1<<16) + 0.5f
__ALIGN8 static short tg_2_16[4]    = { DUP4(  27146 ) };       // tg * (1<<16) + 0.5f
__ALIGN8 static short tg_3_16[4]    = { DUP4( -21746 ) };       // tg * (1<<16) + 0.5f
__ALIGN8 static short cos_4_16[4]   = { DUP4( -19195 ) };       // cos * (1<<16) + 0.5f

// Table for rows 0,4 - constants are multiplied on cos_4_16
__ALIGN8 static short tab_i_04[] = {
    16384, 21407, 16384, 8867,      // w05 w04 w01 w00
    16384, 8867, -16384, -21407,    // w07 w06 w03 w02
    16384, -8867, 16384, -21407,    // w13 w12 w09 w08
    -16384, 21407, 16384, -8867,    // w15 w14 w11 w10
    22725, 19266, 19266, -4520,     // w21 w20 w17 w16
    12873, 4520, -22725, -12873,    // w23 w22 w19 w18
    12873, -22725, 4520, -12873,    // w29 w28 w25 w24
    4520, 19266, 19266, -22725      // w31 w30 w27 w26
};

// Table for rows 1,7 - constants are multiplied on cos_1_16
__ALIGN8 static short tab_i_17[] = {
    22725, 29692, 22725, 12299,     // w05 w04 w01 w00
    22725, 12299, -22725, -29692,   // w07 w06 w03 w02
    22725, -12299, 22725, -29692,   // w13 w12 w09 w08
    -22725, 29692, 22725, -12299,   // w15 w14 w11 w10
    31521, 26722, 26722, -6270,     // w21 w20 w17 w16
    17855, 6270, -31521, -17855,    // w23 w22 w19 w18
    17855, -31521, 6270, -17855,    // w29 w28 w25 w24
    6270, 26722, 26722, -31521      // w31 w30 w27 w26
};
```

```
// Table for rows 2,6 - constants are multiplied on cos_2_16
__ALIGN8 static short tab_i_26[] = {
   21407, 27969, 21407, 11585,     // w05 w04 w01 w00
   21407, 11585, -21407, -27969,   // w07 w06 w03 w02
   21407, -11585, 21407, -27969,   // w13 w12 w09 w08
   -21407, 27969, 21407, -11585,   // w15 w14 w11 w10
   29692, 25172, 25172, -5906,     // w21 w20 w17 w16
   16819, 5906, -29692, -16819,    // w23 w22 w19 w18
   16819, -29692, 5906, -16819,    // w29 w28 w25 w24
   5906, 25172, 25172, -29692      // w31 w30 w27 w26
};

// Table for rows 3,5 - constants are multiplied on cos_3_16
__ALIGN8 static short tab_i_35[] = {
   19266, 25172, 19266, 10426,     // w05 w04 w01 w00
   19266, 10426, -19266, -25172,   // w07 w06 w03 w02
   19266, -10426, 19266, -25172,   // w13 w12 w09 w08
   -19266, 25172, 19266, -10426,   // w15 w14 w11 w10
   26722, 22654, 22654, -5315,     // w21 w20 w17 w16
   15137, 5315, -26722, -15137,    // w23 w22 w19 w18
   15137, -26722, 5315, -15137,    // w29 w28 w25 w24
   5315, 22654, 22654, -26722      // w31 w30 w27 w26
};

__ALIGN8 static const unsigned int rounder_0[2] = { DUP2( RND_INV_ROW - BIAS_SCALE( 2048 ) + 65536 ) };
__ALIGN8 static const unsigned int rounder_1[2] = { DUP2( RND_INV_ROW + BIAS_SCALE( 3755 ) ) };
__ALIGN8 static const unsigned int rounder_2[2] = { DUP2( RND_INV_ROW + BIAS_SCALE( 2472 ) ) };
__ALIGN8 static const unsigned int rounder_3[2] = { DUP2( RND_INV_ROW + BIAS_SCALE( 1361 ) ) };
__ALIGN8 static const unsigned int rounder_4[2] = { DUP2( RND_INV_ROW + BIAS_SCALE(    0 ) ) };
__ALIGN8 static const unsigned int rounder_5[2] = { DUP2( RND_INV_ROW - BIAS_SCALE( 1139 ) ) };
__ALIGN8 static const unsigned int rounder_6[2] = { DUP2( RND_INV_ROW - BIAS_SCALE( 1024 ) ) };
__ALIGN8 static const unsigned int rounder_7[2] = { DUP2( RND_INV_ROW - BIAS_SCALE( 1301 ) ) };

#define DCT_8_INV_ROW( table, rounder ) \
   __asm movq      mm2, mm0                    /* 2: x3 x2 x1 x0*/ \
   __asm movq      mm3, qword ptr [table+ 0]   /* 3: w05 w04 w01 w00*/ \
   __asm pshufw    mm0, mm0, 10001000b         /* x2 x0 x2 x0*/ \
   __asm movq      mm4, qword ptr [table+ 8]   /* 4: w07 w06 w03 w02*/ \
   __asm movq      mm5, mm1                    /* 5: x7 x6 x5 x4*/ \
   __asm pmaddwd   mm3, mm0                    /* x2*w05+x0*w04 x2*w01+x0*w00*/ \
   __asm movq      mm6, qword ptr [table+32]   /* 6: w21 w20 w17 w16*/ \
   __asm pshufw    mm1, mm1, 10001000b         /* x6 x4 x6 x4*/ \
   __asm pmaddwd   mm4, mm1                    /* x6*w07+x4*w06 x6*w03+x4*w02*/ \
   __asm movq      mm7, qword ptr [table+40]   /* 7: w23 w22 w19 w18*/ \
   __asm pshufw    mm2, mm2, 11011101b         /* x3 x1 x3 x1*/ \
   __asm pmaddwd   mm6, mm2                    /* x3*w21+x1*w20 x3*w17+x1*w16*/ \
   __asm pshufw    mm5, mm5, 11011101b         /* x7 x5 x7 x5*/ \
   __asm pmaddwd   mm7, mm5                    /* x7*w23+x5*w22 x7*w19+x5*w18*/ \
   __asm paddd     mm3, qword ptr rounder      /* +rounder */ \
   __asm pmaddwd   mm0, qword ptr [table+16]   /* x2*w13+x0*w12 x2*w09+x0*w08*/ \
   __asm paddd     mm3, mm4                    /* 4: a1=sum(even1) a0=sum(even0)*/ \
   __asm pmaddwd   mm1, qword ptr [table+24]   /* x6*w15+x4*w14 x6*w11+x4*w10*/ \
   __asm movq      mm4, mm3                    /* 4: a1 a0 */ \
   __asm pmaddwd   mm2, qword ptr [table+48]   /* x3*w29+x1*w28 x3*w25+x1*w24*/ \
   __asm paddd     mm6, mm7                    /* 7: b1=sum(odd1) b0=sum(odd0)*/ \
   __asm pmaddwd   mm5, qword ptr [table+56]   /* x7*w31+x5*w30 x7*w27+x5*w26*/ \
   __asm paddd     mm3, mm6                    /* a1+b1 a0+b0*/ \
   __asm paddd     mm0, qword ptr rounder      /* +rounder*/ \
   __asm psrad     mm3, SHIFT_INV_ROW          /* y1=a1+b1 y0=a0+b0*/ \
   __asm paddd     mm0, mm1                    /* 1: a3=sum(even3) a2=sum(even2)*/ \
   __asm psubd     mm4, mm6                    /* 6: a1-b1 a0-b0 */ \
```

```
    __asm movq      mm7, mm0                    /* 7: a3 a2 */ \
    __asm paddd     mm2, mm5                    /* 5: b3=sum(odd3) b2=sum(odd2)*/ \
    __asm paddd     mm0, mm2                    /* a3+b3 a2+b2*/ \
    __asm psrad     mm4, SHIFT_INV_ROW          /* y6=a1-b1 y7=a0-b0*/ \
    __asm psubd     mm7, mm2                    /* 2: a3-b3 a2-b2*/ \
    __asm psrad     mm0, SHIFT_INV_ROW          /* y3=a3+b3 y2=a2+b2*/ \
    __asm psrad     mm7, SHIFT_INV_ROW          /* y4=a3-b3 y5=a2-b2*/ \
    __asm packssdw  mm3, mm0                    /* 0: y3 y2 y1 y0*/ \
    __asm packssdw  mm7, mm4                    /* 4: y6 y7 y4 y5*/ \
    __asm pshufw    mm7, mm7, 10110001b         /* y7 y6 y5 y4 */

#define DCT_8_INV_COL_4 \
    __asm movq      mm1, qword ptr tg_3_16      /* 1: tg_3_16 */ \
    __asm movq      mm2, mm0                    /* 2: x5 */ \
    __asm movq      mm3, qword ptr [edx+3*16]   /* 3: x3 */ \
    __asm pmulhw    mm0, mm1                    /* x5*tg_3_16 */ \
    __asm movq      mm4, qword ptr [edx+7*16]   /* 4: x7 */ \
    __asm pmulhw    mm1, mm3                    /* x3*tg_3_16 */ \
    __asm movq      mm5, qword ptr tg_1_16      /* 5: tg_1_16 */ \
    __asm movq      mm6, mm4                    /* 6: x7 */ \
    __asm pmulhw    mm4, mm5                    /* x7*tg_1_16 */ \
    __asm paddsw    mm0, mm2                    /* x5*tg_3_16 */ \
    __asm pmulhw    mm5, [edx+1*16]             /* x1*tg_1_16 */ \
    __asm paddsw    mm1, mm3                    /* x3*tg_3_16 */ \
    __asm movq      mm7, qword ptr [edx+6*16]   /* 7: x6 */ \
    __asm paddsw    mm0, mm3                    /* 3: tm765 = x5*tg_3_16+x3 */ \
    __asm movq      mm3, qword ptr tg_2_16      /* 3: tg_2_16 */ \
    __asm psubsw    mm2, mm1                    /* 1: tm465 = x5-x3*tg_3_16 */ \
    __asm pmulhw    mm7, mm3                    /* x6*tg_2_16 */ \
    __asm movq      mm1, mm0                    /* 1: tm765 */ \
    __asm pmulhw    mm3, [edx+2*16]             /* x2*tg_2_16 */ \
    __asm psubsw    mm5, mm6                    /* 6: tp465 = x1*tg_1_16-x7 */ \
    __asm paddsw    mm4, [edx+1*16]             /* tp765 = x1+x7*tg_1_16 */ \
    __asm paddsw    mm0, mm4                    /* t7 = tp765 + tm765 */ \
    __asm psubsw    mm4, mm1                    /* 1: tp65 = tp765 - tm765 */ \
    __asm paddsw    mm7, [edx+2*16]             /* tm03 = x2+x6*tg_2_16 */ \
    __asm movq      mm6, mm5                    /* 6: tp465 */ \
    __asm psubsw    mm3, [edx+6*16]             /* tm12 = x2*tg_2_16-x6 */ \
    __asm psubsw    mm5, mm2                    /* tm65 = tp465 - tm465 */ \
    __asm paddsw    mm6, mm2                    /* 2: t4 = tp465 + tm465 */ \
    __asm movq      [edx+7*16], mm0             /* 0: save t7 in y7 (tmp) */ \
    __asm movq      mm1, mm4                    /* 1: tp65 */ \
    __asm movq      mm2, qword ptr cos_4_16     /* 2: cos_4_16 */ \
    __asm paddsw    mm4, mm5                    /* tp65 + tm65 */ \
    __asm movq      mm0, qword ptr cos_4_16     /* 0: cos_4_16 */ \
    __asm pmulhw    mm2, mm4                    /* (tp65 + tm65)*cos_4_16 */ \
    __asm movq      [edx+3*16], mm6             /* 6: save t4 in y3 (tmp) */ \
    __asm psubsw    mm1, mm5                    /* 5: tp65 - tm65 */ \
    __asm movq      mm6, [edx]                  /* 6: x0 */ \
    __asm pmulhw    mm0, mm1                    /* (tp65 - tm65)*cos_4_16 */ \
    __asm movq      mm5, [edx+4*16]             /* 5: x4 */ \
    __asm paddsw    mm4, mm2                    /* 2: t6 = (tp65 + tm65)*cos_4_16 */ \
    __asm paddsw    mm5, mm6                    /* tp03 = x0 + x4 */ \
    __asm psubsw    mm6, [edx+4*16]             /* tp12 = x0 - x4 */ \
    __asm paddsw    mm0, mm1                    /* 1: t5 = (tp65 - tm65)*cos_4_16 */ \
    __asm movq      mm2, mm5                    /* 2: tp03 */ \
    __asm paddsw    mm5, mm7                    /* t0 = tp03 + tm03 */ \
    __asm movq      mm1, mm6                    /* 1: tp12 */ \
    __asm psubsw    mm2, mm7                    /* 7: t3 = tp03 - tm03 */ \
    __asm movq      mm7, [edx+7*16]             /* t7 */ \
    __asm paddsw    mm6, mm3                    /* t1 = tp12 + tm12 */ \
    __asm paddsw    mm7, mm5                    /* t0 + t7 */ \
```

```
    __asm psraw     mm7, SHIFT_INV_COL      /* y0 = t0 + t7 */ \
    __asm psubsw    mm1, mm3                /* 3: t2 = tp12 - tm12 */ \
    __asm movq      mm3, mm6                /* 3: t1 */ \
    __asm paddsw    mm6, mm4                /* t1 + t6 */ \
    __asm movq      [edx], mm7              /* 7: save y0 */ \
    __asm psraw     mm6, SHIFT_INV_COL      /* y1 = t1 + t6 */ \
    __asm movq      mm7, mm1                /* 7: t2 */ \
    __asm paddsw    mm1, mm0                /* t2 + t5 */ \
    __asm movq      [edx+1*16], mm6         /* 6: save y1 */ \
    __asm psraw     mm1, SHIFT_INV_COL      /* y2 = t2 + t5 */ \
    __asm movq      mm6, [edx+3*16]         /* 6: t4 */ \
    __asm psubsw    mm7, mm0                /* 0: t2 - t5 */ \
    __asm paddsw    mm6, mm2                /* t3 + t4 */ \
    __asm psubsw    mm2, [edx+3*16]         /* t3 - t4 */ \
    __asm psraw     mm7, SHIFT_INV_COL      /* y5 = t2 - t5 */ \
    __asm movq      [edx+2*16], mm1         /* 1: save y2 */ \
    __asm psraw     mm6, SHIFT_INV_COL      /* y3 = t3 + t4 */ \
    __asm psubsw    mm5, [edx+7*16]         /* t0 - t7 */ \
    __asm psraw     mm2, SHIFT_INV_COL      /* y4 = t3 - t4 */ \
    __asm movq      [edx+3*16], mm6         /* 6: save y3 */ \
    __asm psubsw    mm3, mm4                /* 4: t1 - t6 */ \
    __asm movq      [edx+4*16], mm2         /* 2: save y4 */ \
    __asm psraw     mm3, SHIFT_INV_COL      /* y6 = t1 - t6 */ \
    __asm movq      [edx+5*16], mm7         /* 7: save y5 */ \
    __asm psraw     mm5, SHIFT_INV_COL      /* y7 = t0 - t7 */ \
    __asm movq      [edx+6*16], mm3         /* 3: save y6 */ \
    __asm movq      [edx+7*16], mm5         /* 5: save y7 */

#define DEQUANTIZE( reg, mem )      __asm pmullw reg, mem

void IDCT_MMX( const short *coeff, const unsigned short *quant, short *dest ) {
    __asm mov       ecx, coeff
    __asm mov       edx, dest
    __asm mov       esi, quant

    __asm movq      mm0, [ecx]
    __asm movq      mm1, [ecx+8]
    DEQUANTIZE(     mm0, [esi+ 0] )
    DEQUANTIZE(     mm1, [esi+ 8] )
    DCT_8_INV_ROW( tab_i_04, rounder_0 );           /* row 0 */
    __asm movq      mm0, [ecx+16]
    __asm movq      qword ptr [edx], mm3            /* 3: save y3 y2 y1 y0 */
    __asm movq      mm1, [ecx+24]
    __asm movq      qword ptr [edx+8], mm7          /* 7: save y7 y6 y5 y4 */
    DEQUANTIZE(     mm0, [esi+16] )
    DEQUANTIZE(     mm1, [esi+24] )
    DCT_8_INV_ROW( tab_i_17, rounder_1 );           /* row 1 */
    __asm movq      mm0, [ecx+32]
    __asm movq      qword ptr [edx+16], mm3         /* 3: save y3 y2 y1 y0 */
    __asm movq      mm1, [ecx+40]
    __asm movq      qword ptr [edx+24], mm7         /* 7: save y7 y6 y5 y4 */
    DEQUANTIZE(     mm0, [esi+32] )
    DEQUANTIZE(     mm1, [esi+40] )
    DCT_8_INV_ROW( tab_i_26, rounder_2 );           /* row 2 */
    __asm movq      mm0, [ecx+48]
    __asm movq      qword ptr [edx+32], mm3         /* 3: save y3 y2 y1 y0 */
    __asm movq      mm1, [ecx+56]
    __asm movq      qword ptr [edx+40], mm7         /* 7: save y7 y6 y5 y4 */
    DEQUANTIZE(     mm0, [esi+48] )
    DEQUANTIZE(     mm1, [esi+56] )
    DCT_8_INV_ROW( tab_i_35, rounder_3 );           /* row 3 */
    __asm movq      mm0, [ecx+64]
```

```
    __asm movq      qword ptr [edx+48], mm3         /* 3: save y3 y2 y1 y0 */
    __asm movq      mm1, [ecx+72]
    __asm movq      qword ptr [edx+56], mm7         /* 7: save y7 y6 y5 y4 */
    DEQUANTIZE(     mm0, [esi+64] )
    DEQUANTIZE(     mm1, [esi+72] )
    DCT_8_INV_ROW( tab_i_04, rounder_4 );           /* row 4 */
    __asm movq      mm0, [ecx+80]
    __asm movq      qword ptr [edx+64], mm3         /* 3: save y3 y2 y1 y0 */
    __asm movq      mm1, [ecx+88]
    __asm movq      qword ptr [edx+72], mm7         /* 7: save y7 y6 y5 y4 */
    DEQUANTIZE(     mm0, [esi+80] )
    DEQUANTIZE(     mm1, [esi+88] )
    DCT_8_INV_ROW( tab_i_35, rounder_5 );           /* row 5 */
    __asm movq      mm0, [ecx+96]
    __asm movq      qword ptr [edx+80], mm3         /* 3: save y3 y2 y1 y0 */
    __asm movq      mm1, [ecx+104]
    __asm movq      qword ptr [edx+88], mm7         /* 7: save y7 y6 y5 y4 */
    DEQUANTIZE(     mm0, [esi+96] )
    DEQUANTIZE(     mm1, [esi+104] )
    DCT_8_INV_ROW( tab_i_26, rounder_6 );           /* row 6 */
    __asm movq      mm0, [ecx+112]
    __asm movq      qword ptr [edx+96], mm3         /* 3: save y3 y2 y1 y0 */
    __asm movq      mm1, [ecx+120]
    __asm movq      qword ptr [edx+104],mm7         /* 7: save y7 y6 y5 y4 */
    DEQUANTIZE(     mm0, [esi+112] )
    DEQUANTIZE(     mm1, [esi+120] )
    DCT_8_INV_ROW( tab_i_17, rounder_7 );           /* row 7 */
    __asm movq      qword ptr [edx+112],mm3         /* 3: save y3 y2 y1 y0 */
    __asm movq      mm0, qword ptr [edx+80]         /* 0: x5 */
    __asm movq      qword ptr [edx+120],mm7         /* 7: save y7 y6 y5 y4 */

    DCT_8_INV_COL_4
    __asm movq      mm0, qword ptr [edx+88]         /* 0: x5 */
    __asm add       edx, 8
    DCT_8_INV_COL_4
    __asm emms
}
```

# Appendix E

```
/*
   SSE2 Optimized Integer Inverse Discrete Cosine Transform
   Copyright (C) 2006 Id Software, Inc.
   Original AP922 algorithm is Copyright (C) 1999 - 2000 Intel Corporation.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

#define __ALIGN16                   __declspec(align(16))

#define BITS_INV_ACC                5                       // 4 or 5 for IEEE
#define SHIFT_INV_ROW               16 - BITS_INV_ACC
#define SHIFT_INV_COL               1 + BITS_INV_ACC

#define RND_INV_ROW                 1024 * (6 - BITS_INV_ACC)   // 1 << (SHIFT_INV_ROW-1)
#define RND_INV_COL                 16 * (BITS_INV_ACC - 3)     // 1 << (SHIFT_INV_COL-1)

#define DUP4( X )                   (X),(X),(X),(X)
#define DUP8( X )                   (X),(X),(X),(X),(X),(X),(X),(X)
#define BIAS_SCALE( X )             ( X / ( BITS_INV_ACC - 3 ) )

__ALIGN16 static short M128_tg_1_16[8]  = { DUP8(  13036 ) };       // tg * (1<<16) + 0.5
__ALIGN16 static short M128_tg_2_16[8]  = { DUP8(  27146 ) };       // tg * (1<<16) + 0.5
__ALIGN16 static short M128_tg_3_16[8]  = { DUP8( -21746 ) };       // tg * (1<<16) + 0.5
__ALIGN16 static short M128_cos_4_16[8] = { DUP8( -19195 ) };       // cos * (1<<16) + 0.5

//-----------------------------------------------------------------------------

// Table for rows 0,4 - constants are multiplied on cos_4_16
__ALIGN16 static short M128_tab_i_04[] = {
   16384,  21407,  16384,  8867,       // w05 w04 w01 w00
   16384,  -8867,  16384,  -21407,     // w13 w12 w09 w08
   16384,  8867,   -16384, -21407,     // w07 w06 w03 w02
   -16384, 21407,  16384,  -8867,      // w15 w14 w11 w10
   22725,  19266,  19266,  -4520,      // w21 w20 w17 w16
   12873,  -22725, 4520,   -12873,     // w29 w28 w25 w24
   12873,  4520,   -22725, -12873,     // w23 w22 w19 w18
   4520,   19266,  19266,  -22725      // w31 w30 w27 w26
};

// Table for rows 1,7 - constants are multiplied on cos_1_16
__ALIGN16 static short M128_tab_i_17[] = {
   22725,  29692,  22725,  12299,      // w05 w04 w01 w00
   22725,  -12299, 22725,  -29692,     // w13 w12 w09 w08
   22725,  12299,  -22725, -29692,     // w07 w06 w03 w02
   -22725, 29692,  22725,  -12299,     // w15 w14 w11 w10
   31521,  26722,  26722,  -6270,      // w21 w20 w17 w16
   17855,  -31521, 6270,   -17855,     // w29 w28 w25 w24
```

```
   17855,  6270,   -31521, -17855,     // w23 w22 w19 w18
   6270,   26722,  26722,  -31521      // w31 w30 w27 w26
};

// Table for rows 2,6 - constants are multiplied on cos_2_16
__ALIGN16 static short M128_tab_i_26[] = {
   21407,  27969,  21407,  11585,      // w05 w04 w01 w00
   21407,  -11585, 21407,  -27969,     // w13 w12 w09 w08
   21407,  11585,  -21407, -27969,     // w07 w06 w03 w02
   -21407, 27969,  21407,  -11585,     // w15 w14 w11 w10
   29692,  25172,  25172,  -5906,      // w21 w20 w17 w16
   16819,  -29692, 5906,   -16819,     // w29 w28 w25 w24
   16819,  5906,   -29692, -16819,     // w23 w22 w19 w18
   5906,   25172,  25172,  -29692      // w31 w30 w27 w26
};

// Table for rows 3,5 - constants are multiplied on cos_3_16
__ALIGN16 static short M128_tab_i_35[] = {
   19266,  25172,  19266,  10426,      // w05 w04 w01 w00
   19266,  -10426, 19266,  -25172,     // w13 w12 w09 w08
   19266,  10426,  -19266, -25172,     // w07 w06 w03 w02
   -19266, 25172,  19266,  -10426,     // w15 w14 w11 w10
   26722,  22654,  22654,  -5315,      // w21 w20 w17 w16
   15137,  -26722, 5315,   -15137,     // w29 w28 w25 w24
   15137,  5315,   -26722, -15137,     // w23 w22 w19 w18
   5315,   22654,  22654,  -26722      // w31 w30 w27 w26
};

__ALIGN16 static const unsigned int rounder_0[4] = { DUP4( RND_INV_ROW - BIAS_SCALE( 2048 ) + 65536 ) };
__ALIGN16 static const unsigned int rounder_1[4] = { DUP4( RND_INV_ROW + BIAS_SCALE( 3755 ) ) };
__ALIGN16 static const unsigned int rounder_2[4] = { DUP4( RND_INV_ROW + BIAS_SCALE( 2472 ) ) };
__ALIGN16 static const unsigned int rounder_3[4] = { DUP4( RND_INV_ROW + BIAS_SCALE( 1361 ) ) };
__ALIGN16 static const unsigned int rounder_4[4] = { DUP4( RND_INV_ROW + BIAS_SCALE(    0 ) ) };
__ALIGN16 static const unsigned int rounder_5[4] = { DUP4( RND_INV_ROW - BIAS_SCALE( 1139 ) ) };
__ALIGN16 static const unsigned int rounder_6[4] = { DUP4( RND_INV_ROW - BIAS_SCALE( 1024 ) ) };
__ALIGN16 static const unsigned int rounder_7[4] = { DUP4( RND_INV_ROW - BIAS_SCALE( 1301 ) ) };

#define DCT_8_INV_ROW( table1, table2, rounder1, rounder2 )           \
   __asm   pshuflw     xmm0, xmm0, 0xD8                              \
   __asm   pshufhw     xmm0, xmm0, 0xD8                              \
   __asm   pshufd      xmm3, xmm0, 0x55                             \
   __asm   pshufd      xmm1, xmm0, 0                               \
   __asm   pshufd      xmm2, xmm0, 0xAA                              \
   __asm   pshufd      xmm0, xmm0, 0xFF                             \
   __asm   pmaddwd     xmm1, [table1+ 0]                             \
   __asm   pmaddwd     xmm2, [table1+16]                             \
   __asm   pmaddwd     xmm3, [table1+32]                             \
   __asm   pmaddwd     xmm0, [table1+48]                             \
   __asm   paddd       xmm0, xmm3                                   \
   __asm   pshuflw     xmm4, xmm4, 0xD8                              \
   __asm   pshufhw     xmm4, xmm4, 0xD8                              \
   __asm   paddd       xmm1, rounder1                               \
   __asm   pshufd      xmm6, xmm4, 0xAA                              \
   __asm   pshufd      xmm5, xmm4, 0                               \
   __asm   pmaddwd     xmm5, [table2+ 0]                            \
   __asm   paddd       xmm5, rounder2                               \
   __asm   pmaddwd     xmm6, [table2+16]                             \
   __asm   pshufd      xmm7, xmm4, 0x55                             \
   __asm   pmaddwd     xmm7, [table2+32]                             \
   __asm   pshufd      xmm4, xmm4, 0xFF                             \
   __asm   pmaddwd     xmm4, [table2+48]                             \
   __asm   paddd       xmm1, xmm2                                   \
```

```
    __asm   movdqa      xmm2, xmm1                          \
    __asm   psubd       xmm2, xmm0                        \
    __asm   psrad       xmm2, SHIFT_INV_ROW                     \
    __asm   pshufd      xmm2, xmm2, 0x1B                    \
    __asm   paddd       xmm0, xmm1                        \
    __asm   psrad       xmm0, SHIFT_INV_ROW                     \
    __asm   paddd       xmm5, xmm6                        \
    __asm   packssdw    xmm0, xmm2                          \
    __asm   paddd       xmm4, xmm7                        \
    __asm   movdqa      xmm6, xmm5                          \
    __asm   psubd       xmm6, xmm4                        \
    __asm   psrad       xmm6, SHIFT_INV_ROW                     \
    __asm   paddd       xmm4, xmm5                        \
    __asm   psrad       xmm4, SHIFT_INV_ROW                     \
    __asm   pshufd      xmm6, xmm6, 0x1B                    \
    __asm   packssdw    xmm4, xmm6

#define DCT_8_INV_COL_8                                     \
    __asm   movdqa      xmm6, xmm4                          \
    __asm   movdqa      xmm2, xmm0                          \
    __asm   movdqa      xmm3, XMMWORD PTR [edx+3*16]              \
    __asm   movdqa      xmm1, XMMWORD PTR M128_tg_3_16              \
    __asm   pmulhw      xmm0, xmm1                          \
    __asm   movdqa      xmm5, XMMWORD PTR M128_tg_1_16              \
    __asm   pmulhw      xmm1, xmm3                          \
    __asm   paddsw      xmm1, xmm3                        \
    __asm   pmulhw      xmm4, xmm5                          \
    __asm   movdqa      xmm7, XMMWORD PTR [edx+6*16]              \
    __asm   pmulhw      xmm5, [edx+1*16]                    \
    __asm   psubsw      xmm5, xmm6                        \
    __asm   movdqa      xmm6, xmm5                          \
    __asm   paddsw      xmm4, [edx+1*16]                    \
    __asm   paddsw      xmm0, xmm2                        \
    __asm   paddsw      xmm0, xmm3                        \
    __asm   psubsw      xmm2, xmm1                        \
    __asm   movdqa      xmm1, xmm0                          \
    __asm   movdqa      xmm3, XMMWORD PTR M128_tg_2_16              \
    __asm   pmulhw      xmm7, xmm3                          \
    __asm   pmulhw      xmm3, [edx+2*16]                    \
    __asm   paddsw      xmm0, xmm4                        \
    __asm   psubsw      xmm4, xmm1                        \
    __asm   movdqa      [edx+7*16],  xmm0                   \
    __asm   psubsw      xmm5, xmm2                        \
    __asm   paddsw      xmm6, xmm2                        \
    __asm   movdqa      [edx+3*16],  xmm6                   \
    __asm   movdqa      xmm1, xmm4                          \
    __asm   movdqa      xmm0, XMMWORD PTR M128_cos_4_16             \
    __asm   movdqa      xmm2, xmm0                          \
    __asm   paddsw      xmm4, xmm5                        \
    __asm   psubsw      xmm1, xmm5                        \
    __asm   paddsw      xmm7, [edx+2*16]                    \
    __asm   psubsw      xmm3, [edx+6*16]                    \
    __asm   movdqa      xmm6, [edx]                       \
    __asm   pmulhw      xmm0, xmm1                          \
    __asm   movdqa      xmm5, [edx+4*16]                    \
    __asm   paddsw      xmm5, xmm6                        \
    __asm   psubsw      xmm6, [edx+4*16]                    \
    __asm   pmulhw      xmm2, xmm4                          \
    __asm   paddsw      xmm4, xmm2                        \
    __asm   movdqa      xmm2, xmm5                          \
    __asm   psubsw      xmm2, xmm7                        \
    __asm   paddsw      xmm0, xmm1                        \
```

```
    __asm   paddsw      xmm5, xmm7                          \
    __asm   movdqa      xmm1, xmm6                          \
    __asm   movdqa      xmm7, [edx+7*16]                    \
    __asm   paddsw      xmm7, xmm5                          \
    __asm   psraw       xmm7, SHIFT_INV_COL                 \
    __asm   movdqa      [edx], xmm7                         \
    __asm   paddsw      xmm6, xmm3                          \
    __asm   psubsw      xmm1, xmm3                          \
    __asm   movdqa      xmm7, xmm1                          \
    __asm   movdqa      xmm3, xmm6                          \
    __asm   paddsw      xmm6, xmm4                          \
    __asm   psraw       xmm6, SHIFT_INV_COL                 \
    __asm   movdqa      [edx+1*16], xmm6                    \
    __asm   paddsw      xmm1, xmm0                          \
    __asm   psraw       xmm1, SHIFT_INV_COL                 \
    __asm   movdqa      [edx+2*16],  xmm1                   \
    __asm   movdqa      xmm1, [edx+3*16]                    \
    __asm   movdqa      xmm6, xmm1                          \
    __asm   psubsw      xmm7, xmm0                          \
    __asm   psraw       xmm7, SHIFT_INV_COL                 \
    __asm   movdqa      [edx+5*16],  xmm7                   \
    __asm   psubsw      xmm5, [edx+7*16]                    \
    __asm   psraw       xmm5, SHIFT_INV_COL                 \
    __asm   movdqa      [edx+7*16], xmm5                    \
    __asm   psubsw      xmm3, xmm4                          \
    __asm   paddsw      xmm6, xmm2                          \
    __asm   psubsw      xmm2, xmm1                          \
    __asm   psraw       xmm6, SHIFT_INV_COL                 \
    __asm   movdqa      [edx+3*16],  xmm6                   \
    __asm   psraw       xmm2, SHIFT_INV_COL                 \
    __asm   movdqa      [edx+4*16],  xmm2                   \
    __asm   psraw       xmm3, SHIFT_INV_COL                 \
    __asm   movdqa      [edx+6*16],  xmm3

#define DEQUANTIZE( reg, mem )      __asm pmullw reg, mem

void IDCT_SSE2( const short *coeff, const unsigned short *quant, short *dest ) {
    assert_16_byte_aligned( coeff );
    assert_16_byte_aligned( quant );
    assert_16_byte_aligned( dest );

    __asm mov           eax, coeff
    __asm mov           edx, dest
    __asm mov           esi, quant

    __asm movdqa        xmm0, XMMWORD PTR[eax+16*0]         // row 0
    __asm movdqa        xmm4, XMMWORD PTR[eax+16*2]         // row 2
    DEQUANTIZE(         xmm0, XMMWORD PTR[esi+16*0] )
    DEQUANTIZE(         xmm4, XMMWORD PTR[esi+16*2] )
    DCT_8_INV_ROW( M128_tab_i_04, M128_tab_i_26, rounder_0, rounder_2 );
    __asm movdqa        XMMWORD PTR[edx+16*0],  xmm0
    __asm movdqa        XMMWORD PTR[edx+16*2],  xmm4

    __asm movdqa        xmm0, XMMWORD PTR[eax+16*4]         // row 4
    __asm movdqa        xmm4, XMMWORD PTR[eax+16*6]         // row 6
    DEQUANTIZE(         xmm0, XMMWORD PTR[esi+16*4] )
    DEQUANTIZE(         xmm4, XMMWORD PTR[esi+16*6] )
    DCT_8_INV_ROW( M128_tab_i_04, M128_tab_i_26, rounder_4, rounder_6 );
    __asm movdqa        XMMWORD PTR[edx+16*4],  xmm0
    __asm movdqa        XMMWORD PTR[edx+16*6],  xmm4

    __asm movdqa        xmm0, XMMWORD PTR[eax+16*3]         // row 3
```

```
    __asm movdqa        xmm4, XMMWORD PTR[eax+16*1]         // row 1
    DEQUANTIZE(         xmm0, XMMWORD PTR[esi+16*3] )
    DEQUANTIZE(         xmm4, XMMWORD PTR[esi+16*1] )
    DCT_8_INV_ROW( M128_tab_i_35, M128_tab_i_17, rounder_3, rounder_1 );
    __asm movdqa        XMMWORD PTR[edx+16*3],  xmm0
    __asm movdqa        XMMWORD PTR[edx+16*1],  xmm4

    __asm movdqa        xmm0, XMMWORD PTR[eax+16*5]         // row 5
    __asm movdqa        xmm4, XMMWORD PTR[eax+16*7]         // row 7
    DEQUANTIZE(         xmm0, XMMWORD PTR[esi+16*5] )
    DEQUANTIZE(         xmm4, XMMWORD PTR[esi+16*7] )
    DCT_8_INV_ROW( M128_tab_i_35, M128_tab_i_17, rounder_5, rounder_7 );

    DCT_8_INV_COL_8
}
```

# Appendix F

```
/*
   4:2:0 YCoCg -> RGB Conversion
   Copyright (C) 2006 Id Software, Inc.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

#define NUM_CHANNELS            4

inline ClampByte( int x )       { return ( x < 0 ) ? 0 : ( ( x > 255 ) ? 255 : x ); }

void YCoCgAToRGBA( const short *YCoCgA, byte *rgba, int stride ) {
   int i, j, k;

   // writes out one 8*8 block of the 16*16 tile per iteration
   for( k = 0; k < 4; k++ ) {

      byte *pByte    = rgba + ( (k & 2) * stride + (k & 1) * (NUM_CHANNELS*2) ) * 4;
      const short *py = YCoCgA + k * 64;
      const short *pc = YCoCgA + 256 + ( (k & 2) * 4 + (k & 1) ) * 4;

      // writes out 2 rows of an 8*8 block per iteration
      for( j = 0; j < 4; j++ ) {
         for( i = 0; i < 4; i++ ) {
            int y, co, cg, r, s, t, a;

            co = pc[i+ 0];
            cg = pc[i+64];

            r = co - cg;
            s = cg;
            t = co + cg;

            y = py[i*2+0+0+  0] + 128;
            a = py[i*2+0+0+384] + 128;

            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+0] = ClampByte( y+r );    // Red
            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+1] = ClampByte( y+s );    // Green
            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+2] = ClampByte( y-t );    // Blue
            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+3] = ClampByte( a );      // Alpha

            y = py[i*2+0+1+  0] + 128;
            a = py[i*2+0+1+384] + 128;

            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+0] = ClampByte( y+r );    // Red
            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+1] = ClampByte( y+s );    // Green
            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+2] = ClampByte( y-t );    // Blue
            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+3] = ClampByte( a );      // Alpha
```

```
            pByte += stride;

            y = py[i*2+8+0+  0] + 128;
            a = py[i*2+8+0+384] + 128;

            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+0] = ClampByte( y+r );    // Red
            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+1] = ClampByte( y+s );    // Green
            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+2] = ClampByte( y-t );    // Blue
            pByte[i*2*NUM_CHANNELS+0*NUM_CHANNELS+3] = ClampByte( a );      // Alpha

            y = py[i*2+8+1+  0] + 128;
            a = py[i*2+8+1+384] + 128;

            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+0] = ClampByte( y+r );    // Red
            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+1] = ClampByte( y+s );    // Green
            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+2] = ClampByte( y-t );    // Blue
            pByte[i*2*NUM_CHANNELS+1*NUM_CHANNELS+3] = ClampByte( a );      // Alpha

            pByte -= stride;
          }

          py += 16;
          pc += 8;
          pByte += 2 * stride;
        }
    }
}
```

# Appendix G

```
/*
   MMX Optimized 4:2:0 YCoCg -> RGB Conversion
   Copyright (C) 2006 Id Software, Inc.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

void YCoCgAToRGBA_MMX( const short *YCoCgA, byte *rgba, int stride ) {
   assert( NUM_CHANNELS == 4 );                    // this code assumes four channels per pixel
   assert_16_byte_aligned( rgba );
   assert_16_byte_aligned( YCoCgA );
   assert( ( stride & 15 ) == 0 );

   ALIGN16( short tmm2[4] );
   ALIGN16( short tmm3[4] );
   ALIGN16( short tmm4[4] );
   ALIGN16( short tmm5[4] );
   ALIGN16( short tmm6[4] );
   ALIGN16( short tmm7[4] );

   __asm {
      xor       ecx, ecx                      // ecx = k
      mov       esi, YCoCgA

   // iterates 4 times, writes out one 8*8 block of the 16*16 tile per iteration
   loop1:
      mov       eax, ecx
      and       eax, 2                        // ( k & 2 )
      mov       edx, ecx
      and       edx, 1                        // ( k & 1 )

      lea       edi, [edx+eax*4]
      shl       edi, 3
      add       edi, dword ptr [YCoCgA]
      add       edi, 256*2                    // YCoCgA + ( 256 + ( (k&2) * 4 + (k&1) ) * 4 ) * sizeof( YCoCgA[0] )

      imul      eax, stride
      lea       edx, [eax+edx*(NUM_CHANNELS*2)]
      shl       edx, 2
      add       edx, dword ptr [rgba]         // rgba + ( (k&2) * stride + (k&1) * (NUM_CHANNELS*2) ) * 4

      mov       eax, -4*16*2
      add       esi, 4*16*2

   // iterates 4 times, writes out 2 rows of an 8*8 block per iteration
   loop2:
      movq      mm4, [edi+ 0*2]               // mm4 = co
      movq      mm2, [edi+64*2]               // mm2 = g = cg
```

```
movq       mm3, mm4                        // mm3 = co
psubsw     mm3, mm2                         // mm3 = r = co - cg
paddsw     mm4, mm2                         // mm4 = b = co + cg

pshufw     mm7, mm3, R_SHUFFLE_D( 2, 2, 3, 3 ) // mm7 = r
pshufw     mm6, mm2, R_SHUFFLE_D( 2, 2, 3, 3 ) // mm6 = g
pshufw     mm5, mm4, R_SHUFFLE_D( 2, 2, 3, 3 ) // mm5 = b

movq       tmm7, mm7
movq       tmm6, mm6
movq       tmm5, mm5

pshufw     mm3, mm3, R_SHUFFLE_D( 0, 0, 1, 1 )
pshufw     mm2, mm2, R_SHUFFLE_D( 0, 0, 1, 1 )
pshufw     mm4, mm4, R_SHUFFLE_D( 0, 0, 1, 1 )

movq       tmm3, mm3
movq       tmm2, mm2
movq       tmm4, mm4

movq       mm1, [esi+eax+0*2]
paddsw     mm1, SIMD_MMX_word_128

paddsw     mm3, mm1                        // r0, r1, r2, r3      ( y + r )
paddsw     mm2, mm1                        // g0, g1, g2, g3      ( y + g )
psubsw     mm1, mm4                        // b0, b1, b2, b3      ( y - b )

packuswb   mm3, mm3                         // r0, r1, r2, r3, r0, r1, r2, r3
packuswb   mm2, mm2                         // g0, g1, g2, g3, g0, g1, g2, g3
packuswb   mm1, mm1                         // b0, b1, b2, b3, b0, b1, b2, b3

movq       mm0, [esi+eax+0*2+384*2]
paddsw     mm0, SIMD_MMX_word_128
packuswb   mm0, mm0

punpcklbw  mm1, mm0                          // b0, a0, b1, a1, b2, a2, b3, a3
punpcklbw  mm3, mm2                          // r0, g0, r1, g1, r2, g2, r3, g3
movq       mm4, mm3                         // r0, g0, r1, g1, r2, g2, r3, g3
punpcklwd  mm3, mm1                          // r0, g0, b0, a0, r1, g1, b1, a1
punpckhwd  mm4, mm1                          // r2, g2, b2, a2, r3, g3, b3, a3

movq       [edx+0], mm3
movq       [edx+8], mm4

movq       mm2, [esi+eax+4*2]
paddsw     mm2, SIMD_MMX_word_128

paddsw     mm7, mm2                        // r0, r1, r2, r3      ( y + r )
paddsw     mm6, mm2                        // g0, g1, g2, g3      ( y + b )
psubsw     mm2, mm5                        // b0, b1, b2, b3      ( y - g )

packuswb   mm7, mm7                         // r0, r1, r2, r3, r0, r1, r2, r3
packuswb   mm6, mm6                         // g0, g1, g2, g3, g0, g1, g2, g3
packuswb   mm2, mm2                         // b0, b1, b2, b3, b0, b1, b2, b3

movq       mm0, [esi+eax+4*2+384*2]
paddsw     mm0, SIMD_MMX_word_128
packuswb   mm0, mm0

punpcklbw  mm2, mm0                          // b0, a0, b1, a1, b2, a2, b3, a3
punpcklbw  mm7, mm6                          // r0, g0, r1, g1, r2, g2, r3, g3
```

```
movq       mm5, mm7                        // r0, g0, r1, g1, r2, g2, r3, g3
punpcklwd   mm7, mm2                        // r0, g0, b0, a0, r1, g1, b1, a1
punpckhwd   mm5, mm2                        // r2, g2, b2, a2, r3, g3, b3, a3

movq       [edx+16], mm7
movq       [edx+24], mm5
add        edx, stride

movq       mm7, tmm7
movq       mm6, tmm6
movq       mm5, tmm5

movq       mm3, tmm3
movq       mm2, tmm2
movq       mm4, tmm4

movq       mm1, [esi+eax+8*2]
paddsw     mm1, SIMD_MMX_word_128

paddsw     mm3, mm1                        // r0, r1, r2, r3      ( y + r )
paddsw     mm2, mm1                        // g0, g1, g2, g3      ( y + g )
psubsw     mm1, mm4                        // b0, b1, b2, b3      ( y - b )

packuswb    mm3, mm3                       // r0, r1, r2, r3, r0, r1, r2, r3
packuswb    mm2, mm2                       // g0, g1, g2, g3, g0, g1, g2, g3
packuswb    mm1, mm1                       // b0, b1, b2, b3, b0, b1, b2, b3

movq       mm0, [esi+eax+8*2+384*2]
paddsw     mm0, SIMD_MMX_word_128
packuswb    mm0, mm0

punpcklbw   mm1, mm0                       // b0, a0, b1, a1, b2, a2, b3, a3
punpcklbw   mm3, mm2                       // r0, g0, r1, g1, r2, g2, r3, g3
movq       mm4, mm3                        // r0, g0, r1, g1, r2, g2, r3, g3
punpcklwd   mm3, mm1                       // r0, g0, b0, a0, r1, g1, b1, a1
punpckhwd   mm4, mm1                       // r2, g2, b2, a2, r3, g3, b3, a3

movq       [edx+0], mm3
movq       [edx+8], mm4

movq       mm2, [esi+eax+12*2]
paddsw     mm2, SIMD_MMX_word_128

paddsw     mm7, mm2                        // r0, r1, r2, r3      ( y + r )
paddsw     mm6, mm2                        // g0, g1, g2, g3      ( y + b )
psubsw     mm2, mm5                        // b0, b1, b2, b3      ( y - g )

packuswb    mm7, mm7                       // r0, r1, r2, r3, r0, r1, r2, r3
packuswb    mm6, mm6                       // g0, g1, g2, g3, g0, g1, g2, g3
packuswb    mm2, mm2                       // b0, b1, b2, b3, b0, b1, b2, b3

movq       mm0, [esi+eax+12*2+384*2]
paddsw     mm0, SIMD_MMX_word_128
packuswb    mm0, mm0

punpcklbw   mm2, mm0                       // b0, a0, b1, a1, b2, a2, b3, a3
punpcklbw   mm7, mm6                       // r0, g0, r1, g1, r2, g2, r3, g3
movq       mm5, mm7                        // r0, g0, r1, g1, r2, g2, r3, g3
punpcklwd   mm7, mm2                       // r0, g0, b0, a0, r1, g1, b1, a1
punpckhwd   mm5, mm2                       // r2, g2, b2, a2, r3, g3, b3, a3

movq       [edx+16], mm7
```

```
        movq        [edx+24], mm5
        add         edx, stride

        add         edi, 8*2
        add         eax, 16*2
        jl          loop2

        add         ecx, 1
        cmp         ecx, 4
        jl          loop1

        emms
    }
}
```

# Appendix H

```
/*
   SSE2 Optimized 4:2:0 YCoCg -> RGB Conversion
   Copyright (C) 2006 Id Software, Inc.

   This code is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This code is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.
*/

void YCoCgAToRGBA_SSE2( const short *YCoCgA, byte *rgba, int stride ) {
   assert( NUM_CHANNELS == 4 );                    // this code assumes four channels per pixel
   assert_16_byte_aligned( rgba );
   assert_16_byte_aligned( YCoCgA );
   assert( ( stride & 15 ) == 0 );

   __asm {
      xor       ecx, ecx                      // ecx = k
      mov       esi, YCoCgA

   // iterates 4 times, writes out one 8*8 block of the 16*16 tile per iteration
   loop1:
      mov       eax, ecx
      and       eax, 2                        // (k&2)
      mov       edx, ecx
      and       edx, 1                        // (k&1)

      lea       edi, [edx+eax*4]
      shl       edi, 3
      add       edi, dword ptr [YCoCgA]
      add       edi, 256*2                    // YCoCgA + ( 256 + ( (k&2) * 4 + (k&1) ) * 4 ) * sizeof( YCoCgA[0] )

      imul      eax, stride
      lea       edx, [eax+edx*(NUM_CHANNELS*2)]
      shl       edx, 2
      add       edx, dword ptr [rgba]         // rgba + ( (k&2) * stride + (k&1) * (NUM_CHANNELS*2) ) * 4

      mov       eax, -4*16*2
      add       esi, 4*16*2

   // iterates 4 times, writes out 2 rows of an 8*8 block per iteration
   loop2:
      movq      xmm4, qword ptr [edi+ 0*2]    // xmm4 = co
      punpcklwd   xmm4, xmm4

      movq      xmm2, qword ptr [edi+64*2]    // xmm2 = g = cg
      punpcklwd   xmm2, xmm2

      movdqa    xmm3, xmm4                    // xmm3 = co
      psubsw    xmm3, xmm2                    // xmm3 = r = co - cg
      paddsw    xmm4, xmm2                    // xmm4 = b = co + cg
```

```
movdqa      xmm7, xmm3                      // xmm7 = r
movdqa      xmm6, xmm2                      // xmm6 = g
movdqa      xmm5, xmm4                      // xmm5 = b

movdqa      xmm1, qword ptr [esi+eax+0*2]
paddsw      xmm1, SIMD_SSE2_word_128

paddsw      xmm3, xmm1                      // r0, r1, r2, r3       ( y + r )
paddsw      xmm2, xmm1                      // g0, g1, g2, g3       ( y + g )
psubsw      xmm1, xmm4                      // b0, b1, b2, b3       ( y - b )

packuswb    xmm3, xmm3                       // r0, r1, r2, r3, r0, r1, r2, r3
packuswb    xmm2, xmm2                       // g0, g1, g2, g3, g0, g1, g2, g3
packuswb    xmm1, xmm1                       // b0, b1, b2, b3, b0, b1, b2, b3

movdqa      xmm0, qword ptr [esi+eax+0*2+384*2]
paddsw      xmm0, SIMD_SSE2_word_128
packuswb    xmm0, xmm0

punpcklbw   xmm1, xmm0                       // b0, a0, b1, a1, b2, a2, b3, a3
punpcklbw   xmm3, xmm2                       // r0, g0, r1, g1, r2, g2, r3, g3
movdqa      xmm4, xmm3                       // r0, g0, r1, g1, r2, g2, r3, g3
punpcklwd   xmm3, xmm1                       // r0, g0, b0, a0, r1, g1, b1, a1
punpckhwd   xmm4, xmm1                       // r2, g2, b2, a1, r3, g3, b3, a3

movdqa      [edx+ 0], xmm3
movdqa      [edx+16], xmm4
add         edx, stride

movdqa      xmm2, qword ptr [esi+eax+8*2]
paddsw      xmm2, SIMD_SSE2_word_128

paddsw      xmm7, xmm2                      // r0, r1, r2, r3       ( y + r )
paddsw      xmm6, xmm2                      // g0, g1, g2, g3       ( y + g )
psubsw      xmm2, xmm5                      // b0, b1, b2, b3       ( y - b )

packuswb    xmm7, xmm7                       // r0, r1, r2, r3, r0, r1, r2, r3
packuswb    xmm6, xmm6                       // g0, g1, g2, g3, g0, g1, g2, g3
packuswb    xmm2, xmm2                       // b0, b1, b2, b3, b0, b1, b2, b3

movdqa      xmm0, qword ptr [esi+eax+8*2+384*2]
paddsw      xmm0, SIMD_SSE2_word_128
packuswb    xmm0, xmm0

punpcklbw   xmm2, xmm0                       // b0, a0, b1, a1, b2, a1, b3, a3
punpcklbw   xmm7, xmm6                       // r0, g0, r1, g1, r2, g2, r3, g3
movdqa      xmm5, xmm7                       // r0, g0, r1, g1, r2, g2, r3, g3
punpcklwd   xmm7, xmm2                       // r0, g0, b0, a0, r1, g1, b1, a1
punpckhwd   xmm5, xmm2                       // r2, g2, b2, a2, r3, g3, b3, a2

movdqa      [edx+ 0], xmm7
movdqa      [edx+16], xmm5
add         edx, stride

add         edi, 8*2
add         eax, 16*2
jl          loop2

add         ecx, 1
cmp         ecx, 4
jl          loop1
```

```
    }
}
```