Digital Video Over TCP/IP

Master's Thesis by Jacob Ström

Supervisor: Per Andersson

Lund Institute of Technology
Sweden

Email: J.Ström: d91js@efd.lth.se
P. Andersson: pera@dit.lth.se

July 1995

Abstract

This paper describes an implementation of a digital video system for local area networks using Ethernet and TCP/IP. The system consists of cameras connected to a LAN through camera servers and personal computers attached to the same net. The software development for both the camera server and the PC is described, in particular the development of a greedy video compression technique that is targeted for low bit rates with near static images and bit rates comparable to MPEG1 with moving images. Furthermore, a way to transmit the images without overloading the PC nor the network is designed and described.

Contents

1 Introduction

In many buildings of today, a local area network is present. A video camera server that connects directly to such a network and presents its images on a personal computer (PC) connected to the same network has many advantages over traditional dedicated video networks. Additional cable drawing is avoided, the camera server can easily be moved and the images can be viewed from any PC. The digital medium makes it possible to send not only image data but also control information like dooropening or doorbell signals. In addition to that, the digital video format makes it possible to implement features like automatic storage of video sequences triggered by motion detection.

When constructing such a system, several problems appear. The network must remain functional for other tasks also during video transmission. This means that some kind of video compression technique must be used to reduce the use of bandwidth. A compression algorithm that produces very little data if the image is almost static is preferable. Even with video compression, the camera server must have some bandwidth limitation to guarantee that it does not use up all the network bandwidth. The camera server must also see to that no more data is sent to the PC than it is capable of handle.

In this paper, the three areas shown in figure 1 are covered. A solution to the *communication* problem is presented, a compression *algorithm* is developed and evaluated, and the process of the *implementation* is described.

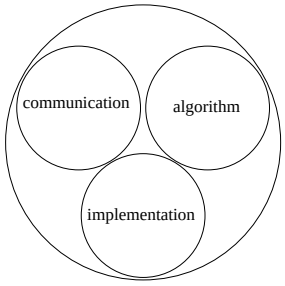

Figure 1: Areas covered by this paper: Solution to the *communication* problem, *algorithm* evaluation and description of the *implementation* process.

The communication problem is divided into two main problems: The *congestion problem* which is about preventing the camera server from overloading the PC with images. This is solved using a simple acknowledgement from the PC for each image. The other problem concerns *bandwidth limitation*, that is how to prevent multiple camera servers from overloading the network. To solve this problem, each camera server has a byte budget that is renewed each second. If the camera runs out of bytes no more images are sent that particular second.

The compression algorithm developed is called *DC-thresholding* and uses DCT transform coding just like MPEG[1]. The way DC-thresholding takes advantage of temporal redundancy (similarities between adjacent images in a sequence) is less sophisticated than in MPEG, but more streamlined to the hardware requirements and still very useful. When compressing almost static sequences DC-thresholding reduces the amount of data to be sent with more than 80%

[1] MPEG is a ISO/IEC standard (no 11172-2) for video compression.

compared to the case when no regard to the temporal redundancy is taken. As MPEG is becoming more and more accepted as the standard format for moving images, a section in the report is devoted to describe how to incorporate video sequences compressed with DC-thresholding into the MPEG-format.

The system was implemented in two programs, one for the camera server and one for the PC, both written in the C/C++ language. The software was optimized to meet the performance of an Intel 486DX66 platform running Windows. The camera server and the PC communicated over a 10Mbit/s Ethernet LAN.

The next section describes the system hardware. A solution to the communication problem is presented in section 3. The compression algorithm is described and evaluated in section 4. The implementation process is covered in section 5 and conclusions are presented in section 6.

2 System hardware

The hardware consists of three parts; the *camera server*, the *network* and the *personal computer*.

2.1 The camera server

The camera server is a piece of hardware designed by Anders Hedberg and is described in detail in his master's thesis [Hedberg95]. A shorter description is presented here.

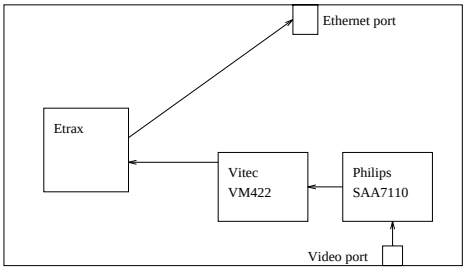

Figure 2: Data flow in the camera server hardware.

The camera is connected to the *video port*, which is a normal video composite connection. The analogue image is digitized by the *Philips SAA7110* chip and is then forwarded to the *Vitec VM422* chip. The image now undergoes a discrete cosine transform (DCT) and a quantization. The next stop for the data is the *Etrax*, a combined I/O and CPU chip. The Vitec chip performs run length coding at the same time as the data is transferred. This means that it encodes a small sequence, sends a 16bit word and encodes again etc. Etrax compresses the data even more using DC-thresholding before it is sent to the net through the *Ethernet port*. Although the camera server is capable of sending images in a variety of sizes, all software was written for a fixed size of 192x144 pixels, using 24bits/pixel colour.

2.2 The network

The network is a normal Ethernet local area network (LAN) with a bandwidth of 10 Mbit/s. Since the protocol used is TCP/IP, a routable protocol, the camera

server will function over larger networks than ordinary LANs. For instance, camera traffic over the Internet is possible. Not to disturb other users of the LAN, the video traffic over the net is specified to use no more than 10% of the total bandwidth. The camera server is thus not allowed to produce more than 1000kbit/s.

2.3 The personal computer

The personal computer, or PC, is the last link in the chain. Although the camera server is independent of the PC system used, the software in this project has been developed and run on an Intel 486DX2/66 based PC running Windows. The network card in the PC is connected via a bus that do not provide direct memory access (DMA) data handling, i.e. the processor must be invoked for moving the data between the buffer memory on the network card and the memory of the PC. The size of the buffer on the network card is less than 3000 bytes. Apart from the network card, no extra hardware is used. The task of the PC is to receive the data from the network, decompress it and display it on the screen in real time. It must also send control information back to the camera server to avoid congestion problems, set desired number of frames per second, etc.

3 Solution to the communication problem

Two problems are to be solved in order to provide smooth communication. First, the camera server must not send more images to the PC than it is capable of handle. This is referred to as *the congestion problem*. Second, the camera server must not overload the network by sending too much data on to the LAN. Therefore a bandwidth limitation must be set up, and this results in two problems. First *the control problem* which is the most obvious one; how to stay within the bandwidth limitation? But before concentrating on this, another problem must be solved: How do we measure the output bandwidth, *the measuring problem*?

3.1 The congestion problem

The camera server must always see to that the PC can handle all the information sent to it. This is for two reasons. First, the camera server do not know the number of frames per second (fps) that the PC is capable of decoding and displaying. It cannot send five frames per second if the PC only can present four fps. The other reason is that the PC can be busy doing something else. For example, the Windows operating system halts all running processes when the user is moving or resizing a window. Serious congestion problems would occur if the camera server would not be hindered to send images during that period of time.

The easiest way of dealing with this problem is to let the PC send an acknowledgement (ACK) for each image it receives, and prevent the camera server from sending another image if the previous one has not been acknowledged. One drawback with this method would be if the camera server spends too much time waiting for acknowledgement, and therefore degrades system performance. The solution could be to implement a so called window. This means that the camera server has a credit of say five images. When it sends an image, the credit decreases by one. Every time it receives an acknowledgement, the credit

increases by one. When the camera server runs out of credits, it is prevented from sending any more images. This would make it possible for the camera to send images to the PC without waiting for the acknowledgement, and yet be sure that no more than five images were to be buffered in the network hardware of the PC.

For such a windowed approach to be successful, the PC must be able to receive data from the network while decompressing the images without time penalty for the communication. This can be done if the network card uses DMA, i.e. it is writing the data directly to the right place in memory without invoking the processor. If so, the PC can start decompressing the next image immediately after having finished the last one, and a windowed transmitting procedure will indeed boost the performance of the system. The network card specified did however not provide DMA, and the data receival and the decompression must therefore be made sequentially. Hence a windowed approach would not increase performance. Therefore the first method (with a simple ACK) was used.

The first approach used was to perform the operations in the following order:

Camera Server	*PC*
grab image	wait for image
send image	receive image
wait for ack	present image
get ack	send ack

This resulted in bad performance since the PC was stalled during the grabbing of the image, which is a fairly time consuming process. The camera server also had to wait for the PC to present the image. By grabbing the image before waiting for acknowledgement the performance was sped up considerably. The two time consuming tasks, image grabbing and image presenting, could now be done simultaneously, as shown in the table below:

Camera Server	*PC*
grab image	present image
wait for ack	send ack
send image	receive image

One disadvantage with this approach was that the image was further delayed. However, the increase in framerate was considered more important.

3.2 The measuring problem

To be able to control the bandwidth usage, the camera must be able to measure it properly. As the camera first grabs the whole image and then sends it, the data stream will be quite bursty, according to figure 3: Most of the time the camera server is idle, but when it is sending data, it uses up all the bandwidth of the network. Trying to control such a dataflow to get an even stream of packages and hence a constant use of bandwidth seems like a hard problem. Fortunately, this does not have to be done. The only thing to prevent is traffic overload, and this can be done by having a low average traffic. It is not a problem with high peaks of bandwidth usage as long as the average bandwidth usage is well below the capacity of the net. Hence, the key to the problem is to control, and measure, the average use of bandwidth.

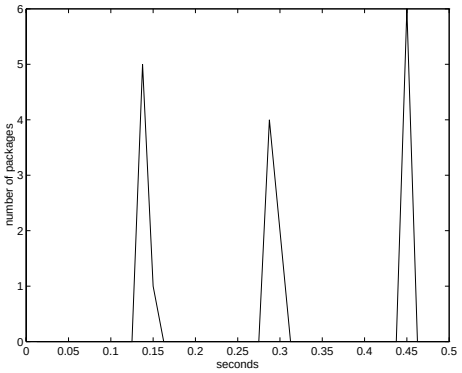

Figure 3: The output of the camera server. Each image sent causes a nail in the diagram.

In the implementation of the camera server, a counter is incremented with the number of bytes sent in each packet. Each second, this counter is read and the average use of bandwidth is calculated, whereafter the counter is reset to zero. Experimental results verify that this gives a stable and reliable measuring of the bandwidth usage. One disadvantage with this is that it only measures the raw data. It does not measure packet header overhead, nor bandwidth usage in the form of retransmissions. Since a user of the system must have this in mind when setting the bandwidth limitation, it could be advantageous to have an estimate of the amount of overhead information that is sent per raw byte. That is, the ratio $\frac{\text{rawdata+overhead}}{\text{rawdata}}$ is of interest. This ratio is however impossible to calculate as it is not static. If the raw data perfectly fit the size of one packet, the minimum overhead ratio is achieved (left part of figure 4). If, however, the raw data contains one byte to much to fit, two headers are required, almost doubling the overhead, as seen in the right part of figure 4.

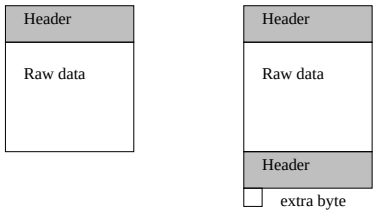

Figure 4: The amount of overhead data per byte is not constant.

This is close to the worst case ratio, which of course can be used as an estimate. The next problem concerns retransmissions. Due to the sending procedure of the CSMA/CD, a packet can theoretically collide, and therefore be resent, an infinite number of times. Due to these problems, this subject was not considered to fall within the scope of this master's thesis, and the user thus has to live with estimations of raw bandwidth use only.

3.3 The control problem

The reason to control the output of the camera server is that it must be made sure that the camera servers attached to the net will not overload the system. This is done by giving each camera a maximum bandwidth limit. If the sum of

the limits is well below 10 Mbit/s, the system will be secure.

To be able to control a system, both a measuring variable and a controlling variable are needed. The measuring variable in this case is undoubtly the bandwidth usage, but when choosing the control variable, at least two are possible; **quality** and **framerate**.

Quality: The Vitec chip gives the opportunity to vary the quality. Choosing larger quantization steps in the DCT compression procedure results in more zeroes in the DCT-matrix and thus fewer bytes of image data.

Framerate: The usage of bandwidth is directly proportional to the number of frames per second (fps). Lowering the fps, results in a lower use of bandwidth.

In the implementation the framerate was used as control variable for several reasons. First, it is easier to implement. Second, even though setting the lowest quality, the data stream does not go to zero. This means that some form of framerate control must be used anyway for low limits. Thirdly, a reduced framerate lowers the need for computational power of the PC when decompressing the images. Even though this is true for quality reductions as well to some extent, the effect is not that big. If zero frames are sent per second, they will not need any processing power from the PC at all. However, if you send images of the lowest quality to the PC, they will use up to 80% of the processing power required for decompressing and displaying images of the highest quality.

When controlling the output bandwidth, two approaches can be used. The first is to look at the limit as the value the bandwidth usage should have. If the usage is less than the limit, the camera server will try to increase it, and if the usage is more than the limit, it will try to decrease it. In order not to exceed the limit, the setpoint for the control algorithm could be for instance 80% of the limit. There are several problems with this approach. One is depicted in figure 5. At (A) in the figure, a network disturbance prevents the camera server

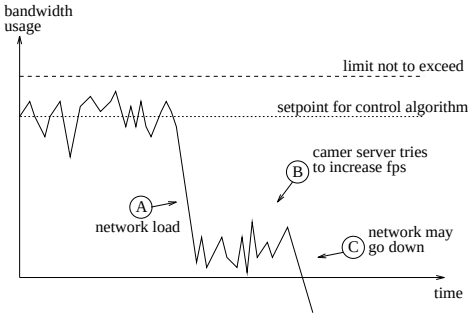

Figure 5: When the network gets heavy loaded the control algorithm will try to increase the fps.

from sending. The control algorithm will discover a decrease of the bandwidth usage, and try to increase it, (B), by sending more data to the already disturbed net. This is the opposite of the desired behaviour – when the network is heavy loaded, the camera tries to send more data. This erroneous behaviour can be avoided by measuring, not the data sent to the net, but the data produced by the compression algorithm. Another problem illustrated in figure 6 will however still occur. When the image sequence is almost static (A), the amount of data

produced by the compression algorithm gets very small. The control algorithm will then increase the framerate (B), and settle to a very high value yielding the desired bandwidth usage. When the image starts to move again (C), the camera server will try to send this high number of frames per second to the net, and the limit will probably be exceeded.

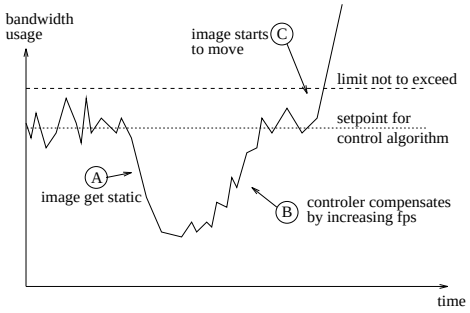

Figure 6: When the sequence is static, the fps will be increased to compensate for this. When the images starts to move again, the bandwidth use explodes.

To avoid these problems, another approach is used. Here, the limit is more like an emergency brake: It will always work, but since it is not used very frequently, the behaviour need not be nice. The user at the PC decides a bandwidth limit not to exceed, and a desired number of frames per second. The limit is set so high that video with the specified framerate will not normally reach the limit. The camera server has a budget of a certain number of bytes per second, stored in a variable. Before sending each image, the variable is decreased by the size of the image in bytes, and if the result is less than zero the image is not sent. The variable is reset every new second. The method is illustrated in figure 7.

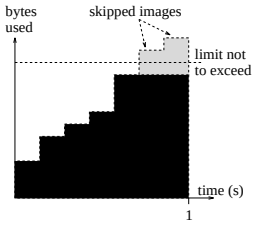

Figure 7: When the camera server has used the whole byte budget, it cannot send more images that particular second.

This approach gives a straightforward and secure solution to the problem. However, if too low a limit is set, bad behaviour will be the result. The algorithm will then send a small number of images rapidly, followed by a long pause.

4 Video format and compression algorithm

The Vitec video compression chip delivers the image in the VMP format, which is different from MPEG. As MPEG achieves better compression ratios than VMP and is a common standard, it would be advantageous to send MPEG data from the camera. To evaluate the performance of the data communication however, VMP is as useful as MPEG, and hence the VMP format was used in the

communication. A translation between VMP and MPEG was thereby avoided. Another advantage was that a decompression program for the VMP format was easier to implement than a MPEG dito. A more thorough description of the MPEG and VMP formats can be found in appendix A.

4.1 Exploiting temporal redundancy

In many video sequences, parts of the image are static throughout the sequence. For instance, a typical scene for a surveillance camera is a person walking in an empty corridor. The person only occupies a small part of the image, and the rest is static. Another example is a video telephone sequence, with a moving head in front of a static background. In situations like this, the method to code each frame separately results in significant amounts of temporal redundancy in the code – the background is sent over and over again. It would be more advantageous to encode the images differently, so that the static parts of the image would not have to be resent.

As the Vitec video compression chip does not provide differential encoding, this cannot be done. An alternative to differential encoding is to send only those blocks that has changed in respect to some metrics. The decision which blocks are to be sent must then be made by the Etrax chip inside the camera server. The most correct way of determining if a block has changed is to calculate the summed root mean square error, $RMSE$:

$$RMSE = \sqrt{\frac{1}{64}\sum_{i=1}^{8}\sum_{j=1}^{8}(A_{ij} - B_{ij})^2}$$

A and B are the blocks from the previous and current images respectively. If the $RMSE$ exceeds a given threshold value, the block is considered changed. As the Vitec chip delivers the blocks in coded form, the Etrax has to decode the blocks in order to calculate the $RMSE$. The previous image also has to be stored in memory. This results in a costly procedure both concerning computation time and memory, and is not feasible with the hardware configuration used.

4.2 DC-thresholding

Instead of using $RMSE$, another metric, based on DC-values[2] only, is used to determine if a block has changed: the absolute DC error, $ADCE$.

$$ADCE = |A_{11} - B_{11}|$$

The block is thus considered to have changed if the DC-component of the current and previous block differs more than a given threshold value. We refer to this technique as *DC-thresholding*. As the DC-component is stored without quantization and sent first in the data stream of a block, the comparison is very straightforward and can be done without using up much computational capacity. The memory requirements are also very moderate with the DC-components using only 864 bytes of storage compared to 40.5 KB for a whole image, calculated on an image size of 192x144.

[2]The DC-value is proportional to the mean value of the block, see Appendix A for further information.

Test images show that a threshold value of 7 used on almost static sequences results in a boost of the compression ratio by a factor of five. Another advantage of this method is that the time to decode a single frame on the PC is reduced by 50%. This is mainly due to the fact that the PC avoids doing the time consuming inverse discrete cosine transform (IDCT) on the blocks that are not sent. To illustrate the result, two test sequences are investigated. The first contains very little movement and illustrates the advantage of using the DC-thresholding technique on sequences that are almost static. In the second sequence, a large part of the image is moving. This test sequence is provided to show that the technique is useful even for non static images. Both sequences use a threshold value of 7 for the DC-thresholding. The same value was used in the implementation of the camera server as it, after long time evaluation, seems to be sufficiently high to give good compression results and sufficiently low to avoid too much degradation of the image.

The first sequence consists of the two consecutive images in fig 8 and fig 9, the second one occupying 5538 bytes of storage, yielding a compression ratio of 15.08[3]. Even though they seem quite similar to the naked eye, there is a difference between them. This difference is calculated and plotted in figure 10. The DC-thresholding algorithm now decides which blocks to send in order to produce an image that is similar to the second image in the sequence (fig 9). The selection is shown in figure 11. The image produced, shown in figure 12, consists of 1052 bytes which means a reduction with over 80%, or a compression ratio of 78.84. The error introduced by the DC-thresholding, that is the difference between the image produced (fig 12) and the second image (fig 9), is plotted in figure 13.

Figure 8: First image in the sequence.

Figure 9: Second image, 5538 bytes.

Figure 10: Difference between 1st and 2nd.

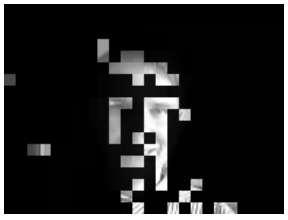

Figure 11: Blocks selected by the DC thresholding algorithm.

Figure 12: 2nd image using DC thresholding, 1052 bytes.

Figure 13: Error introduced by DC thresholding.

[3] $\text{compression ratio} = \frac{\text{original size}}{\text{compressed size}} = \frac{192 \times 144 \times 3}{5538} = 15.08$

The differential images are calculated as:

$$D_{i,j} = \text{CLAMP}(0, A_{ij} - B_{ij} + 128, 255)$$

where A and B are the starting images and the CLAMP operator sees to that the value sticks to the range $[0, 255]$. Due to he $+128$ term, zero differences will be represented with a gray colour.

The next sequence (fig 14 & 15) contains more movement. This is easily recognized in the differential image 16. This means that the DC-thresholding algorithm must send more blocks as seen in figure 17. The resulting image (figure 18) is 3680 bytes in size[4] which is to be compared to 5646 bytes (fig 15) if DC-thresholding had not been used – a reduction with 35%. This lower value is due to the movement in the sequence. These extra bytes are well spent – the error introduced by the DC-thresholding, shown in fig 19, remains low.

Figure 14: First image in the sequence.

Figure 15: Second image, 5646 bytes.

Figure 16: Difference between 1st and 2nd.

Figure 17: Blocks selected by the DC algorithm.

Figure 18: 2nd image using DC thresholding, 3680 bytes.

Figure 19: Error introduced by DC thresholding.

To be able to send empty blocks the VMP-format (as described in appendix A) has to be slightly modified. If a sequence of blocks are considered static, they are encoded as 1001rrrrrrrrrrr, where 1001 are control bits to differ the word from an ordinary DC-value (which starts with 1000), and r is the number of blocks that are static. For totally static images, such as an empty corridor, each frame could theoretically be represented with a single 16 bit word for each colourplane (i.e. Y, Cr and Cb), making the bandwidth usage drop to $6 \times 48 = 288$ bps[5]. In reality however, image noise changes the DC-values sufficiently to trigger the DC-thresholding mechanism. A more reasonable estimation of bandwidth usage transmitting static images would be 1000 bytes per image, or 47Kbps for the

[4] The compression ratio has improved from 14 to 23.
[5] Calculated with 6 frames per second.

same 6-fps-sequence. Without DC-thresholding the average number of bytes per image is 5500 resulting in a bandwidth of 256Kbps.

4.3 Errors introduced by DC-thresholding

The way that the DC-thresholding compresses the data is by approximating part of an image with the same part in an earlier image. If they are not exactly the same, which they never are, this introduces error. What is the visual impact of these errors? They can appear as traces after moving objects. When an object changes large parts of a block, its DC-component changes and the block is sent (figure 20). When the object moves out of the block, the moment before disapearing completely, it may just touch the block so that only a few pixels are altered (figure 21). In the next frame, the object is outside the block, but the DC-component has not changed enough since the last frame, so the block will not be resent (figure 22). Even if the object does not leave a trace,

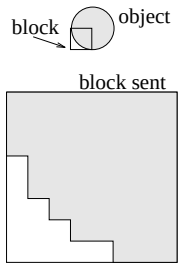

Figure 20: First image, block sent.

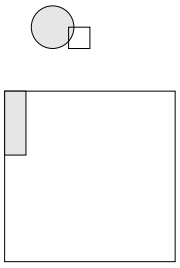

Figure 21: Second image, block sent.

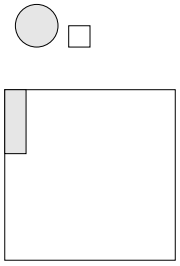

Figure 22: Third image, block not sent.

the movement can look somewhat peculiar. The first time an object touches a block, it may not change enough pixels to trigger the DC-thresholding mechanism. The block is not sent and the object seems to be shorter. Suddenly enough pixels are changed by the object to trigger the algorithm, and the block is sent. The object seems to grow in the moving direction. This enlonging and shrinking movement can make distant lorrys to look like larvae! However, this and the previous problem are only appearent on distant objects, and is therefore not that annoying. Possible remedies may be to periodically flush old blocks through bypassing the DC-thresholding algorithm and thereby sending all blocks. As a matter of fact this is likely to happen spontaneously, as weather changes influences the lighting conditions so that all DC-values change at once.

Another type of error is when the content of a block changes dramatically but the DC-value happens to be the same. In figure 17 it is possible to notice such an error made on the persons right arm (left in the picture). The effect of this erroneous behaviour is mildered by the fact that such blocks tend to blend into the context since the mean value (i.e. the DC-value) is correct.

4.4 DC-thresholding in MPEG

The DC-thresholding technique makes it possible to exploit temporal redundancy. The method may be blunt and suboptimal, but it has the advantage of being fast and easy to implement. Today most low cost real time MPEG encoders completely disregard the similarities between the images, each frame

is coded separately. Very little extra hardware would make it possible to do DC-thresholding as well, and thereby increasing the value of the product. The technique must then be made compatible with the MPEG format. An MPEG sequence can contain I and P frames[6]. In the I-frames, all macroblocks are I-blocks, i.e. they consist of encoded DCT coefficients without reference to other images which is illustrated in fig 23. In the P frames, the macroblocks can be

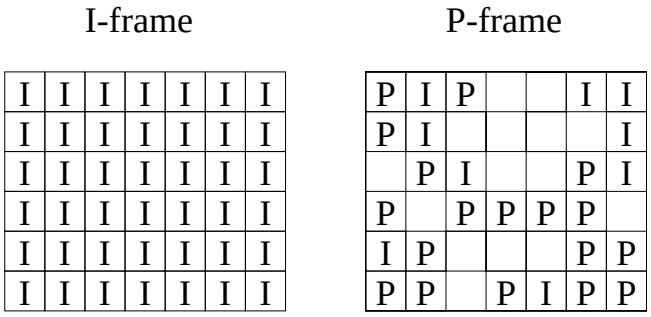

Figure 23: I-frames can only contain I-blocks. P-frames can contain I-blocks, P-blocks and skipped blocks.

I-blocks, P-blocks, or skipped macroblocks, as seen in the same figure[7]. A P-block includes a reference to a piece of an earlier image that looks almost like the block to code. It also contains a DCT encoded error correction that is DCT encoded. A skipped block is interpreted as a P-block with reference to the same block in the previous image, and with no error correction. A DC-thresholded image is coded as a P-frame, with the selected macroblocks being coded as I-blocks and the other blocks coded as skipped blocks.

If this procedure went on, all frames would be P-frames. This is not allowed in the MPEG standard. Therefore the MPEG/DC-thresholding encoder must periodically bypass the DC-thresholding step to allow an I-frame to be produced.

4.5 Motion detection

A spin off effect of the DC-thresholding technique is a simple way of making motion detection. If the number of blocks that has changed exceeds a certain threshold value, the image is considered changed. The PC application can be programmed to automatically enlarge or flash the window containing the moving image. If no operator is available, the video sequence can be stored to disk etc.

5 Implementation

5.1 Software developing environment

The camera server was programmed in C++ using a gcc version that produced Etrax- assembly code. A TCP/IP stack protocol was already implemented. The developing environment on the PC was Microsoft Visual C++ (MSVC) and the TCP/IP communication was done through Winsockets. To handle incoming data correctly, the winsocket was programmed to create a Windows message when data was available. The program was then to jump to a procedure that

[6]The B and D frames also possible are not interesting in this context.
[7]Skipped macroblocks are shown as empty squares in the figure.

processed the incoming data.

In order to test the PC program, it was constructed to communicate with another PC running the same program. When this worked properly, the program was easily modified to communicate with the camera server.

The camera server was now programmed to communicate with the PC. As the functionality of the PC program was already verified, bugs in the system had to originate from the camera server software, which made it easier to make the system work.

An interesting issue was the performance of the system – the higher data rate, the better image quality could be expected. Maximum performance was a data rate of 1Mbit/s. To verify the presumption that the camera server was the bottleneck, a PC to PC performance test was also made, yielding 2Mbit/s.

The software for presenting the image on the PC was then written. The blocks building up the picture were first decoded, then processed by an inverse discrete cosine transform (IDCT). This resulted in the three colour planes Y, Cr and Cb which, after upsampling of the latter two, could be transformed into RGB and drawn as a bitmap on the screen. The inverse discrete cosine transform procedure was taken from the Berkeley MPEG software and incorporated in the program.

The performance of the decoding and on-screen-presentation of the images were critical. Without any optimization of the code, a grayscale image was decoded and drawn to the screen in 0.98s, giving a theoretical maximum framerate of 1.02 frames per second[8]. By optimizing the code in different ways the framerate rose to 6.25 fps, and by decoding only those blocks that had changed, the framerate almost doubled to approximately 12, depending on the threshold value and the type of image.

When going into colour the framerate dropped to 1.2 fps and a new era of optimization started. One way of optimizing was to draw only the area of the screen that had changed, another was to use matrixes smaller than 64Kb which is the memory segment length of the Intel processors. Much time was consumed on the transformation from YCrCb back to RGB. By calculating in fixpoint- instead of floating point- arithmetics, and by using precalculated tables instead of multiplying, the framerate was increased to 9.9 fps for the test sequence used.

6 Results and Conclusions

Three areas have been described in this paper; the communications problem, the algorithm developed and the implementation process. In the first area, a solution to the problem has been presented. In the second area, a technique for improving the compression ratio over I-frame-MPEG has been found that is useful, fast and of low complexity, both concerning hardware and software. In the third area, an implementation has been made that worked well, with a performance of 2-3 frames per seconds in the first hardware and software iteration. Possible improvements would be to alter the camera server hardware in order to make DMA handled image grabbing possible. This, combined with DMA image receival at the PC side would enhance performance.

[8] The time to transfer the image from the camera server is not included here.

7 Acknowledgements

I would like to thank Sven Ekström at Axis for his secure guiding through the jungle of the Windows operating system. I would also like to thank Kenny Ranerup, my additional supervisors Martin Gren and Carl-Axel Petersson and a large number of other persons working at Axis. Finally, I would like to thank my supervisor Per Andersson and my friend Tomas Möller for support.

A The MPEG and VMP format

This section serves as an introduction to the MPEG and VMP formats. It is a step-by-step explanation of how to code an image in the two formats. In the MPEG format, three types of frames are possible, I-, B- and P-frames. The coding of an I-frame resembles the coding of a VMP format image very much, so they are presented side by side. For information about the B- and P-frames, look above in the section "DC-Thresholding in MPEG" or consult [mpeg93].

A.1 The coding of an MPEG I-frame/VMP image

Both when coding MPEG I-frames and VMP format images, the image is first transformed from the RGB space into the YCrCb space through a matrix multiplication (left part of figure 24):

$$\begin{pmatrix} \mathrm{Y} \\ \mathrm{Cr} \\ \mathrm{Cb} \end{pmatrix} = \begin{pmatrix} 0.2990 & 0.5870 & 0.1140 \\ 0.5000 & -0.4187 & -0.0813 \\ -0.1687 & -0.3313 & 0.5000 \end{pmatrix} \begin{pmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{pmatrix}$$

In the latter colour representation Y is the luminance and Cr & Cb represent colour. As the human visual system (HVS) is more sensitive to changes in the luminance than in the chrominance, it is possible to encode the chrominance less accurate without degrading the image much. This is done by subsampling the colour planes Cr and Cb, i.e. throwing away every second pixel in both the x and y direction (right part of figure 24).

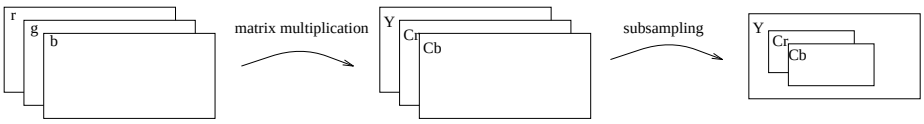

Figure 24: The change of colour representation followed by subsampling.

Now the colour planes are divided into blocks of 8×8 pixels. Since the colour planes are subsampled, four luminance blocks are needed to cover the same area as one chrominance block (figure 25). Therefore four luminance blocks and two chrominance blocks together form a macroblock, which thus covers a 16×16 pixel area in the original image.

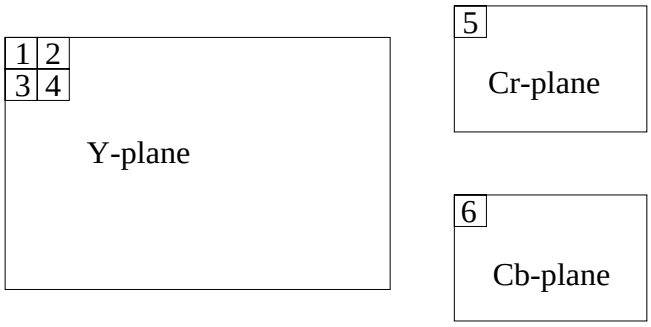

Figure 25: The first macroblock consists of these six 8×8 blocks.

Each 8×8 block is now transformed into the frequency domain using a discrete

cosine transform (DCT). This results in a matrix of frequency components. Here is an example 8×8 block:

$$\begin{pmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 156 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 162 & 162 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 162 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{pmatrix}$$

After DCT transform, the following matrix is obtained:

$$\begin{pmatrix} 235.6 & -1.0 & -12.1 & -5.2 & 2.1 & -1.7 & -2.7 & 1.3 \\ -22.6 & -17.5 & -6.2 & -3.2 & -2.9 & -0.1 & 0.4 & -1.2 \\ -10.9 & -9.3 & -1.6 & 1.5 & 0.2 & -0.9 & -0.6 & -0.1 \\ -7.1 & -1.9 & 0.2 & 1.5 & 0.9 & -0.1 & 0.0 & 0.3 \\ -0.6 & -0.8 & 1.5 & 1.6 & -0.1 & -0.7 & 0.6 & 1.3 \\ 1.8 & -0.2 & 1.6 & -0.3 & -0.8 & 1.5 & 1.0 & -1.0 \\ -1.3 & -0.4 & -0.3 & -1.5 & -0.5 & 1.7 & 1.1 & -0.8 \\ -2.6 & 1.6 & -3.8 & -1.8 & 1.9 & 1.2 & -0.6 & -0.4 \end{pmatrix}$$

The top left element of this matrix is proportional to the mean value of all pixels values in the block and is called the DC element. The other elements represents the different frequency components of the block, with elements near the DC element representing lower frequencies and elements near the bottom right corner representing higher frequencies.

The human visual system is more sensitive to lower than to higher frequency components. This means that elements in the bottom right corner can be suppressed without degrading the image much. This is done in the next step, the quantization. The matrix is now divided with a quantization matrix that has higher numbers in the bottom right corner and lower numbers near the DC-value. Here comes an example of such a matrix:

$$\begin{pmatrix} 1 & 13 & 13 & 13 & 13 & 13 & 50 & 50 \\ 13 & 13 & 13 & 13 & 13 & 50 & 50 & 50 \\ 13 & 13 & 13 & 13 & 50 & 50 & 50 & 50 \\ 13 & 13 & 13 & 50 & 50 & 50 & 50 & 100 \\ 13 & 13 & 50 & 50 & 50 & 50 & 100 & 100 \\ 13 & 50 & 50 & 50 & 50 & 100 & 100 & 100 \\ 50 & 50 & 50 & 50 & 100 & 100 & 100 & 100 \\ 50 & 50 & 50 & 100 & 100 & 100 & 100 & 100 \end{pmatrix}$$

After integer division, most of the coefficients are now zeroed:

$$\begin{pmatrix} 235.6 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Now the data is to be run length encoded, to exploit the large number of zeroes. As most of the zeroes are situated in the bottom right corner, it is advantageous to code the elements in a zig-zag order, as seen in figure 26. The DC value, being the most important component for the HVS, is coded separately. The other coefficients are coded as two-tuples, (r,l), where r is the number of zeroes preceding the value l. The array above is therefore coded as: 235.6, (0,-2), (1,-1), (0,-1), (0,-1), (1,-1) (56,0).

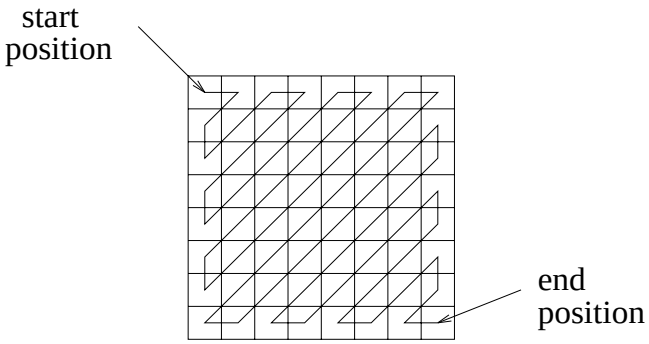

Figure 26: The zig-zag order.

Up to now, the description has been valid for both MPEG I-frame and the VMP format. The first difference is that in the MPEG format, any quantization matrix can be used, whereas the VMP format creates a quantization matrix from only 4 values using the zig-zag order. The top left element is always one. The following 15 elements in the zig-zag order are set to the second of the four values above. The next 26 elements are set to the third value, and the rest of the elements are set to the fourth value. The quantization matrix in the example above is created using that technique.

The way to encode the run length tuples also differs between the two formats. The MPEG format uses bit words of variable length that are looked up in a fixed table. The example above results in the following bitstream: 01001 (0,-2) 0111 (1,-1) 111 (0,-1) 111 (0,-1) 0111 (1,-1) 10 (end of block). The DC-component is encoded separately. For further information, consult [mpeg93].

A.2 The VMP format

The VMP format is somewhat simpler. First, the DC component is coded separately using a word of 16 bits: 1000vvvvvvvv where v is the unquantizised twelve bits of the DC component. Then the run length tuples are coded. If the value l in the tuple (r,l) is in the range of [-8,7] the tuple is coded as a byte 0rrrllll where the r bits encode the number of zeroes preceding the value coded by the l bits. If the value is not in the range of [-8,7] the tuple is coded as a word: 1rrrlllllllllll using 12 bits of value information. When the there are only zeroes left, or when all tuples have beed encoded, an end of block word is inserted having the bit pattern 1000100000000000. If the data stream is not word aligned, a padding byte of 10000000 is inserted before the end of block word. The example (r,l) tuples would be coded as: 1000011101011100 (DC) 00001110 (0,-2) 00011111 (1,-1) 00001111 (0,-1) 00001111(0,-1) 00011111 (1,-1) 10000000 (padding byte) 100001000000000000 (end of block).

Whereas the MPEG format saves one macroblock at a time, the VMP format saves colourplane for colourplane, starting with the blocks building up the Y plane. Preceding the block information is a small header containing this information:

- image width in pixels (2 bytes)
- image height in pixels (2 bytes)
- the four values building up the quantization matrix (4 bytes)
- pointers to the three colour planes (4*3 = 12 bytes)

A.3 Modifications to the VMP format

In order to simplify the implementation, a fixed header was used. The image size was 192x144 and the quantizing values were 1,9,9,9. As the image was decompressed sequentially, the pointers for the colour planes were not needed. Thus the whole header was now redundant and therefore not sent. In order to know how much data every image contained, another header was inserted containing the image size in bytes, and the images data structure actually sent was:

- image size in bytes (2 bytes)
- Y-plane
- Cr-plane
- Cb-plane

References

[Hedberg95] Anders Hedberg. *Netvision – A Hardware Platform for Network Video*, Master's Thesis, University of Luleå, Sweden, July 1995

[mpeg93] ISO/IEC Standard 11172-2. *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 2: Video*, International standard, First edition 1993-08-01.