Katholieke Hogeschool Brugge-Oostende
Departement Industriële Wetenschappen en Technologie

Leeds Metropolitan University

Diplomaproject in het kader van:

Voortgezette Opleiding Elektronisch Systeem Ontwerp
en
MSc in Electronic System Design

FAST IMAGE FILTERING ALGORITHMS IN MICROSOFT VISUAL C++ USING MMX-TECHNOLOGY

door

William De Cat

Supervisor:
Ing. H. Tassignon, Ph. D.

Oostende, september 1998

Abstract

The objective of this work is to create a program that is useful to the image-processing course. This course uses a mathematical approach to the subject, which is supported well by the MATLAB mathematical software package. The problem with MATLAB is that it is an interpreter and is therefore slow, especially for image processing.
There are several solutions to the problem. The preferred solution was to create a stand-alone application that can apply the filters computed by Matlab to any image.
The stand-alone application was written using Microsoft Visual C++. Its inline assembler was used to insert MMX routines speeding up the filtering process.
MMX-technology can definitely speed up application speed. The gain is not just a matter of percentages, but factors. One can expect the MMX-routine to be about 10 to 20 times faster than code using the traditional integer instruction.
A second advantage is the DSP-like features of MMX-technology. Packing and unpacking handle relatively complex operations in one instruction and the saturation feature generates better results.
The main problems with MMX-technology are data alignment and the fact that either assembler programming or standard functions from general MMX-libraries are required to benefit from MMX-technology. Standard libraries optimised for MMX generally use a lot of memory due to alignment conditions. It can be said that using MMX-technology produces faster code and better results, but that as more performance gain is wanted, more work is required.

This work also means to provide examples on how to deal with problems specific to programming using MMX-technology, such as replacing floating-point calculations by integer operations.

Table of Contents

Acknowledgements

At this time, I would like to direct a word of thanks to some people who helped make this publication possible.

Special thanks go to my supervisor Dr. Tassignon for his guidance and the opportunity to explore this recent processor technology in relation to image processing.

I would also like to thank the KHBO, Leeds Metropolitan University and the professors providing for the MSc. course which gave me the necessary knowledge to complete this work.

Also, a word of praise to the people that created the online manuals that can be found on the web-sites of Intel, the Joint Photographic Experts, and the Joint Motion Pictures Experts. These web-sites contain a lot of usable information.

Last but not least, I would like to thank those persons among my family and friends four their encouragement and support which helped me realise this work.

1 Introduction

The objective of this work is to create a program that is useful to the image-processing course. This course uses a mathematical approach to the subject, which is supported well by the MATLAB mathematical software package. The problem with MATLAB is that it is an interpreter and is therefore slow, especially for image processing. For instance, applying a filter to a greyscale image of 320 by 200 pixels can take up to 5 minutes on a Pentium 133! From an educational point of view, this is unacceptable. The newest version 5 of Matlab is significantly faster than version 4, but this means upgrading all licenses in the university, which is quite costly and still not fast enough (the filter still takes about 1 minute). Eventually, the combination of newer and faster PC's and the new version of Matlab will solve the problem. In the mean time, another solution is necessary.

There are several solutions to the problem. The preferred solution was to create a stand-alone application that can apply the filters computed by Matlab to any image.

The stand-alone application was written using Microsoft Visual C++. Its inline assembler was used to insert MMX routines speeding up the filtering process. A shareware library taken from [1] takes care of importing and exporting the images to different file formats.

2 Concept and techniques.

The first solution explored was the Matlab to C compiler. This utility transforms the Matlab function files, commonly known as ".m-files" to an ANSI-C file that can be compiled by some compilers, among which Microsoft Visual C++, to produce a MEX-file. This MEX-file is a slightly altered DLL-file in the Windows version of Matlab. Because the function is now compiled and does not have to be interpreted, the function is executed much faster. Tests with this system proved it inadequate: the tool does not always produce code that can be compiled without alterations. The best way to make these MEX-files is to write the C-code manually using the MATLAB C-library. This is unpractical for educational purpose, because the student would be tangling with the complexity of a C language combined with the Matlab library instead of the image-processing problem at hand.

Another disadvantage of the Matlab library is that it does not support MMX-technology, so a stand-alone application would be much faster if it were to use this new technology.

MMX, short for MultiMedia eXtensions, is based on SIMD (Single Instruction Multiple Data) instructions. The floating-point registers on board a Pentium are used to store one 64-bit, two 32-bit, four 16-bit or eight 8-bit integer values. The MMX-instructions added to the Pentium MMX and Pentium II instruction set perform the same operation (the usual accumulator and shift operations and some instructions specific to the data-formats) on the multiple integers in one of the MMX-registers. This way, several integers are processed in one instruction. For instance, if eight 8-bit integers have to be squared this can be done in one instruction! This is extremely interesting for image processing, because a pixel is usually represented by 3 bytes. Without MMX technology, each byte needs to be handled separately; with MMX technology, one can process multiple pixels at once.

Another advantage of the MMX instructions is that no wrap-around occurs. Suppose one tries to multiply 200 by 200 (result = 40000) and write the result to a 16-bit-register that is supposed to contain a signed word. The result of this operation would be that the overflow flag is set and the register would contain –25536, a negative value. MMX instruction can saturate if desired. This means the result of the above operation is the value closest to the theoretical result that is still in range, in this case 32767. In image processing wrap-around is very disturbing. That is why use of the saturation possibility in the MMX-instruction set produces far better results than traditional integer instructions.

Because the MMX registers overlap with the floating-point registers, MMX instructions cannot be mixed with floating-point instructions. That is why an MMX-processor must be locked in an MMX-state when executing MMX instructions. This happens automatically when the processor executes the first MMX-instruction. Until the processor encounters the EMMS (Empty MultiMedia State) instruction, all floating-point registers are inaccessible and their contents are lost when the first MMX-instruction is executed.

That is why, instead of multiplying an integer A with a floating-point number B and rounding the result to an integer C = round(A*B), one should multiply the floating-point number B with coefficient D and round to an integer E=round(B*D). Now, A can be multiplied by the integer E and integer-divided by the coefficient D to yield the result C=(A*B)/D. The result C is now calculated using integer operations only. If the coefficient D is a power of 2, integer dividing is the same as shifting, which is a simple and very fast operation.

In this project, MMX-technology was used to create a stand-alone Windows 32-bit application that is able to apply the filters computed with Matlab to an image. These images are read in from the common file-types by a freeware-library for the Microsoft Visual C++ programming language. This library stores the image to memory in the Windows DIB-format. This format can be displayed by a routine present in Visual C++. That is why this format is used throughout the project.

The image read in by the library is then converted to a 24-bit per pixel (24-bpp) image (the DIB-format supports a variety of data formats). This image is split to 6 8-bpp images (R, G, B, Y, U and V channels) to which the filters are applied using normal convolution. After applying the filter to one or more of these channels, the new 24-bpp image is computed from these altered channels and the other channels are updated if necessary. This approach has the advantage that one can view any channel immediately, because the channels are kept in memory from the first moment they are needed until the image is closed.

This approach makes it possible to apply any filter to the image that is done by multiplying in the frequency domain (Fourier transform). Among those filters are the standard high-pass, low-pass, band-pass and Wiener filters. The kernel generated with Matlab is written to a text file, which is written to the plug-in directory of the application. The application reads in those kernels and their names and puts them in a menu, enabling the student to apply those kernels to the image. Of course, having to restart the application each time a new kernel is computed would be impractical. That is why the application can update this menu at the user's request.

The shareware-library used to read in the images also includes filters that cannot be applied using convolution, such as the median filter. These filters can also be applied to the image, but they were not optimised for MMX-technology.

While this work was in progress, Intel launched their Intel Image Processing Library or IPL, which is available on their web-site for free downloading. Integrating this library could add functionality to the program, but not replace the work that was done here. First of all, gathering information and expertise in the field of MMX-programming was also a goal in this project. Since the IPL is compiled and the source code is unavailable, we would still not know how to go about writing programs using MMX-instructions. Second, the IPL transforms DIB-images to its own format, does the processing, and then converts back to DIB-format. The instructions in this project work on the DIB-format itself, so no conversion is necessary. This means the method presented here is more straightforward and shorter, which means the code can be used as an example to teach students how to program using MMX-technology. In other words, the IPL is a wonderful library if the goal is to create an application, but is useless from an educational point of view.

3 Colorspace Conversion

3.1 What are colorspaces?

In computer technology, images are usually represented by an array of pixels. Each pixel is defined by its red, green and blue component. This means that a pixel can be represented as a vector in a 3-dimensional space with saturated red, saturated green and saturated blue as multiples of unity vectors. These three unity vectors form a basis for this space. It is now possible to define a new basis in this space by choosing any 3 linearly independent unity vectors. Three new variables with a limited resolution and range can now be defined as multipliers for the new unity vectors. These 3 new variables combined with the new unity vectors define a finite volume in the space. It is this finite volume that is called a colorspace in this work. In plain English, each set of 3 variables representing a pixel generates a new colorspace. Here are the most common colorspaces:

- RGB: Red Green Blue
- YUV or YC_BC_R: luminance or brightness (Y) and two chrominance variables (U and V or C_B and C_R).
- HSB: Hue Saturation Brightness
- L*a*b*: again luminance and two chrominance variables.

For printing purposes, 3 variables are not enough and 4 variables are used: CMYK (Cyan Magenta Yellow and the black component K). CMYK is also considered to be a colorspace.

3.2 Why colorspace conversion?

The human eye is more sensitive (in terms of resolution) to luminance (intensity or brightness of light) than it is to the chrominance (colour of light). That is one reason why it is most interesting to filter images by filtering only the luminance component and leaving the chrominance information unchanged. In addition, filtering chrominance is too abstract. For instance, one could say that a filter removes high frequencies from the blue chrominance. What this means and how one can 'see' the effect is not explicable in plain English. Removing high frequencies from the luminance simply means that the image becomes more fluent, meaning edges become less abrupt. If one were to filter one or some of the variables in RGB colorspace, both luminance and chrominance would be affected at the same time, making it almost impossible to predict the result.

If filtering chrominance in a comprehensible way is desired, the HSB (Hue Saturation Brightness) colorspace is needed. The problem with HSB is that it takes floating point arithmetic to compute and therefore cannot be optimised for MMX-technology:

$$S = \sqrt{(B-Y)^2 + (R-Y)^2}$$

$$H = \arctan\left(\frac{R-Y}{B-Y}\right)$$

The square root and arctangent functions are quite slow no matter what equipment is used. That is why HSB is not supported in this project.

3.3 Constants in colorspace conversion

Conversion between spaces with different bases is usually performed using conversion matrices. In this case, since it is a three-dimensional space, it requires a 3 by 3 matrix. We will first handle the calculation from RGB to YUV variable per variable and then combine these equations into matrices in order to calculate the coefficients to transform back.

How the luminance (Y) value has to be computed from an RGB-triplet, is fully defined and agreed upon. This is done using the following formula:

$$y = 0.299 \cdot r + 0.587 \cdot g + 0.114 \cdot b$$

In this text, the following identifiers will replace the above constants:

$$y = c_{ry} \cdot r + c_{gy} \cdot g + c_{by} \cdot b$$

The coefficients come from measurements of the eye's sensitivity to red, green and blue light respectively.

Because red and blue are not dominantly present in the Y-signal, it is logical to let U and V be mainly dependant of respectively blue and red, so that the conversion from YUV to RGB increases the accuracy of the calculated R and B values. The calculated G value already has a large accuracy, because Y consists mostly of G.

In contrast with the Y-component, U and V components are not always computed in exactly the same way. For instance, calculation of U and V for television broadcast happens as follows:

$$u = 0.493 * (b - y)$$
$$v = 0.877 * (r - y)$$

When performing JPEG image compression, the method of calculation is different:

$$c_b = u = -\left(0.5 \cdot \frac{c_{ry}}{c_{ry} + c_{gy}}\right) \cdot r - \left(0.5 \cdot \frac{c_{gy}}{c_{ry} + c_{gy}}\right) \cdot g + 0.5 \cdot b$$

$$c_r = v = 0.5 \cdot r - \left(0.5 \cdot \frac{c_{gy}}{c_{gy} + c_{by}}\right) \cdot g - \left(0.5 \cdot \frac{c_{by}}{c_{gy} + c_{by}}\right) \cdot b$$

Calculating these constants yields:

$$c_b = u = -0.16874 \cdot r - 0.33126 \cdot g + 0.5 \cdot b$$
$$c_r = v = 0.5 \cdot r - 0.41869 \cdot g - 0.08131 \cdot b$$

Rearranging the above formula shows us that:

$$c_b = u = \left\{ \frac{0.5}{1 - c_{by}} = 0.56433 \right\} \cdot (b - y)$$

$$c_r = v = \left\{ \frac{0.5}{1 - c_{ry}} = 0.71327 \right\} \cdot (r - y)$$

An MMX-routine obtained from Intel uses constants that are rounded values from television broadcast systems:

$$u = 0.49 * (b - y)$$
$$v = 0.88 * (r - y)$$

This demonstrates that U and V are all calculated in the same basic way, except for two constant multiplication factors.

For our purpose, the CCIR 601-1 standard is used. This is the same standard as used in JPEG image compression.

To sum things up, we present the floating-point equations we will start from in this work:

$$\begin{aligned} y &= 0.299 \cdot r + 0.587 \cdot g + 0.114 \cdot b \\ c_b = u &= -0.16874 \cdot r - 0.33126 \cdot g + 0.5 \cdot b = 0.56433 \cdot (b - y) \\ c_r = v &= 0.5 \cdot r - 0.41869 \cdot g - 0.08131 \cdot b = 0.71327 \cdot (r - y) \end{aligned}$$

The constants above will also be represented by the following symbols:

$$y = c_{ry} \cdot r + c_{gy} \cdot g + c_{by} \cdot b$$
$$u = c_{ru} \cdot r + c_{gu} \cdot g + c_{bu} \cdot b$$
$$v = c_{rv} \cdot r + c_{gv} \cdot g + c_{bv} \cdot b$$

Now we have to convert the above floating-point operations to integer operations, since MMX-technology handles only integer operations. In computer vision, a pixel is usually represented by 3 bytes, one byte per variable. Higher resolutions are used only in highly professional systems (such as RGB 16-bit mode in PhotoShop) and cannot be displayed on a standard VGA-monitor or low-end colour printers.

In the used standard, r, g, b and y are floating point numbers ranging from 0 to 1. Representing these variables by an unsigned byte simply means multiplying all the equations by 255 and rounding. This does not affect the conversion constants for Y:

$$255 \cdot y = c_{ry} \cdot 255 \cdot r + c_{gy} \cdot 255 \cdot g + c_{by} \cdot 255 \cdot b$$
$$\Downarrow$$
$$Y = c_{gy} \cdot R + c_{gy} \cdot G + c_{by} \cdot B$$

Things are somewhat different for U and V. Since u and v range from –0.5 to 0.5, an offset has to be added to make U and V positive at all times. There are two ways to do this. The first is to add 0.5 to u and then truncate the result:

$$\begin{aligned} 255\cdot(u+0.5) &= 255\cdot(c_{ru}\cdot r + c_{gu}\cdot g + c_{bu}\cdot b + 0.5) \\ U &= trunc(c_{ru}\cdot R + c_{gu}\cdot G + c_{bu}\cdot B + 127.5) \\ 255\cdot(v+0.5) &= 255\cdot(c_{rv}\cdot r + c_{gv}\cdot g + c_{bv}\cdot b + 0.5) \\ V &= trunc(c_{rv}\cdot R + c_{gv}\cdot G + c_{bv}\cdot B + 127.5) \end{aligned}$$

The reason why this approach shouldn't be used here is that pure greyscale values are no longer greyscales after conversion. If u and v are zero, U and V will be 127. When calculating backward, u and v would no longer be zero and colour would be introduced into the image. One could also argue that a new floating-point number is introduced into the equation and this method is therefore not suited for MMX optimisation. This is not the case, since the whole equation still has to be multiplied in order to eliminate the floating-point coefficients. The advantage of this method is that the range of u and v is symmetrical.

The second approach is to add an integer offset to the truncated 255*u value:

$$\begin{aligned} 255\cdot u + offset &= c_{ru}\cdot 255\cdot r + c_{gu}\cdot 255\cdot g + c_{bu}\cdot 255\cdot b + offset \\ U &= c_{ru}\cdot R + c_{gu}\cdot G + c_{bu}\cdot B + offset \\ 255\cdot v + offset &= c_{rv}\cdot 255\cdot r + c_{gv}\cdot 255\cdot g + c_{bv}\cdot 255\cdot b + offset \\ V &= c_{rv}\cdot R + c_{gv}\cdot G + c_{bv}\cdot B + offset \end{aligned}$$

This method solves the problem of u and v being 0, but introduces another problem. In this application, *offset* could be either 127 or 128. To be compatible with common JPEG routines, the value 128 was chosen here. When calculating backwards, u and v now range from –0.502 to 0.498.

This means sacrificing exact representation of maximum red and maximum blue in order to get exact greyscales. This is no real problem, since saturated red and saturated blue hardly ever occur in natural images. These colours only occur in artificially generated images that usually do not require filtering.

We now still have floating-point coefficients. Before I can explain how to eliminate them, I have to explain an MMX-instruction that is the heart of all the routines in this work, namely the *pmaddwd* instruction. It is an acronym for *p*acked *m*ultiply and *add* *w*or*d*s. MMX registers are 64-bit registers, but they can also contain two 32-bit, four 16-bit or eight 8-bit integers. There are instructions for each data format. The *pmaddwd* instruction takes two registers filled with signed words and produces a register with two signed 32-bit results:

A	B	C	D
E	F	G	H
A*B+E*F		C*G+D*H	

All words are supposed to be signed 16-bit integers ranging from –32768 to +32767. The input variables range from 0 to 255 and can simply be transformed to words by left padding with zeros. The floating-point coefficients can be multiplied with a constant factor C_M. Dividing the result by the same factor would give us the desired result:

$$Y = \frac{C_M \cdot c_{gy} \cdot R + C_M \cdot c_{gy} \cdot G + C_M \cdot c_{by} \cdot B}{C_M}$$

$$U = \frac{C_M \cdot c_{ru} \cdot R + C_M \cdot c_{gu} \cdot G + C_M \cdot c_{bu} \cdot B}{C_M} + offset$$

$$V = \frac{C_M \cdot c_{rv} \cdot R + C_M \cdot c_{gv} \cdot G + C_M \cdot c_{bv} \cdot B}{C_M} + offset$$

The division can be replaced by a shift operation if C_M is a power of 2. C_M also has to be as large as possible: the larger it is, the more accurate the result is. Because the absolute values of the floating-point conversion coefficients range from 0.114 to .587, the maximum possible value for C_M is:

$$C_M^{\max} = \frac{32767}{.587} \approx 55821$$

The exponent of the largest power of 2 that is smaller than 55821 is:

$$trunc\left(\frac{\log(55821)}{\log(2)}\right) = 15$$

So C_M must be 2^{15} or 32768. Let's calculate the new coefficients; we will not round at this time:

$$Y = (9797.63 \cdot R + 19234.82 \cdot G + 3735.55 \cdot B) >> 15$$
$$U = (-5529.14 \cdot R - 10854.90 \cdot G + 16384 \cdot B) >> 15 + offset$$
$$V = (16384 \cdot R - 13719.55 \cdot G - 2664.45 \cdot B) >> 15 + offset$$

Above constants will be replaced by the following identifiers:

$$Y = (CRY \cdot R + CGY \cdot G + CBY \cdot B) >> 15$$
$$U = (CRU \cdot R + CGU \cdot G + CBU \cdot B) >> 15 + offset$$
$$V = (CRV \cdot R + CGV \cdot G + CBV \cdot B) >> 15 + offset$$

There could be some discussion as to how CRY, CBY, CGV and CBV have to be rounded. However, there are some rules to keep in mind when rounding the above values. If the pixel is a pure greyscale value, then R=G=B=Y and U and V must be 0. This means:

$$Y = (CRY \cdot Y + CGY \cdot Y + CBY \cdot Y) / 2^{15}$$
$$offset = (CRU \cdot Y + CGU \cdot Y + CBU \cdot Y) / 2^{15} + offset$$
$$offset = (CRV \cdot Y + CGV \cdot Y + CBV \cdot Y) / 2^{15} + offset$$

$$\Downarrow$$

$$CRY + CGY + CBY = 2^{15} = 32768$$
$$CRU + CGU + CBU = 0$$
$$CRV + CGV + CBV = 0$$

This a good time to start using the matrix notation:

$$\begin{bmatrix} R & G & B \end{bmatrix} \times \frac{1}{2^{15}} \begin{bmatrix} CRY & CRU & CRV \\ CGY & CGU & CGV \\ CBY & CBU & CBV \end{bmatrix} = \begin{bmatrix} Y & U & V \end{bmatrix}$$

Assuming CRY, CBY, CGV and CBV can be rounded both towards 0 and towards $\pm\infty$, only the 4 following combinations satisfy the above rules:

$$\begin{bmatrix} 9797 & -5529 & 16384 \\ 19235 & -10855 & -13719 \\ 3736 & 16384 & -2665 \end{bmatrix} or \begin{bmatrix} 9797 & -5529 & 16384 \\ 19235 & -10855 & -13720 \\ 3736 & 16384 & -2664 \end{bmatrix}$$

$$or$$

$$\begin{bmatrix} 9798 & -5529 & 16384 \\ 19235 & -10855 & -13719 \\ 3735 & 16384 & -2665 \end{bmatrix} or \begin{bmatrix} 9798 & -5529 & 16384 \\ 19235 & -10855 & -13720 \\ 3735 & 16384 & -2664 \end{bmatrix}$$

In order to make a choice we will need the coefficients for the reverse calculation:

$$\begin{bmatrix} R & G & B \end{bmatrix} = \begin{bmatrix} Y & U & V \end{bmatrix} \times \left(\frac{1}{2^{15}} \begin{bmatrix} CRY & CRU & CRV \\ CGY & CGU & CGV \\ CBY & CBU & CBV \end{bmatrix} \right)^{-1}$$

With the accuracy used below, the result of the inversion is the same for all 4 possible RGB-YUV conversion matrices:

$$\begin{bmatrix} c_{yr} & c_{yg} & c_{yb} \\ c_{ur} & c_{ug} & c_{ub} \\ c_{vr} & c_{vg} & c_{vb} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ -0.00001 & -0.34408 & 1.77203 \\ 1.40197 & -0.71414 & -0.00003 \end{bmatrix}$$

Again, we need to multiply this matrix with the largest possible power of 2 smaller than $(2^{15}-1)/1.77203$, being 2^{14} or 16384. Multiplying by 2^{14} again yields the same result for all 4 RGB-YUV matrices and the used accuracy. No rounding is performed at this time:

$$\begin{bmatrix} CYR & CYG & CYB \\ CUR & CUG & CUB \\ CVR & CVG & CVB \end{bmatrix} = \begin{bmatrix} 16384 & 16384 & 16384 \\ -0.14 & -5637.46 & 29032.93 \\ 22969.95 & -11700.44 & -0.45 \end{bmatrix}$$

Again, some variables are up for discussion: CUG and CVG. CVB should be 0 for two reasons:

- CVB should be 0 theoretically, the –0.45 value is due to rounding in the RGB-YUV matrix.
- Introducing a one here would mean having to perform an extra multiply-add instruction and that would decrease conversion speed significantly.

The pure greyscale case does not help us to decrease possibilities here: it only tells us that CYR=CYG=CYB=16384, which is already the case. So we have the following four possible YUV-RGB conversion matrices:

$$\begin{bmatrix} 16384 & 16384 & 16384 \\ 0 & -5637 & 29033 \\ 22970 & -11700 & 0 \end{bmatrix} or \begin{bmatrix} 16384 & 16384 & 16384 \\ 0 & -5638 & 29033 \\ 22970 & -11700 & 0 \end{bmatrix}$$

or

$$\begin{bmatrix} 16384 & 16384 & 16384 \\ 0 & -5637 & 29033 \\ 22970 & -11701 & 0 \end{bmatrix} or \begin{bmatrix} 16384 & 16384 & 16384 \\ 0 & -5638 & 29033 \\ 22970 & -11701 & 0 \end{bmatrix}$$

We now have 4 possible RGB-YUV conversion matrices and 4 possible YUV-RGB conversion matrices. The question now is which one to choose. To make that decision, I wrote a program that converts all possible RGB values (all 16.8 million of them!) to YUV and back to RGB again. In the process, the mean Euclidean distance of the conversion is recorded. Executing this program for all 16 combinations of possible conversion matrices yields the mean Euclidean distance in function of the constants used. The mean Euclidean distance is computed as follows:

$$R_n, G_n, B_n \rightarrow Y_n, U_n, V_n \rightarrow R_{n'}', G_n', B_n'; n = 0...255$$

$$MED = \frac{1}{256^3} \cdot \sum_{n=0}^{255} \sqrt{\left(R_n' - R_n\right)^2 + \left(G_n' - G_n\right)^2 + \left(B_n' - B_n\right)^2}$$

The formula is derived from the Euclidean distance between the two vectors (R, G, B) and (R', G', B'). This distance was chosen, because the human eye will experience the difference according to this distance. Taking the mean is done to make the numbers smaller and easier to interpret. For further explanation of the program itself and its output, refer to appendix A. Using this procedure, we found that the following combination yields a minimum MED of 2.870744:

$$\underset{RGB-YUV}{\begin{bmatrix} 9798 & -5529 & 16384 \\ 19235 & -10855 & -13719 \\ 3735 & 16384 & -2665 \end{bmatrix}} \underset{YUV\text{-}RGB}{\begin{bmatrix} 16384 & 16384 & 16384 \\ 0 & -5637 & 29033 \\ 22970 & -11700 & 0 \end{bmatrix}}$$

3.4 Properties of the conversion

The reader may have noticed that the mean Euclidean distance of the conversion is rather high. Only 372 out of 16.8 million pixels are unaffected by the conversion to YUV and back. All 256 pure greyscale values are among those 372. It is therefore safe to say that, apart from some exceptions, only greyscale pixels are unaffected by the conversion. Remember that this is mainly because of decisions we made when calculating the constants: not affecting greyscale pixels has always been our primary concern.

These choices, however, are not the ones to blame for this poor result. The main reason the MED is that poor is simply because the RGB-colorspace is smaller than the YUV-colorspace. In other words, all RGB pixels have a corresponding pixel in YUV space, but not all pixels in YUV space have a corresponding pixel in RGB space. For instance, let Y=0; U and V must equal offset. If they don't, you end up with negative R, G, or B values which are rounded to zero by the pack and saturate commands in the routine. This fact alone means that 65535 pixels in YUV-space do not have a corresponding pixel in RGB space. This means that the pixels converted from RGB-space have a smaller resolution in YUV space.

Measurements show that the 16.8 million pixels in RGB-space are projected to only 1.4 million pixels in YUV-space. This means that, in average, 12 pixels in RGB-space are projected to the same pixel in YUV space. It is this decrease in resolution that is responsible for the pour MED. It is inherent to the colorspace conversion and cannot be helped, except by an increase of the number of bits used in YUV space. This was not done here, because it is uncommon and because it would double the size of an already large data structure!

All this might seem very alarming, but only true experts with perfect vision can distinguish between a picture with 16 million possible colours and the same picture with only 65536 possible colours! In our case, we still have over 1 million possible colours. So we can be sure the average human eye will never know the difference!

One should also note that, assuming the RGB unity vectors are orthogonal, the YUV unity vectors are not. That is part of the reason why the RGB-space is smaller.

3.5 MMX algorithm for conversion from RGB to YUV

3.5.1 Introduction

First, a word on how Windows stores DIB's (Device Independent Bitmaps). This data format starts with a header containing information on how the pixels are stored. Our algorithm works with only one data format, namely 3 bytes per pixel. An array of pixels follows the header. In the data format we support in our routines, pixels are stored as 3 consecutive bytes, with B on the lowest address, followed by G and R.

The MMX instruction MOVQ puts the byte on the lowest address in the least significant byte (LSB) in the MMX register. This means that a MOVQ instruction will fill the MMX register with RGBRGB…RGB values when reading the MMX register from left to right.

More detailed explanations of MMX-instructions can be found in [3].

To make optimal use of memory transfer capabilities of MMX-technology, the data must be transferred 8 bytes at a time. A pixel is 3 bytes wide. To avoid unnecessary loads, the data should be processed by 8 pixels or 8*3=21 bytes.

3.5.2 Arithmetic with PMADDWD: the heart of the conversion

Here is how the 8 Y values can be computed from the input using PMADDWD and PADDD instructions:

R3 G3 B3 R2	B3 R2 G2 B2	R1 G1 B1 R0	B1 R0 G0 B0
CBY CGY 0 CRY	CBY 0 CGY CBY	CRY CGY 0 CRY	CBY 0 CGY CBY
PMADDWD: YRG3 YR2	PMADDWD: YB3 YGB2	PMADDWD: YRG1 YR0	PMADDWD: YB1 YGB0
PADDD: Y3	Y2	PADDD: Y1	Y0

R7 G7 B7 R6	B7 R6 G6 B6	R5 G5 B5 R4	B5 R4 G4 B4
CBY CGY 0 CRY	CBY 0 CGY CBY	CRY CGY 0 CRY	CBY 0 CGY CBY
PMADDWD: YRG7 YR6	PMADDWD: YB7 YGB6	PMADDWD: YRG5 YR4	PMADDWD: YB5 YGB4
PADDD: Y7	Y6	PADDD: Y5	Y4

Table 2-3.1

As can be seen, two consecutive Y values are calculated by two PMADDWD instructions, each calculating intermediate results. A PADDD (Packed ADD Doubleword) instruction adds up the corresponding intermediate results, yielding two completely calculated Y values. Let's take a more detailed look at how Y1 and Y0 are calculated:

op	source operand	destination operand before operation	destination operand after operation
PMADDWD	CBY|0|CGY|CBY	B1|R0|G0|B0	YB1|YGB0
PMADDWD	CRY|CGY|0|CRY	R1|G1|B1|R0	YRG1|YR0
PADDD	YB1|YGB0	YRG1|YR0	Y1|Y2

Where:

- YB1 = CBY*B1+ 0 *R0 = CBY*B1
- YGB0 = CGY*G0+CBY*B0
- YRG1 = CRY*R1+CGY*G1
- YR0 = 0 *B1+CRY*R0 = CRY*R0
- Y1 = YB1 + YRG1 = CRY*R1+CGY*G1+CBY*B1
- Y0 = YGB0 + YR0 = CRY*R0+CGY*G0+CBY*B0

In our routine, the source operands for the PMADDWD instructions above are not kept in registers, because there are not enough registers to do so. The PMADDWD-instruction can also read its source operand from memory, and this is the case here. This hardly slows things down, because they are loaded only once from main memory and kept in the primary cache, which is practically as fast as the internal registers.

The rest of the Y values are calculated in a similar way as shown in table 1-1 above.

Exactly the same procedure is followed to calculate U and V, only the constants differ.

Now that I have explained the heart of the algorithm, two things remain to be explained: how the words in the source operands are generated from the byte input and how to convert eight 32-bit words to eight bytes.

3.5.3 Transforming the input

Transforming the input is quite straightforward. The following tables present an overview:

B5 R4 G4 B4 R3 G3 B3 R2	G2 B2 R1 G1 B1 R0 G0 B0	
PUNPCKLBW: R3 G3 B3 R2	MOVQ: G2 B2 R1 G1 B1 R0 G0 B0	G2 B2 R1 G1 B1 R0 G0 B0
MOVQ: R3 G3 B3 R2	PSRLQ: 0 0 G2 B2 R1 G1 B1 R0	PUNPCKLBW:
PSLLQ: B3 R2 0 0	MOVQ: 00G2B2R1G1B1R0 / 00G2B2R1G1B1R0	
	PUNPCKHBW: 0 0 G2 B2	PUNPCKLBW:
PADDW: B3 R2 G2 B2		
R3 G3 B3 R2		B1 R0 G0 B0

R7 G7 B7 R6 G6 B6 R5 G5		B5 R4 G4 B4 R3 G3 B3 R2
MOVQ: R7 G7 B7 R6 G6 B6 R5 G5	R7 G7 B7 R6 G6 B6 R5 G5	PUNPCKHBW: B5 R4 G4 B4
PUNPCKHBW:	PSLLQ: B7 R6 G6 B6 R5 G5 0 0	MOVQ: B5 R4 G4 B4
	MOVQ: B7R6G6B6R5G500 / B7R6G6B6R5G500	PSRLQ:
	PUNPCKHBW:	PUNPCKLBW: R5 G5 0 0
		0 0 B5 R4
	PADDW: R5G5B5R4	
R7 G7 B7 R6		B5 R4 G4 B4

Table 2-3.2

The important instructions used here are the PUNPCKLBW (Packed UNPaCK Low Byte to Word) and PUNPCKHBW (Packed UNPaCK High Byte to Word) instructions. These instructions convert bytes to words by interleaving two quadwords byte-wise. Assuming the destination operand is filled with the bytes A7 through A0 and the source operand with bytes B7 through B0, the result of the unpack-byte-to-word would be the words B7A7 through B0A0. The PUNPCKLBW instruction returns the lower 4 words, namely B3A3 through B0A0, while the PUNPCKHBW instruction returns the higher 4 words, namely B7A7 through B3A3. In our application, the source operand is always 0. This means that the lower or higher 4 bytes in the destination operand are transformed from bytes to words by putting the byte in the least significant portion of the word. Note that putting the byte in the highest significant portion of the word would also be possible, but since the words are supposed to be signed and the bytes aren't, this isn't a good idea: bytes with a value greater than 127 would become negative words.

Now that we know how the PUNPCKXBW instructions work, we can look further into the transformation process.

Here is the list of instructions processing the first loaded quadword; the other quadwords are subjected to a similar process:

- MOVQ: load `G2B2R1G1B1R0G0B0` into an MMX register
- MOVQ: copy `G2B2R1G1B1R0G0B0` to another MMX register
- PSRLQ: shift `G2B2R1G1B1R0G0B0` right by 2 bytes, shifting in zeros yielding `00G2B2R1G1B1R0`
- MOVQ: copy `00G2B2R1G1B1R0` to another MMX register
- PUNPCKHBW: unpack `00G2B2R1G1B1R0` high, yielding `00G2B2`
- PUNPCKLBW: unpack `00G2B2R1G1B1R0` low, yielding **`R1G1B1R0`**
- PUNPCKLBW: unpack `G2B2R1G1B1R0G0B0` low, yielding **`B1R0G0B0`**
- PADDW: The 00G2B2 result will be added byte-per-byte to B3R200 resulting from the transformation process of the second quadword.

MOVQ: MOVe Quadword.

PSRLQ: Packed Shift Right Logical Quadword.

PADDW: Packed Add Word.

The other two quadwords are processed using a similar process according to table 1-2.

3.5.4 Formatting the output: packing.

The results of the PMADDWD operations are multiplied by 2^{15} and still need to be shifted right by 15 bits. This can be done using the PSRAD (Packed Shift Right Arithmetic) instruction. An arithmetic shift will retain the sign bit: if the value is negative, ones are shifted in, otherwise, zeros are shifted in. Therefore, an arithmetic right shift is the same as dividing a 2's complement integer by a power of 2, which is exactly what we want.

After the shift, the results can be packed from signed doublewords to signed words using the PACKSSDW (PACK and Saturate Signed Doublewords to Words) instruction and from signed words to unsigned bytes by the PACKUSWB (PACK Unsigned and Saturate Words to Bytes) instruction. Both instructions take two quadwords and reduce the size of the contained integers by half and put them one after the other in one quadword. If the integer cannot fit in its new size, the new integer gets the closest possible value to the old integer (this is called saturating). That is why this instruction is so interesting: no wrap-around occurs.

Using traditional truncation, trying to fit 40000 in a signed word would result in –25536; using PACKSSDW it results in 32767.

For U and V, an offset must be added before packing from words to bytes. This can be done in two places: before or after the PACKSSDW instruction. Naturally, the constants that have to be used for this differ depending on the place adding the offset occurs. Here, it will be done in whatever place is most convenient for coding. The reasons for this will be explained later.

Here is an overview of the output processing:

Y7 Y6	Y5 Y4	Y3 Y2	Y1 Y0
PSRAD: Y7 Y6	PSRAD: Y5 Y4	PSRAD: Y3 Y2	PSRAD: Y1 Y0
PACKSSDW: Y7 Y6 Y5 Y4		PACKSSDW: Y3 Y2 Y1 Y0	
PACKUSWB: Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0			

U7 U6	U5 U4	U3 U2	U1 U0	V7 V6	V5 V4	V3 V2	V1 V0
PSRAD: U7 U6	PSRAD: U5 U4	PSRAD: U3 U2	PSRAD: U1 U0	PSRAD: V7 V6	PSRAD: V5 V4	PSRAD: V3 V2	PSRAD: V1 V0
PADDD: U7 U6	PADDD: U5 U4	PACKSSDW: U3 U2 U1 U0		PACKSSDW: V7 V6 V5 V4		PACKSSDW: V3 V2 V1 V0	
PACKSSDW: U7 U6 U5 U4		PADDWD: U3 U2 U1 U0		PADDWD: V7 V6 V5 V4		PADDWD: V3 V2 V1 V0	
PACKUSWB: U7 U6 U5 U4 U3 U2 U1 U0				PACKUSWB: V7 V6 V5 V4 V3 V2 V1 V0			

3.5.5 Combining concurrent processes: pairing and register limitations

You will not find the above operations being executed one after the other in the MMX routines. Instead, the operations for transforming the input, calculating the values and formatting the output are intertwined. There are two reasons for this.

The first reason is that there are not enough MMX registers to store all intermediate results of all processes. This means that the first quadword has to be loaded and fully processed before the second quadword can be loaded. This alone means we have to intertwine the processes calculating Y, U and V.

The second reason is pairing. A Pentium processor can execute two instructions in parallel, but there are restrictions. An obvious restriction is dependence: if a second instruction uses the result of the previous instruction, these two instructions cannot be processed at the same time because the second instruction has to wait for the first one to complete. Because instructions in the same process always depend on the result of previous instruction, two instructions from one process can never be paired. That is why instructions from different processes have to follow each other, so they can be paired. More detailed information on pairing can be found in [2].
Because of these two reasons, an instruction belonging to one process is followed by an instruction belonging to another process in the final code. Because that can be quite confusing, it was decided to explain the separate processes instead of explaining the instruction sequence.

3.6 MMX algorithm for conversion from YUV to RGB

3.6.1 Introduction

There are two zeroes in the conversion matrix. This means that converting YUV to RGB takes two multiply-add operations less. This should result in a faster algorithm, but in this case, it does not. The conversion from YUV to RGB is, on the contrary, slower! The reason for this is that the time gained in the actual calculation of RGB is lost during transforming the input and output of the conversion routine. As will become apparent, there are a lot of problems associated with the fact that the Y, U, and V components come from different DIB's. The conversion requires a lot of shift-operations that cannot be paired.

3.6.2 Arithmetic with PMADDWD: the heart of the conversion

Ironically, the reason why the YUV to RGB routine is slower is the fact that it takes less multiply-add operations to calculate the result. The problem is that because of this, results are in the wrong position and shift operations are necessary to add intermediate results. The following table demonstrates this:

Y1 V1 Y1 U1	Y1 V1 Y1 U1	Y0 V0 Y0 U0	Y0 V0 Y0 U0
CYR CVR 0 CUG	CYG CVG CYB CUB	CYR CVR 0 CUG	CYG CVG CYB CUB
PMADDWD: R1 IGU1	PMADDWD: IGV1 B1	PMADDWD: R0 IGU0	PMADDWD: IGV0 B0
	MOVQ: IGV1 B1 / IGV1 B1	MOVQ: R0 IGU0 / R0 IGU0	
	PSRLQ: 0 IGV1 / PSLLQ: B1 0	PSRLQ: 0 R0 / PSLLQ: IGU0 0	
PADDD: R1 G1	PADDD: B1 R0	PADDD: G0 B0	

The same thing happens for the other 6 pixels. One could of course argue that the shift operations would not be necessary if intermediate results were calculated with an extra PMADDWD-instruction. The following table shows this:

Y1 V1 Y1 U1	Y1 U1 Y1 V1	Y1 U1 Y0 V0	Y0 U0 Y0 V0	Y0 V0 Y0 U0
PMADDWD: R1 IGU1	PMADDWD: 0 IGV1	PMADDWD:	PMADDWD: IG0 0	PMADDWD: IG0 B0
PADDD: R1 G1		B1 R0	PADDD: G0 B0	

This shows that only 7 operations would replace the 13 operations, but it also shows that this makes transforming the input more complex and longer, so the progress made here would be totally lost when transforming the input. That is why the first algorithm is chosen.

3.6.3 Transforming the input

As mentioned before, transforming the input is not as straightforward as it was in the RGB to YUV routine. This is mainly because the input is not organised by pixel, but by channel.
The following table demonstrates the preparation procedure:

Y7-0		Y7-0		Y7-0		Y7-0	
V7-0		U7-0		V7-0		U7-0	
PUNPCKHBW: YV7-4		PUNPCKHBW: YU7-4		PUNPCKLBW: YV3-0		PUNPCKLBW: YU3-0	
MOVQ: YV7-4		YV7-4		MOVQ: YV3-0		YV3-0	
YU7-4		YU7-4		YU3-0		YU3-0	
PUNPCKHWD: *YVYU7&6*		PUNPCKLWD: *YVYU5&4*		PUNPCKHWD: *YVYU3&2*		PUNPCKLWD: *YVYU1&0*	
MOVQ: YVYU7&6	MOVQ: YVYU7&6	MOVQ: YVYU5&4	MOVQ: YVYU5&4	MOVQ: YVYU3&2	MOVQ: YVYU3&2	MOVQ: YVYU1&0	MOVQ: YVYU1&0
0	0	0	0	0	0	0	0
PUNPCKHBW: YVYU7	PUNPCKLBW: YVYU6	PUNPCKHBW: YVYU5	PUNPCKLBW: YVYU4	PUNPCKHBW: YVYU3	PUNPCKLBW: YVYU2	PUNPCKHBW: YVYU1	PUNPCKLBW: YVYU0
OFFSETWN	OFFSETWN	OFFSETWN	OFFSETWN	OFFSETWN	OFFSETWN	OFFSETWN	OFFSETWN
PADDW: YVYU7	PADDW: YVYU6	PADDW: YVYU5	PADDW: YVYU4	PADDW: YVYU3	PADDW: YVYU2	PADDW: YVYU1	PADDW: YVYU0

The last operation subtracts the offset from U and V by adding 0;-128;0;-128 (=OFFSETWN) word-by-word (PADDW instruction) to the YVYU-variables.
The values printed in italic show the last stage of conversion where there are only 4 registers to store. That is why the preparation stage stops there in the code. Further unpacking is done as the conversion progresses. For instance, YVYU1 and YVYU0 are calculated and converted to RGB while YVYU7&6, YVYU5&4 and YVYU3&2 have not been processed yet. This way, all variables can be kept in registers and no memory access is necessary.

The PUNPCKXWD instructions are similar to the PUNPCKXBW instructions, except that they form doublewords from words instead of words from bytes. Suppose the destination operand of the instruction is filled with four words A3 through A0 and the source operands filled with the four words B3 through B0, the result of the PUNPCKLWD instruction would be B1A1B0A0. PUNPCKHWD would result in B3A3B2A2. In this case the words are composed by two bytes, for instance Y7U7.

3.6.4 Formatting the output: packing.

Packing in this routine is quite simple: all data is in the right sequence and no offsets need to be added. All that has to be done is dividing by 2^{14} by shifting right 14 bits:

R7 G7	B7 R6	G6 B6	R5 G5	B5 R4	G4 B4	R3 G3	B3 R2	G2 B2	R1 G1	B1 R0	G0 B0
PSRAD: R7 G7	PSRAD: B7 R6	PSRAD: G6 B6	PSRAD: R5 G5	PSRAD: B5 R4	PSRAD: G4 B4	PSRAD: R3 G3	PSRAD: B3 R2	PSRAD: G2 B2	PSRAD: R1 G1	PSRAD: B1 R0	PSRAD: G0 B0
PACKSSDW: R7 G7 B7 R6		PACKSSDW: G6 B6 R5 G5		PACKSSDW: B5 R4 G4 B4		PACKSSDW: R3 G3 B3 R2		PACKSSDW: G2 B2 R1 G1		PACKSSDW: B1 R0 G0 B0	
PACKUSWB: R7 G7 B7 R6 G6 B6 R5 G5				PACKUSWB: B5 R4 G4 B4 R3 G3 B3 R2				PACKUSWB: G2 B2 R1 G1 B1 R0 G0 B0			

The instructions used here are the same as in 2.5.4

3.7 MMX algorithms and data alignment

3.7.1 Avoiding invalid memory accesses

The routines represented above process 8 pixels in one run. A problem arises when processing the last pixels of the last line in the image buffer. If there are less than 8 pixels left, these routines try to read in pixels that do not exist and read memory beyond the memory reserved by Windows for that image and generate a protection fault. To avoid this, simplified version of the above routines were created processing 4, 2, or 1 pixel. These routines take into account that each line of a Windows DIB is doubleword (32-bit) aligned. This means that for a line consisting of 9 pixels and therefore requiring 27 bytes, 28 bytes will be reserved and 1 byte extra can be read, even though it does not contain anything useful. This might be useful, because being able to read in 4 bytes in stead of 3 bytes means being able to use the MMX MOVD instruction moving a doubleword from or to memory to or from the lower doubleword of an MMX register. This saves the move operation from normal integer registers to MMX registers and is therefore faster.

The C-routines containing these routines use the 8-pixel routine as many times as possible, then, if there are 4 or more pixels to process, process the next 4 pixels with the 4-pixel routine, the rest with the 2-pixel and 1-pixel routines if necessary. This way, unauthorised memory accesses are avoided. Note that Windows 95/98 would never complain about memory transgressions but Windows NT does. The strategy used here guarantees correct functioning under both operating systems.

3.7.2 Optimising alignment for speed

There is another aspect of MMX memory transfer instructions that needs to be taken into account. The memory transfers perform poorly when the data is not correctly aligned. The MOVQ instruction performs adequately only when the pointer to memory is a multiple of 8 (quadword-aligned); the MOVD operation performs best when the pointer is a multiple of 4 (doubleword aligned). The difference cannot be ignored: the RGB to YUV routine is 24% slower when the data is not properly aligned, while the YUV to RGB routine 12% slower. How the effect of data alignment was tested and the results of the tests can be found in appendix B.

This shows the importance of starting the conversion with the right pixel. When processing selected areas instead of the entire image, the 8-bit routine should start with a pixel that has a pointer to its B-byte that is a multiple of 8. The 8-byte routine could start with the first pixel in the line that is within the selection area and satisfies the alignment condition. The pixels before that pixel can be processed with the 4, 2 and 1-pixel routines. Another solution would be to convert more pixels than necessary. One could start with the first pixel outside the selection area that satisfies the alignment condition. This is faster in some cases, because the 8-pixel routine is much faster than the 4, 2, and 1-pixel routines. The problem is that it is not always applicable, because these pixels might not exist.

Mapping all possible alignment conditions for the first pixel to be processed and calculating the time required for getting to the first pixel inside the selection area satisfying alignment requirements show which method is faster. The following tables do this:

RGB- YUV

p%8	p′-p (bytes)	p′-p (pels)	time 1-pel (ns)	time 2-pel (ns)	time 4-pel (ns)	total time (µs)	p″-p (bytes)	p′-p″ (bytes)	time (µs)
0	0	0	0	0	0	0	0	0	0
1	15	5	491	0	535	1.026	-9	24	**0.650**
2	6	2	0	490	0	**0.490**	-18	24	0.650
3	21	7	490	490	535	1.515	-27	48	**1.300**
4	12	4	0	0	535	**0.535**	-12	24	0.650
5	3	1	491	0	0	**0.491**	-21	24	0.650
6	18	6	0	490	535	**1.025**	-30	48	1.300
7	9	3	492	490	0	0.982	-15	24	**0.650**

YUV-RGB

p%8	p′-p (bytes)	p′-p (pels)	time 1-pel (ns)	time 2-pel (ns)	time 4-pel (ns)	total time (µs)	p″-p (bytes)	p′-p″ (bytes)	time (µs)
0	0	0	0	0	0	0	0	0	0
1	15	5	491	0	597	1.088	-9	24	**0.803**
2	6	2	0	583	0	**0.583**	-18	24	0.803
3	21	7	430	583	597	1.610	-27	48	**1.606**
4	12	4	0	0	597	**0.597**	-12	24	0.803
5	3	1	490	0	0	**0.490**	-21	24	0.803
6	18	6	0	583	597	**1.180**	-30	48	1.606
7	9	3	430	583	0	1.013	-15	24	**0.803**

When enlarging the selection area is faster, does not depend on the routine used. It is faster to increase the selection area if possible in three cases, namely if the rest of division of the pointer by 8 is 1, 3 or 7. Note that they are all cases where the 1-pixel routine was used. This routine is very slow compared to the other routines. This is because this routine loads its variables from memory to integer registers and copies them from integer register to MMX registers, an operation that takes a lot of time.

The routine calling the MMX-routines checks alignment first and then uses the fastest way to start the conversion, followed by several executions of the 8-pixel routine.

The same problem occurs at the end of the line being processed. Also here, enlarging the area can mean faster processing. The following tables calculate the optimum solutions:

RGB- YUV

number of pixels left	time 4-pel (ns)	time 2-pel (ns)	time 1-pel (ns)	total time (µs)	time 8-pel (µs)
0	0	0	0	0	0
1	0	0	476	**0.476**	0.650
2	0	475	0	**0.475**	0.650
3	0	475	491	0.966	**0.650**
4	520	0	0	**0.520**	0.650
5	520	0	474	0.994	**0.650**
6	520	490	0	1.010	**0.650**
7	520	490	490	1.500	**0.650**

YUV-RGB

number of pixels left	time 4-pel (ns)	time 2-pel (ns)	time 1-pel (ns)	total time (µs)	time 8-pel (µs)
0	0	0	0	0	0
1	0	0	430	**0.430**	0.803
2	0	468	0	**0.468**	0.803
3	0	468	429	0.896	**0.803**
4	545	0	0	**0.545**	0.803
5	545	0	429	0.974	**0.803**
6	545	468	0	1.013	**0.803**
7	545	468	430	1.443	**0.803**

The execution times used here take into account alignment. Here the selection area is increased if there are 3, 5, 6 or 7 pixels are left and if enlarging the area is possible.

More detailed information on alignment can be found in [2].

4 Filtering algorithms

4.1 Why circular convolution

Most filters applied to images are based on point-by-point multiplication in the frequency-domain. Normally, this requires applying the Fourier transform to the image, multiplying the Fourier transform point-by-point with a filtering matrix, often referred to as a kernel, and applying the inverse Fourier transform to the result. The problem with this approach is that the Fourier transform is a calculation-intensive operation with complex values as a result. The result is a process that is slow and requires a lot of memory. However, there is a way around the Fourier transform, namely circular convolution. In this approach, the inverse Fourier transform is applied to the kernel designed for use in the Fourier space. Once this kernel is available in the space-domain, it can be applied to the image using circular convolution as many times as necessary without having to calculate the Fourier-transform of the image or using complex values.

The only instructions used in circular convolution are multiplication and addition. In fact, the circular convolution can be calculated entirely using multiply-add instructions only. This makes circular convolution ideal for MMX-optimisation. The only problem is again that the space kernel usually contains floating-point numbers smaller than one. This problem will be dealt with in exactly the same way as was done in the colorspace conversion, except that here, the multiplication factor will be variable.

4.2 Circular convolution

The strict mathematical definition of the 2-dimensional convolution is:

$$y(m',n') = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x(m,n)\cdot h(m'-m,n'-n)\cdot dm\cdot dn$$

$$y(m,n) = x(m,n) * h(m,n)$$

There are many ways to implement the convolution with digital systems, but they always have the following general form:

$$y(m',n') = \sum_m \sum_n x(m,n)\cdot h(m'-m,n'-n)$$

The difference in digital convolution algorithms is the way in which the negative indices in this summation are dealt with. The most common linear convolution method folds and shifts the h-sequence to generate the h(m'-m, n'-n) sequence. This method yields a result with a larger dimension than the input sequence.

Circular convolution assumes the kernel h and input signal x is periodic and the incoming sample is exactly one period long. This means that if the signal contains M by N samples,

$$h\big((m+u\cdot M),(n+v\cdot N)\big) = h(m,n)$$

$$u,v \in Z$$

If an index is negative, we can simply add M or N to the index making the index positive. The result is a matrix with exactly the same dimension as the input sequence. The input sequence and kernel are the same dimension in this work. This brings us to the final equation of the circular 2-dimensional convolution that will be used in this work:

$$y(m',n') = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x(m,n)\cdot h(m'-m,n'-n)$$

$$y(m,n) = x(m,n) \circledast h(m,n)$$

The circular convolution has the following property:

$$x(m,n) \circledast h(m,n) = \mathcal{F}^{-1}(\mathcal{F}(x(m,n))\cdot \mathcal{F}(h(m,n)))$$

This means the Fourier transform can be avoided by using circular convolution, as mentioned before.

The problem now is how to calculate this double sum. In this work, the 2-dimensional convolution is done by summing the results of a set of 1-dimensional convolutions. This is done by looking upon a matrix as a collection of vectors. Each row in the matrix is a vector: if x(m,n) is a matrix, than $x_m(n)$ is the m-th row in the matrix x. Now we can rewrite the above equation:

$$y_{m'}(n') = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x_m(n) \cdot h_{m'-m}(n'-n)$$

$$y_{m'}(n) = \sum_{m=0}^{M-1} x_m(n) \circledast h_{m'-m}(n)$$

$$y_{m'} = \sum_{m=0}^{M-1} y_m^{m'}$$

This shows one row of the 2-dimensional circular convolution can be calculated by summing the results of a series of 1-dimensional circular convolutions. The 1D circular convolution routine will calculate an entire row at once (all values of n) using one row from the input and one row from the kernel. The corresponding elements in all resulting rows will be added to form an entire row of the result.

Usually, during calculation the h-sequence is rotated to compute the result for consecutive values of N:

$$y_m^{m'}(n') = \sum_{n=0}^{N-1} x_m(n) \cdot h_{m'-m}(n'-n)$$

$$\begin{aligned}
y_m^{m'}(0) &= x_m(0) \cdot h_{m'-m}(0) + x_m(1) \cdot h_{m'-m}(N-1) + \cdots + x_m(N-1) \cdot h_{m'-m}(1) \\
y_m^{m'}(1) &= x_m(0) \cdot h_{m'-m}(1) + x_m(1) \cdot h_{m'-m}(0) + \cdots + x_m(N-1) \cdot h_{m'-m}(2) \\
&\vdots \\
y_m^{m'}(N-1) &= x_m(0) \cdot h_{m'-m}(N-1) + x_m(1) \cdot h_{m'-m}(N-2) + \cdots + x_m(N-1) \cdot h_{m'-m}(0)
\end{aligned}$$

In this case, because the input has to be converted from byte to words and therefore requires processing anyway, the input is rotated. This is possible if the terms in the equation above are sorted differently:

$$y_m^{m'}(0) = x_m(0)\cdot h_{m'-m}(0) + x_m(N-1)\cdot h_{m'-m}(1) + \cdots + x_m(1)\cdot h_{m'-m}(N-1)$$
$$y_m^{m'}(1) = x_m(1)\cdot h_{m'-m}(0) + x_m(0)\cdot h_{m'-m}(1) + \cdots + x_m(2)\cdot h_{m'-m}(N-1)$$
$$\vdots$$
$$y_m^{m'}(N-1) = x_m(N-1)\cdot h_{m'-m}(0) + x_m(N-2)\cdot h_{m'-m}(1) + \cdots + x_m(0)\cdot h_{m'-m}(N-1)$$

This shows the input is needed in reverse order. This will be taken care of as the input bytes are converted to words. Note that it saves one rotation if $y^{m'}{}_m(N-1)$ is calculated first, followed by $y^{m'}{}_m(N-2)$, and so on. This means that the input sequence will need to be rotated from right to left instead of from left to right.

4.3 MMX 2-D convolution routine

4.3.1 MMX 1D convolution routine

All the convolution routine needs to do is read a quadword from the rotating x-row and a quadword from the h-row and perform the multiply-add (PMADDWD) instruction. The result of this operation is added to an intermediate result, which is set to 0 at the beginning of the routine. Because one PMADDWD-instruction actually does two multiply-add instruction and yields two packed 32-bit results, a copy and shift operation would be necessary after each PMADDWD-instruction to add both results to the intermediate result. This is why instead of one intermediate result, two intermediate results in the form of one quadword are used. The two final intermediate results are added by a shift and copy operation. The following might make this more clear:

$x_m(n'-(n+3))$	$x_m(n'-(n+2))$	$x_m(n'-(n+1))$	$x_m(n'-n)$
$h_{m'-m}(n+3)$	$h_{m'-m}(n+2)$	$h_{m'-m}(n+1)$	$h_{m'-m}(n)$
PMADDWD:			
p(n+2)+p(n+3)		p(n)+p(n+1)	
p(2)+p(3)+p(6)+p(7)+…+p(n-2)+p(n-1)		p(0)+p(1)+p(4)+p(5)+…+p(n-4)+p(n-3)	
PADDD			
p(2)+p(3)+p(6)+p(7)+…+ p(n+2)+p(n+3)		p(0)+p(1)+p(4)+p(5)+…+p(n)+p(n+1)	

where $p(n) = x_m(n'-n)*h_{m'-m}(n)$

Note that if $h_{m'-m}$ and x_m are padded with zeros, the result is unaffected. This is done here, so there are always a multiple of 4 elements in $h_{m'-m}$ and x_m. The routine rotating x_m, however, must take this into account.

After the above procedure is repeated enough times, the two intermediate results need to be added as follows:

p(2)+p(3)+p(6)+p(7)+…+ p(N-2)+p(N-1)		p(0)+p(1)+p(4)+p(5)+…+p(N-4)+p(N-3)	
MOVQ:			
p(2)+…+p(N-1)	p(0)+…+p(N-3)		
PSRLQ:			
0	p(2)+…+p(N-1)	p(2)+…+p(N-1)	p(0)+…+p(N-3)
PADDD:			
p(2)+…+p(N-1)		p(0)+…+p(N-1)	
PSRAD:			
		$y^{m'}{}_m(n')$	

The PSRAD instruction is there to perform a division. Remember that the values in the h matrix are multiplied by a power of 2 which depends on the range of the floating-point h values.
Because the above process is not register-intensive and because it is easier to pair instructions from two different processes, the instructions above are intertwined with instructions calculating $y^{m'}{}_{m}(n'-1)$. The two results are combined in one quadword and can then be written to a buffer using only one MOVQ-instruction. Another advantage of this pairing is that memory access to the h-matrix is cut in half! The only price to pay for this is that there need to be two x input vectors, one rotated one word in respect to the other. The number of rotations remains the same, but an extra buffer is needed. This is usually not a problem, because kernel and hence input matrices must be rather small to avoid saturation of the result.
The result is a routine calculating $y^{m'}{}_{m}(n'-1)$ and $y^{m'}{}_{m}(n')$, requiring two x vector inputs, one h vector input, the number of iterations (which is N/4 if N is a multiple of 4, N/4+1 otherwise), and a location to store the quadword result.
There is of course a problem in the process encapsulating this routine if N is odd. Although it does not pose a problem to calculate any $y^{m'}{}_{m}(n')$ results, the last time the routine is called, there is no $y^{m'}{}_{m}(n'-1)$ to calculate. The next vector to be calculated does not only need a new x-vector (which would pose no problem) but also needs a new h-vector. Because of this, the 2-result routine cannot be used to calculate a result of the next row. That is why a routine is created calculating only one $y^{m'}{}_{m}$ result. This routine takes less time to complete than the routine calculating two results, but takes more than half the time it takes the 2-result routine to calculate two results. The 2-result routine is called as many times as possible, followed by the 1-result routine if necessary. The reason why an extra routine is written for the case N is odd, is that kernel sizes are often odd. The most common kernel sizes are 3x3, 6x6 and 9x9. Two out of three kernel sizes are odd, which justifies the extra routine.
Because of the way the PMADDWD-instruction operates, this routine will perform optimally with kernel sizes which are a multiples of 4 like 4x4, 8x8 and 12x12. This should be made clear to people generating kernels for this programs.

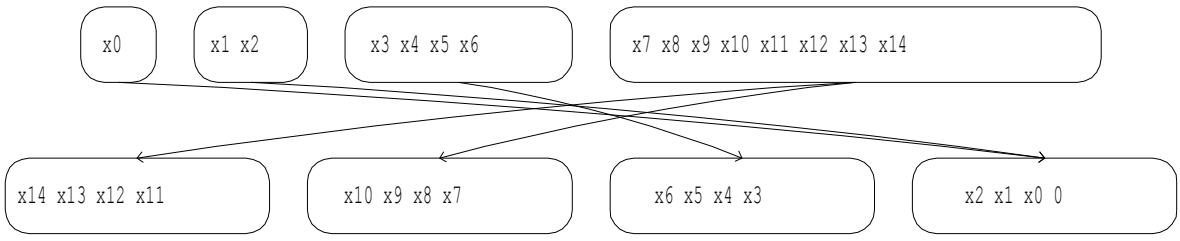
4.3.2 Preparing the input

4.3.2.1 Inverting the sequence

Not only does the input need to be transformed from bytes to words, they also have to be stored in reverse order. Reversing the sequence is not so difficult when the bytes are first converted to words, then converted to doublewords, rotated as doublewords and finally packed back to words. The following table presents an overview:

x7	x6	x5	x4	x3	x2	x1	x0
MOVQ							
x7	x6	x5	x4	x3	x2	x1	x0
PUNPCKLBW:				PUNPCKHBW:			
x3	x2	x1	x0	x7	x6	x5	x4
MOVQ:				MOVQ:			
x3	x2	x1	x0	x7	x6	x5	x4
PUNPCKLWD:		PUNPCKHWD:		PUNPCKLWD:		PUNPCKHWD:	
x1	x0	x3	x2	x5	x4	x7	x6
MOVQ:		MOVQ:		MOVQ:		MOVQ:	
x1	x0	x3	x2	x5	x4	x7	x6
PSLLQ:	PSRLQ:	PSLLQ:	PSRLQ:	PSLLQ:	PSRLQ:	PSLLQ:	PSRLQ:
x0	0	x2	0	x4	0	x6	0
PADDD:		PADDD:		PADDD:		PADDD:	
x0	x1	x2	x3	x4	x5	x6	x7
PACKSSDW:				PACKSSDW:			
x0	x1	x2	x3	x4	x5	x6	x7

Using this process as the input pointer increases and the output pointer decreases, solves the problem. That is, of course, if the size of the input is a multiple of 8. If it is not, similar routines were created to process a different number of values. These altered routines must be applied at the beginning of the conversion, because the first values read in end up in the last quadword in the result and this is where padding occurs. The following presentation of input and output might make things more clear:



This shows the padding problem must be handled at the beginning of the preparation stage. As necessary, a byte, word and doubleword are read, moved to an MMX register and shifted to the proper position for input to the left half of the routine above (generating only one quadword result instead of two).

Adding the now correctly positioned values in the MMX-registers provides a suitable doubleword input for the conversion routine. The following example explains how the situation presented above is handled:

- MOV al, [input] : move the byte x0 from input to al
- MOVD mm0, eax : move 0|0|0|0|0|0|x0 to mm0
- MOV ax,[input+1] : move x2|x1 to ax
- MOVD mm1,eax : move 0|0|0|0|0|x2|x1 to mm1
- PSRLQ mm0,8 : mm0 now contains 0|0|0|0|0|x0|0
- PSRLQ mm1,16 : mm1 now contains 0|0|0|x2|x1|0|0
- PADDB mm0,mm1 : mm0 now contains 0|0|0|x2|x1|x0|0

When handled by the conversion routine, the result will be 0|x0|x1|x2, which will result to x2|x1|x0|0 when written to memory using the MOVQ-instruction.

Similar routines were created in case the last quadword contains 2 and 1 values. The routine handling 4 input bytes does not need this shift-and-add operation, it simply reads in one doubleword and writes out one quadword.

The combination of routines handling 1, 2, 3, 4 and 8 input bytes can handle all possible input sizes.

4.3.2.2 Handling offsets

If the input is read from the U- or V-channel, there is an offset present in the values read from memory. This problem can be easily solved by making slightly altered input handling routines so that they subtract the offset right after unpacking the input. This is the easiest solution, because there are different routines for different padding situation. Depending on the padding, other constants must be used because no offset must be removed from the padding zeros.

The result is that there are in total 10 input routines: 5 routines not subtracting offset (1, 2, 3, 4 and 8 input bytes) and 5 routines subtracting offset.

4.3.3 Rotating the input buffer

The rotation process is extremely simple if the padding problem is not taken into account. Fortunately, padding only causes problems when handling the last quadword. There are 4 possibilities for the last quadword: it can contain 4, 3, 2 or 1 values. The routine processing all quadwords except the last one works as follows:

x(n-3)	x(n-2)	x(n-1)	x(n)	x(n-7)	x(n-6)	x(n-5)	x(n-4)
PSRLQ:				PSLLQ:			
0	x(n-3)	x(n-2)	x(n-1)	x(n-4)	0	0	0
PADDW:							
x(n-4)		x(n-3)		x(n-2)		x(n-1)	

In the loop, only the higher quadword is read from memory by the routine. The lower quadword is kept in register the previous time the loop was executed. This means that to enter the loop, the first quadword needs to be read from memory before entering the loop. At this stage, the first quadword is copied and then shifted so the first word is stored in a separate register in the form x (n') 0 0 0. This word will be needed when processing the last quadword.

The case where the last quadword contains two values will now be explained. The other cases are handled by a similar process.

x(n')	0	0	0	0	0	x(n'+1)	x(n'+2)
PSRLQ:				PSRLQ:			
0	0	x(n')	0	0	0	0	x(n'+1)
PADDW:							
0		0		x(n')		x(n'+1)	

The only difference when the number of values is different is how much x(n') is shifted. It is obvious that this process is much faster than moving every word separately using the normal integer registers.

4.3.4 2D-convolution using 1D-convolution

Assuming the size of the kernel h is M rows by N columns, a buffer is reserved for the h-matrix providing storage space for M lines containing the lowest multiple of 4 higher than N words. This ensures the h-matrix is quad-aligned.

The input h-matrix consists of floating-point numbers, which have to be converted to signed integer words for storage in the h-buffer. To do this, the floating-point numbers have to be multiplied by a constant C_M. This constant C_M must be a power of 2 and must be as high as possible for the purpose of accuracy, but should not be too high because the larger C_M, the higher the probability the convolution will saturate because a lot of large numbers are added together. Saturation also occurs when the size of the kernel is too big. It can be said that if the size of the kernel is large, C_M should be decreased to avoid saturation, but this is at the cost of accuracy. In order to be able to experiment with this trade-off, C_M can be determined by the user of the program.

After all values in h are multiplied by C_M, truncated and stored in the h-buffer, the x-buffer is created. The x-buffer is exactly the same size as the h-buffer and is filled from a greyscale DIB in memory by the input preparation routine discussed in 3.3.2. This is done by letting the input preparation routine handle a section of each line of the DIB and storing the result in consecutive lines in the x-buffer. Now there are two matrices in memory containing input and kernel. The rows of each buffer are quad-aligned and both buffers contain integer words.

There are still 4 buffers needed to store rotated input and intermediate results. They are the following:

- x1-buffer: contains one row from the x-buffer. The rotation routine stores its results in this buffer.
- x2-buffer: same as x1-buffer, except that x2 is rotated once more than x1.
- YM-buffer: stores M by N 32-bit results from the convolution routine.
- YR-buffer: contains N 32-bit values and is used to store one row of the result before packing.

The algorithm to convolute the h- and x-buffer is presented below as C-style pseudo-code:

```
for (m' = 0; m' < M; m'++){
   calculate y^m' and store result in YR-buffer:{
      for (m = 0; m < M; m++){
         calculate the convolution of the m-th row of the x-buffer
         with the (m'-m)-th row of the h-buffer;
         store result in the m-th row of the YM-buffer:{
            copy the m-th row of the x-buffer to the x1- and x2-buffer.
            rotate the x2-buffer.
            N2 = N-1.
            if (N is odd){
               convolute x1 with h_m'-m; store in YM[m, N2].
               rotate x1.
               rotate x2.
               decrement N2.
            }
            for (n = N2; n >= 0; n -= 2){
               convolute x1 and x2 with h_m'-m; store in YM[m, n and n-1].
               rotate x1.
               rotate x2.
            }
         }
         Make the sum of each column in the YM-buffer
         and store the N results in the YR-buffer.
      }
   }
   pack values in YR-buffer from doublewords to bytes
   and store in one line of target DIB.
}
```

4.3.5 Output processing

As can be seen in the pseudo-code above, two steps have not been explained yet. The first is summing the values in the YM-buffer. This is done by reading all the first quadwords of each row, adding them doubleword-by-doubleword (PADDD) and then storing the result to the first quadword in the YR-buffer. Then all the second quadwords are added, and so on. This routine is nothing but a MOVE and PADDD instruction in a double loop.
The second routine is the packing and storing of the values in the YM-buffer. This routine is very similar to the packing algorithms discussed in paragraphs 2.5.4 and 2.6.4. This will not be discussed further at this point.

4.4 Alignment issues

Because all intermediate buffers are quadword-aligned, there are no problems with alignment anywhere in the routine, except when reading the input from the Windows DIB and storing to the target Windows DIB. Because the size of the kernel and hence the input is relatively small, considering alignment is not worth the effort. Aligning pointers as was done with the YUV-routines would not pay off here, because more time would be lost calculating the optimal reading strategy than is gained by proper reading.

5 Integration of MMX-code

The routines discussed in chapter 2 and 3 are put in an object called MMX_2D_Proc as static functions. Another object called 2D_Proc was created and contains the same static functions as MMX_2D_Proc, except that all functions are implemented without using MMX-technology. They behave in exactly the same way, but are much slower. Both objects are put in a Win32 static library. This library is then used to replace the image processing routines in the library ImageObject available from [1]. After adding some extra functionality to the ImageObject-library, this library is used inside the application framework generated by Visual C++.
Some extra code was created to read in and organise files generated by Matlab and present them in a menu. This is standard C-code not worthy of our attention here.

6 Conclusions

MMX-technology can definitely speed up application speed. The gain is not just a matter of percentages but factors: one can expect the MMX-routine to be about 10 to 20 times faster than code using the traditional integer instruction.

A second advantage is the DSP-like features of MMX-technology. In particular, the saturation feature is greatly appreciated in DSP-applications. Extra code could provide saturation using traditional integer technology, but this takes extra time, while saturating does not require extra time when using MMX-technology. Also, the packing and unpacking instructions solve a problem that usually takes a lot of shifting and adding. A relatively complex process is handled in one instruction. This entire work is based on use of the multiply-add instruction, which is of course one of the main assets of MMX-technology.

The main problem with MMX-technology is that data alignment is very important. If the data structures handled can be defined by the author of the code, this poses no problem. If the data structures are already defined and are not quad-aligned, some of the performance gain is lost to slower reading of the data-structure. If the size of the data-structure is large, some techniques can be used to make sure most of the data read in is aligned. When the amount of data is small, alignment mismatches are practically unavoidable. Although alignment is important, it does not mean that improper alignment causes the MMX-technology to be slower than traditional integer operations. It just means the code slows down about 20%, but it is still much faster than traditional technology.

Another disadvantage of MMX-technology at the moment is that MMX-instructions are not used by any compiler, which means that to use MMX-technology, one needs to program in assembler. This is a labour-intensive way of programming. The simplest way to handle this problem is spot-optimising. This means an interesting instruction is given a function equivalent and then used as a function in the rest of the process. The result is that a lot of memory copying occurs each time that the instruction is needed. To avoid this, one can of course write function that handle more complex problems but that means more assembler code and more time.

This problem is partly being solved by free libraries released by Intel, but these libraries have three disadvantages. The first disadvantage is that the source code is unavailable, so if one needs a slightly altered version of a certain function, that function cannot be altered.

The second disadvantage is that there are libraries for the most common signal processing techniques, but this does not help if the problem at hand needs a custom approach.
The last and third disadvantage is that these libraries waste a lot of memory. To avoid alignment problems, new aligned data formats are created. Functions are provided to convert existing data formats to the new ones. Processing is then done on the new format and the data converted back to the old format. This approach guarantees small and fast code, but requires a lot of memory. This might be a problem in some situations where little memory is available.
Until compilers are made that recognise code that can be optimised for MMX-technology, and include code for both MMX- and non-MMX-processors, assembler routines must be written for those custom problems. Such a compiler will undoubtedly produce code using a lot of memory and will probably not find the best solution. Applications that are speed- and memory-critical will always have to be constructed using assembler. This is not only true for MMX-technology, but also for traditional programming. Therefore, it will be important to master assembly programming for some time to come.
In conclusion, it can be said that using MMX-technology produces faster code and better results, but that as more performance gain is wanted, more work is required.

7 Appendices

7.1 Appendix A: Optimum coefficients tests

7.1.1 Measuring Method

To calculate the Mean Euclidean Distance or MED of the conversion from RGB to YUV and back to RGB, buffers were created to contain 256*256 pixels for the input RGB, output Y, U, V and RGB signals. There are in total 256^3 pixel possibilities, but it would take to much memory to create buffers to hold all these pixels three times. That is why a buffer of 256^2 pixels is used. This buffer is large enough to hold all possibilities for green and blue. The red component is the same for all elements in the buffer. This buffer is converted to YUV using the MMX-routine and converted from YUV back to the second RGB-buffer. The values in both buffers are compared. If the original pixel was R=0, G=1 and B=0 and the result would be R'=0, G'=0 and B'=0, then the distance would be $\sqrt{(R'-R)^2+(G'-G)^2+(B'-B)^2}=1$. The distance is divided into categories according to the integer values under the square root (X'-X). Only certain values of the distance are possible, depending on how much times a certain integer appears in the (X'-X) values. The following table displays all possible cases and the resulting distance:

(X'-X)	distance	index
3 * 0	0.0	0
1 * 1	1.0	1
2 * 1	1.4	2
3 * 1	1.7	3
1 * 2	2.0	4
2 * 2	2.8	5
1 * 3	3.0	6
3 * 2	3.4	7
2 * 3	4.2	8
3 * 3	5.1	9

An array of integers is used to store how many times a certain case has occurred. If a certain case is detected, the integer with the corresponding index is incremented.

This process is repeated for each pixel in the buffer.

The above is repeated for each R-value, after which all possible pixels have been tested. The MED is then the sum of the integers in the array multiplied with their corresponding distance value divided by the total number of values, which is 256^3. The results are printed to a text-file.

In the main testing programs, the constants are kept in arrays and pointers to them are given to the conversion routines. Changing the values and calculating the MED with each set of constants yields MED in function of constants.

7.1.2 Function calculating MED

```
double CalcDist(FILE *f){
   /*
   Making a buffer to hold 3 times 256*256 pixels, 3 bytes a pixel:
      RGB_in : 1*3*256*256
      Y,U,V  : 3*1*256*256
      RGB_out: 1*3*256*256
   */
   const long NR_PIX  = (1<<16);
   const long BUFSIZE = 3*(3*NR_PIX);
   unsigned char RGB_in[BUFSIZE];
   unsigned char *Y, *U, *V, *RGB_out;
   Y = &RGB_in[(3*BUFSIZE/9)-1];
   U = &RGB_in[(4*BUFSIZE/9)-1];
   V = &RGB_in[(5*BUFSIZE/9)-1];
   RGB_out = &RGB_in[(2*BUFSIZE/3)-1];

   double distance;
   int dist[10] = {0,0,0,0,0,0,0,0,0,0};
   double med;

   //fill G and B values of the input buffer:
   for(int g=0; g < 256; g++)
      for(int b=0; b < 256; b++){
         RGB_in[(256*g+b)*3+1] = g;
         RGB_in[(256*g+b)*3+2] = b;
      }

   for(int r = 0; r < 256; r++){
      for(int pix=0; pix < NR_PIX; pix++)
         RGB_in[3*pix] = r;
      rgb2yuv((void *)RGB_in, (void *)Y, (void *)U, (void *)V, NR_PIX);
      yuv2rgb((void *)Y, (void *)U, (void *)V, (void *)RGB_out, NR_PIX);

      for(int p=0; p < NR_PIX; p++){
         distance  = abs(RGB_in[p*3]   - RGB_out[p*3])   << 1;
         distance += abs(RGB_in[p*3+1] - RGB_out[p*3+1]) << 1;
         distance += abs(RGB_in[p*3+2] - RGB_out[p*3+2]) << 1;
         distance = sqrt(distance);
         /*
            distance possibilities:
               3 * 0 : 0
               1 * 1 : 1
               2 * 1 : 1.4
               3 * 1 : 1.7
               1 * 2 : 2

               2 * 2 : 2.8
               1 * 3 : 3.0
               3 * 2 : 3.4
               2 * 3 : 4.2
               3 * 3 : 5.1
         */
         if(distance < 2.4)
            if(distance < 1.5)
               if(distance < 1.2)
                  if(distance < 0.5) dist[0]++;
                  else dist[1]++;
               else dist[2]++;
            else
               if(distance <1.9) dist[3]++;
               else dist[4]++;
         else
            if(distance < 3.8)
               if(distance < 3.2)
                  if(distance < 2.9) dist[5]++;
                  else dist[6]++;
               else dist[7]++;
            else
```

```
                if(distance <4.7) dist[8]++;
                else dist[9]++;
      }
   }
   const double N = 16777216;
   med  = dist[1]/N + (dist[2]/N)*sqrt(2.0) + (dist[3]/N)*sqrt(3.0) + (dist[4]/N)*2.0;
   med += (dist[5]/N)*sqrt(8.0) + (dist[6]/N)*3.0 + (dist[7]/N)*sqrt(12.0);
   med += (dist[8]/N)*sqrt(18.0) + (dist[9]/N)*sqrt(27.0);
   fprintf(f,"\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d", CRY, CBY, CGV, CBV,
         CUG,CVG);
   fprintf(f,"\ndistance:\n");
   fprintf(f,"   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1\n");
   fprintf(f,"  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)\n"
         );
   for(int d=0; d < 9; d++)
      fprintf(f,"%7d ",dist[d]);
   fprintf(f,"%7d\n",dist[9]);
   fprintf(f,"MED (Mean Euclidean Distance): %f\n",med);
   return(med);
}
```

7.1.3 Function calls in main program

```
__int16 CRY_f, CBY_f, CGV_f, CBV_f, CUG_f, CVG_f;
   double med_f, med_c;
   FILE *fCD;
   fCD = fopen("Distance.txt","w");
   CRY=9797; CBY=3736; CGV = -13719; CBV=-2665;
      CUG=-5637; CVG=-11700;
         InitVars();
         printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
         med_f = CalcDist(fCD);
      CUG=-5638; CVG=-11700;
         InitVars();
         printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
         med_c = CalcDist(fCD);
         if(med_c < med_f){
            med_f = med_c;
            CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
         }
      CUG=-5637; CVG=-11701;
         InitVars();
         printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
         med_c = CalcDist(fCD);
         if(med_c < med_f){
            med_f = med_c;
            CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
         }
      CUG=-5638; CVG=-11701;
         InitVars();
         printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
         med_c = CalcDist(fCD);
         if(med_c < med_f){
            med_f = med_c;
            CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
         }
   CRY=9797; CBY=3736; CGV = -13720; CBV=-2664;
      CUG=-5637; CVG=-11700;
         InitVars();
         printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
         med_c = CalcDist(fCD);
         if(med_c < med_f){
            med_f = med_c;
            CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
         }
      CUG=-5638; CVG=-11700;
         InitVars();
         printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
         med_c = CalcDist(fCD);
         if(med_c < med_f){
            med_f = med_c;
            CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
         }
```

```
   CUG=-5637; CVG=-11701;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
   CUG=-5638; CVG=-11701;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
CRY=9798; CBY=3735; CGV = -13719; CBV=-2665;
   CUG=-5637; CVG=-11700;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
   CUG=-5638; CVG=-11700;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
   CUG=-5637; CVG=-11701;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
   CUG=-5638; CVG=-11701;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
CRY=9798; CBY=3735; CGV = -13720; CBV=-2664;
   CUG=-5637; CVG=-11700;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
   CUG=-5638; CVG=-11700;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
   CUG=-5637; CVG=-11701;
      InitVars();
      printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
            CUG,CVG);
      med_c = CalcDist(fCD);
      if(med_c < med_f){
         med_f = med_c;
         CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
      }
```

```
      CUG=-5638; CVG=-11701;
         InitVars();
         printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
         med_c = CalcDist(fCD);
         if(med_c < med_f){
            med_f = med_c;
            CRY_f = CRY; CBY_f = CBY; CGV_f = CGV; CBV_f = CBV; CUG_f = CUG; CVG_f = CVG;
         }
   fprintf(fCD,"\n\nminimum MED = %f for constants:",med_f);
   printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY,CBY,CGV,CBV,
               CUG,CVG);
   printf("\n\nminimum MED = %f for constants:",med_f);
   printf("\nCRY = %d; CBY = %d; CGV = %d; CBV = %d; CUG = %d; CVG = %d",CRY_f,CBY_f,CGV_f,
          CBV_f,CUG_f,CVG_f);
```

7.1.4 Output

```
CRY = 9797; CBY = 3736; CGV = -13719; CBV = -2665; CUG = -5637; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24692       0 1343138 9681974 3838244 1888796       0       0
MED (Mean Euclidean Distance): 2.870778

CRY = 9797; CBY = 3736; CGV = -13719; CBV = -2665; CUG = -5638; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24781       0 1344241 9680701 3835979 1891142       0       0
MED (Mean Euclidean Distance): 2.870782

CRY = 9797; CBY = 3736; CGV = -13719; CBV = -2665; CUG = -5637; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24775       0 1343549 9680953 3836148 1891419       0       0
MED (Mean Euclidean Distance): 2.870829

CRY = 9797; CBY = 3736; CGV = -13719; CBV = -2665; CUG = -5638; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24882       0 1345884 9676574 3835373 1894131       0       0
MED (Mean Euclidean Distance): 2.870799

CRY = 9797; CBY = 3736; CGV = -13720; CBV = -2664; CUG = -5637; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24689       0 1343465 9681623 3837513 1889554       0       0
MED (Mean Euclidean Distance): 2.870784

CRY = 9797; CBY = 3736; CGV = -13720; CBV = -2664; CUG = -5638; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24776       0 1344399 9680689 3835089 1891891       0       0
MED (Mean Euclidean Distance): 2.870794

CRY = 9797; CBY = 3736; CGV = -13720; CBV = -2664; CUG = -5637; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24772       0 1343795 9680683 3835417 1892177       0       0
MED (Mean Euclidean Distance): 2.870838

CRY = 9797; CBY = 3736; CGV = -13720; CBV = -2664; CUG = -5638; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24878       0 1346035 9676183 3834868 1894880       0       0
MED (Mean Euclidean Distance): 2.870816

CRY = 9798; CBY = 3735; CGV = -13719; CBV = -2665; CUG = -5637; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
   372       0   24744       0 1342708 9683407 3838651 1887334       0       0
MED (Mean Euclidean Distance): 2.870744
```

```
CRY = 9798; CBY = 3735; CGV = -13719; CBV = -2665; CUG = -5638; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
    372       0   24833       0 1343811 9682134 3836386 1889680       0       0
MED (Mean Euclidean Distance): 2.870748


CRY = 9798; CBY = 3735; CGV = -13719; CBV = -2665; CUG = -5637; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
    372       0   24827       0 1343119 9682386 3836555 1889957       0       0
MED (Mean Euclidean Distance): 2.870795

CRY = 9798; CBY = 3735; CGV = -13719; CBV = -2665; CUG = -5638; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
    372       0   24934       0 1345454 9678007 3835780 1892669       0       0
MED (Mean Euclidean Distance): 2.870765

CRY = 9798; CBY = 3735; CGV = -13720; CBV = -2664; CUG = -5637; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
    372       0   24741       0 1343035 9683056 3837920 1888092       0       0
MED (Mean Euclidean Distance): 2.870749

CRY = 9798; CBY = 3735; CGV = -13720; CBV = -2664; CUG = -5638; CVG = -11700
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
    372       0   24828       0 1343969 9682122 3835496 1890429       0       0
MED (Mean Euclidean Distance): 2.870760

CRY = 9798; CBY = 3735; CGV = -13720; CBV = -2664; CUG = -5637; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
    372       0   24824       0 1343365 9682116 3835824 1890715       0       0
MED (Mean Euclidean Distance): 2.870804

CRY = 9798; CBY = 3735; CGV = -13720; CBV = -2664; CUG = -5638; CVG = -11701
distance:
   0.0     1.0     1.4     1.7     2.0     2.8     3.0     3.4     4.2     5.1
  (3*0)   (1*1)   (2*1)   (3*1)   (1*2)   (2*2)   (1*3)   (3*2)   (2*3)   (3*3)
    372       0   24930       0 1345605 9677616 3835275 1893418       0       0
MED (Mean Euclidean Distance): 2.870781


minimum MED = 2.870744 for constants:
CRY = 9798; CBY = 3735; CGV = -13719; CBV = -2665; CUG = -5637; CVG = -11700
```

7.2 Appendix B: Data alignment and speed tests

7.2.1 Measuring techniques

Windows has a timer for every process running in the Windows environment. This timer shows how many milliseconds of processing time were allocated to the process since it started. It is important to use this timer instead of the general timer, because Windows regularly interrupts the processes to check if the start button was pressed, etc. This timer, however, does not function adequately in Windows 95/98. Running a second process in the background yields lower speed results for the tested routine, showing that the time indicated by the timer is not the time the process has been running, but the time Windows has been running. The problem is obviously known by Microsoft, because Visual C++ refuses to profile an application under Windows 95/98, but profiles perfectly under Windows NT. The speed tests were also unaffected when running a second process, meaning time is measured accurately only when running Windows NT. This is very important and must be considered if one wishes to test the speed of these routines on other machines.

Because it takes much less than a millisecond (under 1μs!) to execute any of the routines once, the routines are executed a number of times and the measured time is then divided by the number of times they were executed. To allow easy comparison of the execution time in milliseconds, the number of times a routine is executed depends on how many pixels are processed in one run. For instance, if the routine processing 8 pixels at once is executed N times, than the routine processing 4 pixels at once is executed 2*N times. This way, the result is the number of milliseconds needed to process a fixed number of pixels using a routine. The time needed to sustain the loop (incrementing the counter and testing for end-of-loop) are subtracted from the measured time, so the time measured is the time in the conversion routines only.

The input pointer is shifted one byte every time to test the effect of alignment on speed. There are two possibilities to alter alignment. The first is to keep alignment fixed and test the speed of the different routines, while the second is to change alignment while keeping the used routine constant. Both approaches are explored here.

The values obtained are transformed to million pixels per second and time for one execution in nanoseconds for reading convenience. All results are written to a text-file. Only the case where alignment changes in the inner loop is used to calculate the times for single execution, because it best represents the situation of preparing for alignment, were these values are used to determine optimal reading strategy.

The syuv2rgb and srgb2yuv routines mentioned are routines written in normal C-language, not using MMX-technology.

The values displayed are those obtained on a Pentium MMX 166MHz computer with 64MB RAM running Windows NT.

7.2.2 Function calculating speed

```
void TestSpeed(FILE *f){
   //variables to store number of processor ticks:
   clock_t start, finish, funcstart, funcend;
   funcstart = clock();
   long t_loop[9];
   long t1_rgb2yuv[8][6];
   long t1_yuv2rgb[8][6];
   long t2_rgb2yuv[8][6];
   long t2_yuv2rgb[8][6];

   //number of iterations:
   const long NR_IT = 1048576;

   //create Buffers
   unsigned char rgb_in[3*8+16];
   unsigned char y[8+16];
   unsigned char u[8+16];
   unsigned char v[8+16];
   unsigned char rgb_out[3*8+16];

   //create and align pointers to buffers
   unsigned char *RGB_in, *Y, *U, *V, *RGB_out;
   __int64 temp = (__int64) &rgb_in[0];
   temp /= 8;
   RGB_in = (unsigned char *) ((temp+1)*8);

   temp = (__int64) &y[0];
   temp /= 8;
   Y = (unsigned char *) ((temp+1)*8);

   temp = (__int64) &u[0];
   temp /= 8;
   U = (unsigned char *) ((temp+1)*8);

   temp = (__int64) &v[0];
   temp /= 8;
   V = (unsigned char *) ((temp+1)*8);

   temp = (__int64) &rgb_out[0];
   temp /= 8;
   RGB_out = (unsigned char *) ((temp+1)*8);

   //variables to be used in loops:
   long p;
   int  i,j, nr_pix, mul;

   //determining time required by for-loop
   for(i=0;i<9;i++){
      start = clock();
      for(p=0; p < NR_IT*i;p++);
      finish = clock();
      t_loop[i] = (long)(finish - start);
   }
```

```
for(i = 0; i<8; i++){
   for(j = 0; j < 5; j++){
      nr_pix = j;
      if(j == 3) nr_pix = 4;
      if(j == 4) nr_pix = 8;

      if(nr_pix==0) mul=1;
      else     mul = 8 / nr_pix;

      start = clock();
      for(p=0; p < (NR_IT*mul);p++)
         rgb2yuv((void *)RGB_in, (void *)Y, (void *)U, (void *)V, nr_pix);
      finish = clock();
      t1_rgb2yuv[i][j] = (long)(finish - start) - t_loop[mul];

      start = clock();
      for(p=0; p < (NR_IT*mul);p++)
         yuv2rgb((void *)Y, (void *)U, (void *)V, (void *)RGB_out, nr_pix);
      finish = clock();
      t1_yuv2rgb[i][j] = (long)(finish - start) - t_loop[mul];
      printf("²");
   }
   start = clock();
   for(p=0; p < NR_IT;p++)
      srgb2yuv((void *)RGB_in, (void *)Y, (void *)U, (void *)V, 8);
   finish = clock();
   t1_rgb2yuv[i][5] = (long)(finish - start) - t_loop[1];

   start = clock();
   for(p=0; p < NR_IT;p++)
      syuv2rgb((void *)Y, (void *)U, (void *)V, (void *)RGB_out, nr_pix);
   finish = clock();
   t1_yuv2rgb[i][5] = (long)(finish - start) - t_loop[1];

   RGB_in++;
   Y++;
   U++;
   V++;
   RGB_out++;
   printf("²");
}
printf("\n");
fprintf(f,"\nmaximum number of clock ticks for loop   : %d\n",t_loop[8]);
fprintf(f,"number of clock ticks per second : %d\n",CLOCKS_PER_SEC);

fprintf(f,"\nALTER ALIGNMENT IN OUTER LOOP, PIXEL COUNT IN INNER LOOP:\n");
fprintf(f,"\nnumber of clock ticks to execute rgb2yuv enough times\n");
fprintf(f,"to process %d pixels(w/o loop time):\n",NR_IT*8);
fprintf(f," p %% 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv\n");
for(i = 0; i <8; i++){
   fprintf(f,"%4d    ",i);
   for(j = 0; j < 6; j++)
      fprintf(f," %8ld", t1_rgb2yuv[i][j]);
   fprintf(f,"\n");
}
fprintf(f,"\nnumber of clock ticks to execute yuv2rgb enough times\n");
fprintf(f,"to process %d pixels(w/o loop time):\n",NR_IT*8);
fprintf(f," p %% 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb\n");
for(i = 0; i <8; i++){
   fprintf(f,"%4d    ",i);
   for(j = 0; j < 6; j++)
      fprintf(f," %8ld", t1_yuv2rgb[i][j]);
   fprintf(f,"\n");
}

RGB_in -= 8;
Y -= 8;
U -= 8;
V -= 8;
RGB_out -= 8;

for(j = 0; j < 5; j++){
   nr_pix = j;
   if(j == 3) nr_pix = 4;
   if(j == 4) nr_pix = 8;

   if(nr_pix==0) mul=1;
   else     mul = 8 / nr_pix;
```

```
      for(i = 0; i<8; i++){
         start = clock();
         for(p=0; p < (NR_IT*mul);p++)
            rgb2yuv((void *)RGB_in, (void *)Y, (void *)U, (void *)V, nr_pix);
         finish = clock();
         t2_rgb2yuv[i][j] = (long)(finish - start) - t_loop[mul];

         start = clock();
         for(p=0; p < (NR_IT*mul);p++)
            yuv2rgb((void *)Y, (void *)U, (void *)V, (void *)RGB_out, nr_pix);
         finish = clock();
         t2_yuv2rgb[i][j] = (long)(finish - start) - t_loop[mul];
         RGB_in++;
         Y++;
         U++;
         V++;
         RGB_out++;
         printf("²");
      }
      RGB_in -= 8;
      Y -= 8;
      U -= 8;
      V -= 8;
      RGB_out -= 8;
   }
   for(i = 0; i<8; i++){
      start = clock();
      for(p=0; p < NR_IT;p++)
         srgb2yuv((void *)RGB_in, (void *)Y, (void *)U, (void *)V, 8);
      finish = clock();
      t2_rgb2yuv[i][5] = (long)(finish - start) - t_loop[1];

      start = clock();
      for(p=0; p < NR_IT;p++)
         syuv2rgb((void *)Y, (void *)U, (void *)V, (void *)RGB_out, 8);
      finish = clock();
      t2_yuv2rgb[i][5] = (long)(finish - start) - t_loop[1];
      RGB_in++;
      Y++;
      U++;
      V++;
      RGB_out++;
          printf("²");
   }
   printf("\n");
   RGB_in -= 8;
   Y -= 8;
   U -= 8;
   V -= 8;
   RGB_out -= 8;

   fprintf(f,"\nALTER PIXEL COUNT IN OUTER LOOP, ALIGNMENT IN INNER LOOP:\n");
   fprintf(f,"\nnumber of clock ticks to execute rgb2yuv enough times\n");
   fprintf(f,"to process %d pixels(w/o loop time):\n",NR_IT*8);
   fprintf(f," p %% 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 0; j < 6; j++)
         fprintf(f," %8ld", t2_rgb2yuv[i][j]);
      fprintf(f,"\n");
   }
   fprintf(f,"\nnumber of clock ticks to execute yuv2rgb enough times\n");
   fprintf(f,"to process %d pixels(w/o loop time):\n",NR_IT*8);
   fprintf(f," p %% 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 0; j < 6; j++)
         fprintf(f," %8ld", t2_yuv2rgb[i][j]);
      fprintf(f,"\n");
   }

   //display test results in pixels/second

   const double factor_v = 8.0 * double(NR_IT) * (double(CLOCKS_PER_SEC) / 1.0E6);

   fprintf(f,"\nALTER ALIGNMENT IN OUTER LOOP, PIXEL COUNT IN INNER LOOP:\n");
   fprintf(f,"\nspeed of rgb2yuv in million pixels/second\n");
   fprintf(f," p %% 8    1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 1; j < 6; j++)
         fprintf(f," %8.4f", (factor_v / (double)t1_rgb2yuv[i][j]));
      fprintf(f,"\n");
   }
```

```
   fprintf(f,"\nspeed of yuv2rgb in million pixels/second\n");
   fprintf(f," p %% 8    1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 1; j < 6; j++)
         fprintf(f," %8.4f", (factor_v / (double)t1_yuv2rgb[i][j]));
      fprintf(f,"\n");
   }
   fprintf(f,"\nALTER PIXEL COUNT IN OUTER LOOP, ALIGNMENT IN INNER LOOP:\n");
   fprintf(f,"\nspeed of rgb2yuv in million pixels/second\n");
   fprintf(f," p %% 8    1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 1; j < 6; j++)
         fprintf(f," %8.4f", (factor_v / (double)t2_rgb2yuv[i][j]));
      fprintf(f,"\n");
   }

   fprintf(f,"\nspeed of yuv2rgb in million pixels/second\n");
   fprintf(f," p %% 8    1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 1; j < 6; j++)
         fprintf(f," %8.4f", (factor_v / (double)t2_yuv2rgb[i][j]));
      fprintf(f,"\n");
   }

   //display time needed for 1 execution

   const double factor_t = ((1.0E9 / CLOCKS_PER_SEC) / NR_IT) / 8.0;

   fprintf(f,"\nALTER PIXEL COUNT IN OUTER LOOP, ALIGNMENT IN INNER LOOP:\n");
   fprintf(f,"\ntime needed to execute rgb2yuv once in nanoseconds\n");
   fprintf(f," p %% 8    1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 1; j < 6; j++){
         nr_pix = j;
         if(j == 3) nr_pix = 4;
         if(j == 4) nr_pix = 8;
         if(j == 5) nr_pix = 1;
         fprintf(f," %8.1f", t2_rgb2yuv[i][j] * nr_pix * factor_t);
          }
      fprintf(f,"\n");
   }

   fprintf(f,"\ntime needed to execute yuv2rgb once in nanoseconds\n");
   fprintf(f," p %% 8    1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb\n");
   for(i = 0; i <8; i++){
      fprintf(f,"%4d    ",i);
      for(j = 1; j < 6; j++){
         nr_pix = j;
         if(j == 3) nr_pix = 4;
         if(j == 4) nr_pix = 8;
         if(j == 5) nr_pix = 1;
         fprintf(f," %8.1f", (t2_yuv2rgb[i][j] * nr_pix * factor_t));
          }
      fprintf(f,"\n");
   }

   long lcctime = 0;
   for(i = 0; i <8; i++){
      for(j = 0; j < 6; j++){
         lcctime += t1_rgb2yuv[i][j];
         lcctime += t2_rgb2yuv[i][j];
         lcctime += t1_yuv2rgb[i][j];
         lcctime += t2_yuv2rgb[i][j];
      }
   }
   double cctime = (double)lcctime / (double)CLOCKS_PER_SEC;

   funcend = clock();
   double ftime = (double)(funcend - funcstart) / (double)CLOCKS_PER_SEC;
   double ccpct = cctime/ftime*100;
   fprintf(f,"%.3f seconds in TestSpeed,\n", ftime);
   fprintf(f,"of which %.3f seconds in color space conversion routines (%.2f%%)", cctime, ccpct);
}
```

7.2.3 Output

```
maximum number of clock ticks for loop   : 0
number of clock ticks per second : 1000

ALTER ALIGNMENT IN OUTER LOOP, PIXEL COUNT IN INNER LOOP:

number of clock ticks to execute rgb2yuv enough times
to process 8388608 pixels(w/o loop time):
 p % 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv
   0          260     3985     1992     1092      681     8562
   1          251     4116     2053     1442      891     8583
   2          250     4116     2063     1442      901     8613
   3          250     4116     2683     1442      902     8572
   4          250     3976     2053     1132      901     8573
   5          250     4116     2053     1432      892     8572
   6          240     4125     2053     1442      882     8602
   7          250     4116     2694     1432      891     8572

number of clock ticks to execute yuv2rgb enough times
to process 8388608 pixels(w/o loop time):
 p % 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb
   0          261     3616     1973     1141      842     7170
   1          260     3595     2444     1452      951     7250
   2          261     3595     2443     1443      951     7260
   3          251     4116     2634     1442      951     7171
   4          260     3605     1963     1252      951     7260
   5          261     3595     2443     1452      951     7251
   6          261     3606     2433     1442      951     7191
   7          260     4116     2634     1442      952     7210

ALTER PIXEL COUNT IN OUTER LOOP, ALIGNMENT IN INNER LOOP:

number of clock ticks to execute rgb2yuv enough times
to process 8388608 pixels(w/o loop time):
 p % 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv
   0          251     3996     1993     1091      681     8573
   1          260     4106     2063     1462      901     8562
   2          260     4116     2053     1472      892     8612
   3          260     4116     2684     1462      901     8573
   4          261     3976     2053     1121      891     8572
   5          260     4126     2063     1472      902     8582
   6          250     4106     2053     1452      891     8612
   7          261     4116     2694     1472      901     8583

number of clock ticks to execute yuv2rgb enough times
to process 8388608 pixels(w/o loop time):
 p % 8   overhead  1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb
   0          260     3605     1963     1142      842     7170
   1          251     3605     2443     1452      951     7251
   2          261     3595     2444     1442      951     7260
   3          250     4116     2633     1453      952     7170
   4          250     3595     1963     1252      951     7251
   5          261     3605     2444     1442      951     7241
   6          260     3605     2443     1452      952     7180
   7          250     4106     2634     1452      941     7210

ALTER ALIGNMENT IN OUTER LOOP, PIXEL COUNT IN INNER LOOP:

speed of rgb2yuv in million pixels/second
 p % 8    1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv
   0       2.1050   4.2111   7.6819  12.3181   0.9797
   1       2.0380   4.0860   5.8173   9.4148   0.9774
   2       2.0380   4.0662   5.8173   9.3103   0.9739
   3       2.0380   3.1266   5.8173   9.3000   0.9786
   4       2.1098   4.0860   7.4104   9.3103   0.9785
   5       2.0380   4.0860   5.8580   9.4043   0.9786
   6       2.0336   4.0860   5.8173   9.5109   0.9752
   7       2.0380   3.1138   5.8580   9.4148   0.9786

speed of yuv2rgb in million pixels/second
 p % 8    1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb
   0       2.3199   4.2517   7.3520   9.9627   1.1700
   1       2.3334   3.4323   5.7773   8.8208   1.1570
   2       2.3334   3.4337   5.8133   8.8208   1.1555
   3       2.0380   3.1847   5.8173   8.8208   1.1698
   4       2.3269   4.2734   6.7002   8.8208   1.1555
   5       2.3334   3.4337   5.7773   8.8208   1.1569
   6       2.3263   3.4478   5.8173   8.8208   1.1665
   7       2.0380   3.1847   5.8173   8.8116   1.1635
```

```
ALTER PIXEL COUNT IN OUTER LOOP, ALIGNMENT IN INNER LOOP:

speed of rgb2yuv in million pixels/second
 p % 8    1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv
   0       2.0993   4.2090   7.6889  12.3181   0.9785
   1       2.0430   4.0662   5.7378   9.3103   0.9797
   2       2.0380   4.0860   5.6988   9.4043   0.9741
   3       2.0380   3.1254   5.7378   9.3103   0.9785
   4       2.1098   4.0860   7.4831   9.4148   0.9786
   5       2.0331   4.0662   5.6988   9.3000   0.9775
   6       2.0430   4.0860   5.7773   9.4148   0.9741
   7       2.0380   3.1138   5.6988   9.3103   0.9774

speed of yuv2rgb in million pixels/second
 p % 8    1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb
   0       2.3269   4.2734   7.3455   9.9627   1.1700
   1       2.3269   3.4337   5.7773   8.8208   1.1569
   2       2.3334   3.4323   5.8173   8.8208   1.1555
   3       2.0380   3.1860   5.7733   8.8116   1.1700
   4       2.3334   4.2734   6.7002   8.8208   1.1569
   5       2.3269   3.4323   5.8173   8.8208   1.1585
   6       2.3269   3.4337   5.7773   8.8116   1.1683
   7       2.0430   3.1847   5.7773   8.9146   1.1635

ALTER PIXEL COUNT IN OUTER LOOP, ALIGNMENT IN INNER LOOP:

time needed to execute rgb2yuv once in nanoseconds
 p % 8    1 pixel 2 pixels 4 pixels 8 pixels srgb2yuv
   0        476.4    475.2    520.2    649.5   1022.0
   1        489.5    491.9    697.1    859.3   1020.7
   2        490.7    489.5    701.9    850.7   1026.6
   3        490.7    639.9    697.1    859.3   1022.0
   4        474.0    489.5    534.5    849.7   1021.9
   5        491.9    491.9    701.9    860.2   1023.1
   6        489.5    489.5    692.4    849.7   1026.6
   7        490.7    642.3    701.9    859.3   1023.2

time needed to execute yuv2rgb once in nanoseconds
 p % 8    1 pixel 2 pixels 4 pixels 8 pixels syuv2rgb
   0        429.7    468.0    544.5    803.0    854.7
   1        429.7    582.5    692.4    906.9    864.4
   2        428.6    582.7    687.6    906.9    865.5
   3        490.7    627.8    692.8    907.9    854.7
   4        428.6    468.0    597.0    906.9    864.4
   5        429.7    582.7    687.6    906.9    863.2
   6        429.7    582.5    692.4    907.9    855.9
   7        489.5    628.0    692.4    897.4    859.5
532.175 seconds in TestSpeed,
of which 532.135 seconds in color space conversion routines (99.99%)
```

7.3 References

[1] Richard C. Leinecker, Visual C++ 5 Power Toolkit, Ventana Communications group, ISBN 1-56604-528-2

[2] MMX technology developers guide, Intel, http://developer.intel.com/drg/mmx/manuals/dg/devguide.htm

[3] Intel Architecture MMX™ Technology Programmer's Reference manual, Intel, http://developer.intel.com/drg/mmx/manuals/prm/prm.htm