DIPLOMA THESIS

GRAVE Cave
General Video Client

Trond Arve Wasskog

January 1995

Division of Computer Systems and Telematics
Norwegian Institute of Technology
Universisty of Trondheim

Preface

This report presents the results of a diploma thesis performed at the Department of Computer Systems and Telematics at the Norwegian Institute of Technology. Assistant Professor Roger Midtstraum and PhD student Rune Hjelsvold working at the Database Group has been my advisors during this work.

Two important goals of this work has been to investigate software based digital video, and network transmission of video data. Digital video is a complex area of research, and the there is a lot of comprehensive and detailed literature available. Thorough basis information has been obtained by studying some of this literature, and this process has been both rewarding and time consuming. When coming to implementation, I had to concentrate on playing of digital video, and put network issues in the background. In all phases of this diploma thesis I have learned a lot, and the work with digital video is interesting and challenging. Much work is still to be done, particularly implementation and testing of network solutions for transmission of digital video.

Acknowlegdments

The author wishes to thank the people that have contributed to this report by constructive discussions, comments and suggestions. Morten Strand, what you don't know about color and colormaps isn't worth knowing. Thank's to Arild Bekkadal for constructive criticism on reading the report. Espen Epos, when would this report be finished without your help ?

Thank's to the Independent JPEG Group for providing the IJG software library for JPEG compression and decompression.

Thank's also to Norwegian Computing Center for the VoDoo video system, which this work is partially based on.

My advisor, Rune Hjelsvold, has encouraged my work by suggestions, comments and discussions. I want to express my thanks for invaluable help and support. Also, assistant professor Roger Midtstraum has aided my work by criticism, suggestions and ideas, for which I'm very grateful.

-Trond Arve Wasskog

Abstract

This project investigates the JPEG standard for still image compression in connection with software processing of digital video, referred to as Motion JPEG. Network transmission protocols for transmission of continuous media are presented and discussed as well, in addition to issues concerning image processing required during playback of digital video.

A software based Motion JPEG video player, called **jpegvid** *is constructed and implemented, and essential subjects of the implementation are described. The video player provides several options in order to trade of image quality versus frame rate, for instance scaling of video frames and forcing grayscale output from originally color video sequences. The jpegvid video player is tested, and the experiments and results are presented.*

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Description

Digital video has become common and popular the last few years, and there are already several *digital video players* available for workstations and PC's. This opens new possibilities; digital video is used in *video conferences*, *video mail*, *distance education*, *video-on-demand* and many other areas. A lot of research has been performed on digital video issues; still, much remains to be done. The use of digital video in *multimedia applications* is driving the research forward.

The Department of Computer Systems and Telematics (IDT) at the Norwegian Institute of Technology (NTH) is currently doing research on multimedia. The Database Group is investigating digital video from a database point of view. A system called *Video Storage and Retrieval* (VideoSTAR) is being developed. The system includes a video database, description of movies, searching in video descriptions and browsing of video sequences. In order to play movies, a video player has been developed in a previous diploma thesis [Skeide93]. This player utilizes a dedicated hardware board to perform real-time playback of video. However, it would be attractive being able to play video without depending on this hardware board. A *software video* player provides the common user with a general tool to play video on an ordinary computer, not having expensive hardware.

The purpose of this project is to investigate a software-only solution for playback of digital video. Digital video is usually stored in *compressed* format, in order to use as little storage as possible. There are many ways of compressing digital video. In this project, the *JPEG* standard for compression of still images is used. This means that digital video is stored as a sequences of independent images which are compressed separately. An image is often referred to as a *frame* when talking about digital video.

During playback of digital video, the video frames must be *decompressed* prior to displaying them. As mentioned above, the existing video player uses a hardware board for this purpose. In this project, decompression of frames is performed in software. The software decompression process is unavoidably slower than a hardware-based solution, but it is interesting to examine the performance of such a solution. Important issues include speed of decompression, variations in playback speed, synchronization and drawing of images on a display screen. Especially, some *quality versus speed tradeoffs* will be explored. This means that it should be possible to gain higher playback speed when decreasing the requirements for image appearance quality, and vica versa.

Network transmission is necessary in order to transfer video data from a remote host to a video client. A lot of research has been done on transmission of video data, and some

proposed solutions are presented in this report. However, a software implementation of a video player imposes particular requirements on network transmission. A purpose of this project is to investigate some requirements for network transmission of video data to a software video player.

In order to gather experience and information, a software-based video player is developed and implemented. Decompression of JPEG compressed frames and displaying of frames on the screen is performed purely in software, contrary to the previous video player used. The player provides some options for trading off playback speed of movies versus quality of image appearance. The system is used for exploring the performance of movie playback by a software digital video player, and thorough testing and evaluation of the system is a significant part of this report.

1.2 Organization of the Report

The rest of this report is organized as follows:

- **Chapter 2** investigates digital video and compression of video data. Particularly the JPEG standard is described.
- **Chapter 3** reviews some existing network transmission protocols which are used for video data transmission. The chapter is concluded by some prior and related work with respect to network transmission and digital video.
- **Chapter 4** contains some requirements for a digital video player. Software processing of images is discussed, and some necessary post processing techniques for JPEG decompressed images are presented. Some important software video requirements for network transmission are also discussed.
- **Chapter 5** describes the functionality of the video player system. This includes the user interface and the processing of video streams.
- **Chapter 6** contains the general architecture of the video player system.
- **Chapter 7** describes the construction and implementation of the video player.
- **Chapter 8** describes the experiments which have been performed and the test results are presented.
- **Chapter 9** contains the conclusions of this project and some suggestions for further work.

Chapter 2

Digital Video and Compression

Work with digital video requires knowledge within a wide area of computer technology. This chapther provides basic knowledge of digital video issues and particularly *video compression*. A brief survey of compression methods is provided before moving to the *JPEG* standard. JPEG is the compression method utilized in this project, and JPEG compression and decompression is examined and described thoroughly.

2.1 Introduction

Most people are familiar with *analogue* video. During the last few years *multimedia* applications have become more common, accompanied by a transition form analogue to *digital* video. Contrary to analogue video, which is continuous, digital video is a discrete medium. *Digitalization* is performed in order to represent video in a format which computers can manipulate.

Representation of digital images demands large storage space and high data rates when transmitted through a network. A digital video film is *a sequence of images* which are displayed sequentially on a frame-by-frame basis. The displaying is performed with high speed to give the impression of motion. An image of size 240x360 samples using 8 bits per sample to represent color, uses almost 700 Kbytes when stored in uncompressed format. A video film lasting 1 hour and sampled at 25 frames/second, would use over 62 Gbytes of storage. Today the common user has a disk capacity of a few hundred Mbytes. Consequently, *compression techniques* are necessary in order to reduce the large data volumes.

2.2 Digital video

Digital video consists of images displayed rapidly in sequence to give the impression of motion. An image, called *frame*, is the basic element of digital video. Sound is achieved by playing audio data synchronously with the video frames. One quality measure of digital video playback is the *frame rate*, ie. the playback speed. The frame rate is the number of frames displayed during a given interval, usually denoted by *frames per second*. The European analoge video standard (PAL) yields 25 frames/sec.

Digital images are large data files. An image of 320x240 pixels, using one byte/pixel to store color, results in 77 Kbytes of data. Transport of image data therefore requires high data rates. Likewise, storing the images demands large storage capacity. *Compression methods* are used to reduce the size of the images, and thereby reducing the requirements

for data rates and storage capacity.

When frames are stored on local disk, the video player retrieve the video data directly from it. If frames are stored on a remote disk, network communication performs the transport of frames over the network. Frames are transported from the remote computer to the video player client. A typical video player system is shown in figure 2.1.

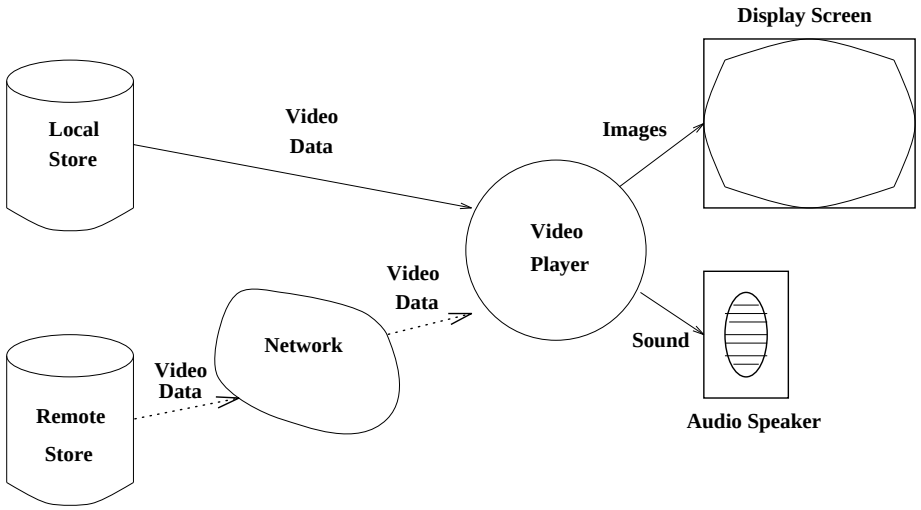

Figure 2.1: Video player system

The main parts of the system are:

1. Fetch frames from store. This means either retrieving data from a local disk or transmitting audio and video frames from a remote computer to a video client.

2. Decompress the compressed frames so they are ready for displaying.

3. Draw the frames on the screen, synchronized with the audio data.

Compression and decompression is the subject of the following sections, while the next chapter is concerned with the transmission of video data through networks.

2.3 Video Compression and Decompression

There are virtually an infinite number of compressing techniques, and there is no such thing as an 'ultimate' technique; one compression algorithm can achieve high compression on one image type, but perform horrible on other types. An application programmer must therefore select a compression algorithm which suits the specific application.

This section reviews some criteria which are used to evaluate compression algorithms. Then some of the most popular and interesting compression algorithms are presented. However, this project is concerned with compression of digital video streams, and the rest of the section is devoted to this compression category.

2.3.1 Compression Evaluation Criteria

There are several evaluation criteria for compression algorithms, and algorithms are likely to be good at some and bad at others. The selection of compression algorithm depends on the criteria which are considered most important in that specific area of use.

Compression methods can be categorised as *lossy* or *lossless*. A lossless compression technique means that all the information in the picture is preserved in the compression process, and decompression produces an image that is identical to the original. A lossy technique will loose some of the picture information during compression, and the decompressed image is not quite the same as the one you started with. Lossy techniques are capable of achieving much greater compression than is possible with lossless methods[1]. Lossy methods exploit that the human eye is not as sensible to some properties of an image as others, and remove this information. The information loss is usually not visible at all. However, if exact image values are required (typical for image analysis), a lossless technique must be applied. Lossless compression also is used for compressing text and other data where losing information is not acceptable.

Another characteristic to consider is the *compression ratio* of a compressor, ie. how much the data volume is reduced. Compression ratio is measured as

$$\frac{Uncompressed\ data\ volume}{Compressed\ data\ volume} \tag{2.1}$$

and usually written as a compression ratio of *M:N* (for instance 3:1). The compression ratio expresses the main purpose of compression, namely reducing the data volume. However, a compression ratio is a useless measurement without a specification of the data that was compressed, because compression techniques achieve different compression ratios on different data.

The speed of a compression algorithm must be considered as well. A compressor can be relatively slow, but obtain high compression ratios (and vica versa). Whether to use such algorithms depends on the criterion you value the most: compression ratio or speed. When discussing speed, the *symmetry* of the algorithm also has to be mentioned. A symmetrical algorithm uses the same order of time both on compression and decompression. A typical non-symmetric algorithm spends longer time performing compression of data than on decompression.

A last criterion to be mentioned is the *standardization* of the compression algorithm. This property is of interest when exchanging data with other people, a topical question in the networking community of today. If a non-standard algorithm has been used to compress data, other people are not likely to have the same decompressor as you, and can not decompress the data. The purpose of *standardization groups*[2] is to define world-wide standards in order to avoid this problem.

2.3.2 Compression Techniques

As stated in the introduction, there are virtually infinite ways of compressing data. However, some techniques have proved to be well suited for certain areas, and are therefore widely used or emerging. This section takes a shallow look at:

1. Wavelet theory

[1] Lossless methods typically compress 2:1 or 3:1, while lossy techniques can achieve 100:1 compression.
[2] For instance ISO, ANSI, CCITT

2. Vector quantization
3. Fractal compression
4. DCT-based compression

Wavelet theory is the new and very promising technique for compression. A *wavelet transform* is used to transform image data into its temporal *and* frequency representation. This is opposed to DCT-based compression (see below), which uses only frequency representation. Wavelet compression is lossy, and achieves high compression ratios compared to current standards. Nevertheless, it is still too computational expensive to be used in real-time. In addition, it has not travelled as far in the standardization process as other compression standards. However, it is predicted to be the image and video standard in the future. Particularly for video compression wavelets seem appropriate, on account of its temporal properties. For an introduction to wavelet theory, see [Hilton94].

In *vector quantization* small arrays of pixel samples in an image are mapped into a fixed subset of representative values. In order to understand this, take a look at *scalar quantization*. Scalar quantization is a many-to-one mapping. For instance, if an image has 10000 different colors and the display screen can only display 256 colors, the 10000 colors are mapped into the 256 colors by scalar quantization. The same principle applies to vector quantization, except that small arrays of pixel samples are mapped, compared to single pixel values for scalar quantization. Vector quantization exploit that colors tend to concentrate in certain areas of an image (for instance, an image of a tree will typically contain much green), and this is represented by vectors. The mapping imposed makes vector quantization a lossy compression technique. The above description is based on [COMP-FAQ94].

Fractal compression uses fractal geometry, a mathematical technique. Originally fractal geometry was used to generate natural looking images, until somebody thought of using it in the reverse direction: compress natural looking images by representing them as fractal geometry mathematics. This inverse problem, unfortunately, remains unsolved. There exists a 'partial' solution, and it has achieved compression ratios of 4:1 to 100:1. The problem with fractal compression is the speed; it is an asymmetric technique and may spend several hours compressing, while decompression is faster. However, it is a promising technique, and a lot of research is being done in this area. The above information is based on [COMP-FAQ94].

DCT-based compression tecniques use the *Discrete Cosines Transform*[Pennebaker93] to transform an image into its frequency representation. In this representation, image redundancy is removed in order to achieve compression. DCT-based techniques are not initially lossy. However, to achieve high compression ratios, lossiness is introduced[3].

DCT-based compression is today's standard for both video and full color, still image compression, and is widely used. ISO[4] has defined the following two compression standards:

- *JPEG* for compression of still images.
- *MPEG* for compression of moving images, ie. video sequences.

Another popular standard for compression of motion images is *H.261* defined by ITU-TS[5]. H.261 is designed particularly for video-conferencing.

[3] JPEG defines a lossless mode, which is not widely used.
[4] International Standardization Organization
[5] International Telecommunications Union, Teleconferencing Sector, former CCITT

The JPEG compression standard is used in this project, and will be thoroughly described in subsequent sections. The two standards for compression of moving images, MPEG and H.261, are presented next.

2.3.3 The MPEG Standards

MPEG stands for *Motion Picture Experts Group*, and is the ISO standard for compression of motion image sequences [Gall91]. MPEG consists of three different standards [Haugen94]:

- MPEG-1: 'Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to 1.5 Mbit/second'. Applications areas:
  - CD-ROM video players
  - Multimedia PC's
  - Multimedia workstations
  - Video electronic mail
- MPEG-2: 'Generic Coding of Moving Pictures and Associated Audio'. Application areas:
  - Digital TV
  - Digital VCRs
  - High Definition Television (HDTV)
- MPEG-4: 'Very Low Bitrate Audio-Visual Coding'. Applications areas:
  - Videophone
  - Video conferencing

The remaining of this section presents some basic information based on the MPEG-1 standard. MPEG-2 is conceptually similar MPEG-1, but includes extensions to cover a wider range of applications. MPEG-4 is suitable for video transmission over analog telephone lines, but has not yet been released. MPEG-2 and MPEG-4 are described in [Haugen94].

MPEG provides VHS TV quality at a compressed rate around 1.5 Mbit/sec. MPEG exploits that the difference between two adjacent images in a video sequence usually is very small. It defines an algorithm for prediction of the next image in a sequence. If the image differs from the prediction, this difference is saved as the new frame. A frame sequence is composed of the following different frames:

- Intra Frames (I): Independently coded frames; the frames are not dependent of any others.
- Predicted Frames (P): Predicted frames coded with reference to the previous I or P frame.
- Bidirectional Frames (B): Predicted frames coded with reference to the previous and the next P frame.

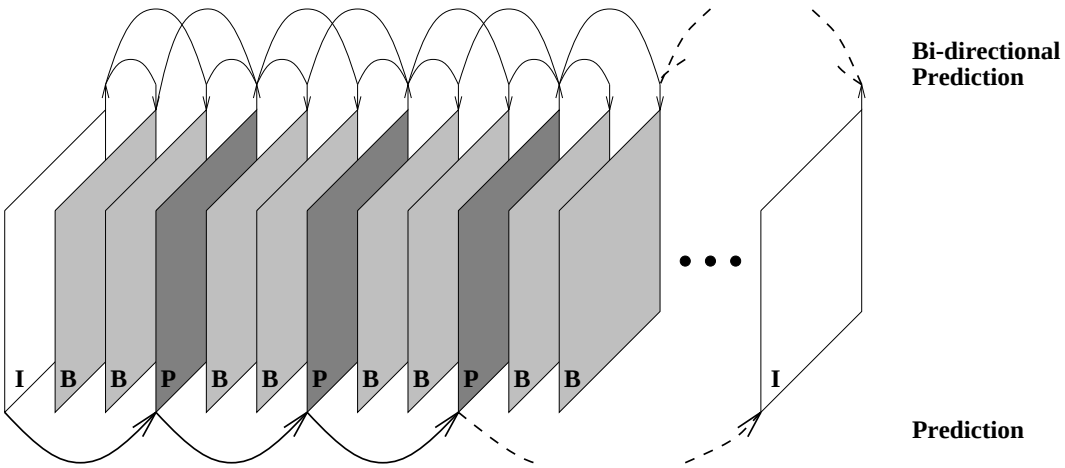

Figure 2.2: MPEG frame sequence

Usually there are 2 B-frames between each P-frame in a sequence. I-frames do appear at variable intervals. There are normally 9 P-frames between two adjacent I-frames, appearing as illustrated in figure 2.2.

The MPEG standard has a sub-group called *MPEG-AUDIO* which compresses audio data that follow the frame sequences. Standards for audio coding are PCM and ADPCM. PCM gives a data rate of 64 Kbit/sec. ADPCM has two formats, one providing 32 Kbit/sec and a second providing variable data rates. MPEG-AUDIO compresses audio already coded in these formats. Actually MPEG specifies a family of three audio coding schemes, simply called *Layer-1, -2, -3*, with increasing encoder complexity and performance. For each layer, MPEG-AUDIO specifies the bitstream format and decoder. For further reference, see [Haugen94].

2.3.4 The H.261 Standard

H.261 (Px64) is like MPEG designed for motion sequences. The purpose is telecommunication at a rate of multiples of 64 Kbit/sec (P*64) [Liou91]. Like MPEG it codes differences between adjacent frames in the sequence. However, the coding is different compared to MPEG, because H.261 does not try to predict the next frame in any way. A sequence is composed of two types of frames:

- Intra Frames: Independently coded frames, like MPEG Intra Frames.
- Inter Frames: Frames coded as the subtraction of this frame from the previous frame.

A sequence of H.261 frames is illustrated in figure 2.3. Notice that H.261 does not define audio compression.

2.4 The JPEG Standard

JPEG is an acronym for *Joint Photographic Expert Group*, the ISO standard for still image compression [Wallace91]. JPEG is designed for compressing either full-colour or gray-scale images of natural, real-world scenes. It works well on photographs, naturalistic artwork, and similar material; not so well on lettering, simple cartoons, or line drawings.

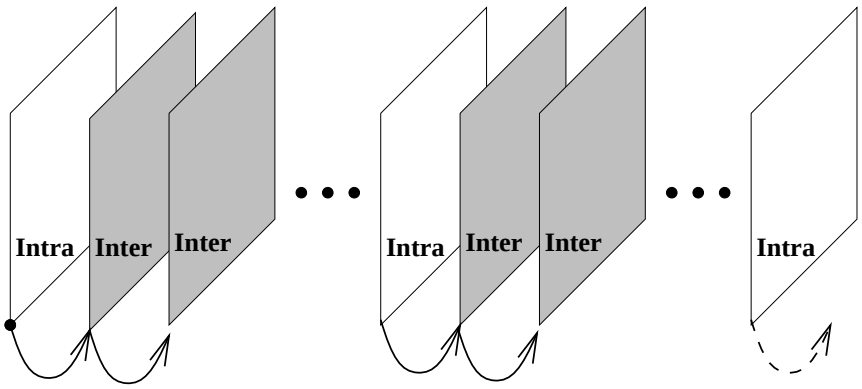

Figure 2.3: H.261 frame sequence

JPEG is a lossy method, as stated above. It is designed to exploit known limitations of the human eye, notably the fact that small color changes are perceived less accurately than small changes in brightness. A useful property of JPEG is that the degree of lossiness can be varied by adjusting compression parameters. This means that the image maker can trade off file size against output image quality. Digital video do not demand the highest quality, and therefore it is possible to obtain a high compression ratio.

Another important property of JPEG is that decoders can trade off decoding speed against image quality, by using fast but inaccurate approximations to the required calculations. This property may be used in digital video, which demands several frames to be displayed per second.

This section is based on two sources. [Wallace91] gives a short overview of the JPEG algorithm. For an extensive description of JPEG, [Pennebaker93] is recommended.

2.4.1 The JPEG Compression Family

The JPEG standard describes a family of image compression techniques rather than a single compression technique. The four JPEG modes of operation are:

1. The sequential DCT-based mode
2. The progressive DCT-based mode
3. The sequential lossless mode
4. The hierarchical mode

In this project a particular restricted form of the sequential DCT-based mode, called the *baseline* system, is used. Therefore, the other three modes will not be further described. The baseline system represents a minimum capability that must be present in all DCT-based decoder systems, and is the most popular mode used. The general terms *JPEG compression/decompression* will still be used when discussing the baseline system.

2.4.2 The Image Compression System

Image compression is the art and science of reducing (compressing) the number of bits required to describe an image. The two basic components of an image compression system

are illustrated in figure 2.4. The device that compresses the source image is called an *encoder*, and the output from the encoder is the compressed data. The compressed data may be either stored or transmitted, but are at some point fed to a *decoder*. The decoder is the device that recreates or reconstructs an image from the compressed data.

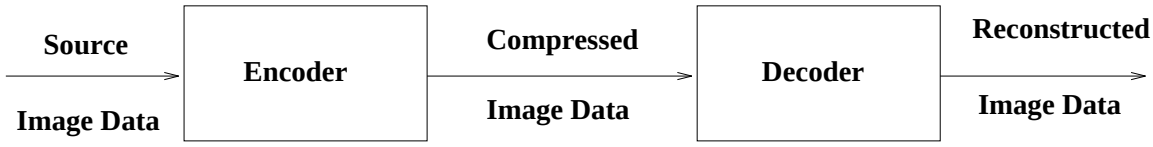

Figure 2.4: Image compression system.

2.4.3 Encoder Structure

JPEG divides the picture to be compressed into groupings of 8x8 blocks of pixels, as pictured in figure 2.5. These blocks are independently fed to the encoder. The encoder process consists of three logical steps:

1. Forward DCT-transform
2. Quantization
3. Entropy encoding

Figure 2.5 shows the encoder process. The FDCT transforms the samples from the image domain to the frequency domain. The following equations are the idealized mathematical definitions of the 8x8 (2-dimensional) FDCT:

$$F(u,v) = \frac{1}{4}C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) * cos\frac{(2x+1)u\pi}{16} cos\frac{(2y+1)v\pi}{16}\right]$$
$$\text{where: } C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u, v = 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.2}$$

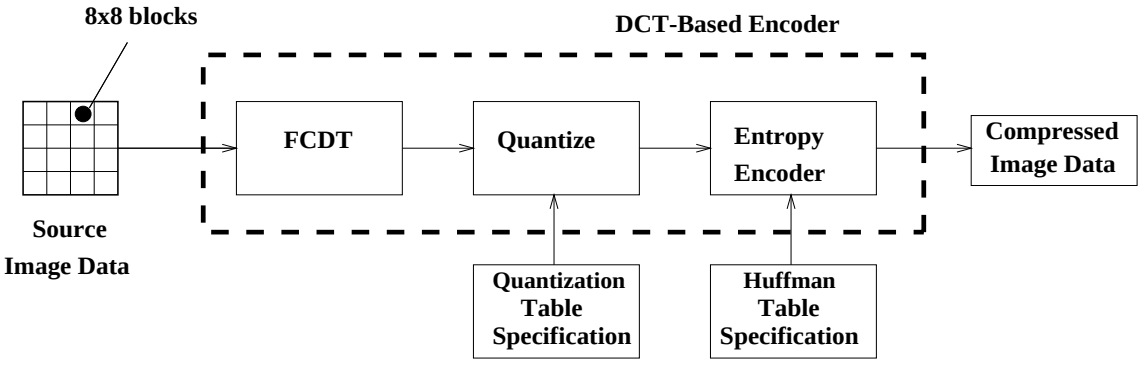

Figure 2.5: The encoder process

Because sample values typically vary slowly from point to point across an image, the FDCT processing step lays the foundation for achieving data compression by concentrating most of the signal in the lower spatial frequencies. For a typical 8x8 sample block from a

typical source image, most of the spatial frequencies have zero or near-zero amplitude, and need not be encoded. Note that the DCT equations contain transcendental functions, and consequently, no physical implementation can compute them with perfect accuracy. JPEG does not specify a unique FDCT algorithm, to preserve freedom for innovation and customization within implementation.

After output from the FDCT, each of the 64 DCT-coefficients is uniformly quantized in conjunction with a 64-element Quantization Table. This table must be specified by the application (or user) as an input to the encoder. The purpose of quantization is to achieve further compression by representing DCT coefficients with no greater precision than necessary to achieve the desired image quality. Quantization is a many-to-one mapping and therefore the process is fundamentally lossy. Actually it is the the principal source of lossiness in DCT-based encoders. JPEG has defined standard quantization tables which can be used to obtain different compression and image quality.

The JPEG standard proposes some Quantization Tables in an appendix. These tables may be scaled by a *q-factor*; ie. all the elements in the table are scaled by the q-factor. Q-factor scaling provides a way to achieve variable compression ratio. Scaling down the elements produces additional zeros in the compressed image, and this again produces lower image quality. Consequently, scaling of Quantization Tables is a way to trade off compression ratio and image quality.

After quantization we end up with 8x8 blocks of samples in the frequency domain. The DCT coefficient with zero frequency in both dimensions (the upper left coefficient) is called the *DC coefficient*, and the remaining 63 coefficients are called *AC coefficients*. The coefficients are ordered in a *zig-zag* sequence, shown in figure 2.6. This ordering helps to facilitate entropy coding by placing low-frequency coefficients, which are more likely to be non-zero, before high-frequency coefficients.

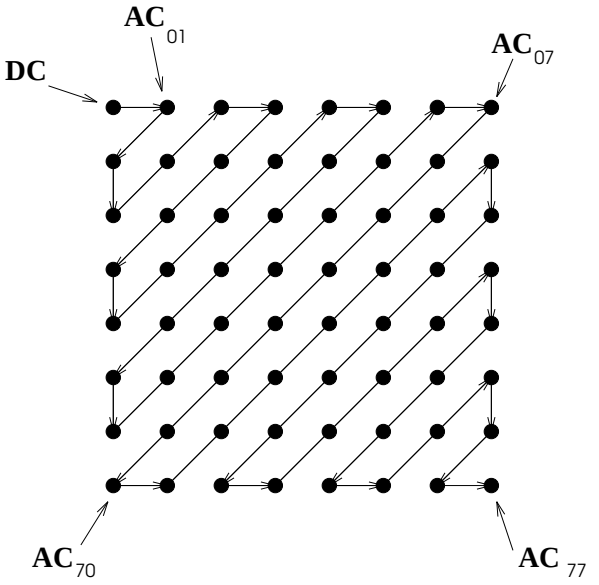

Figure 2.6: Zig-zag sequence of DCT-coefficients.

The final DCT-based encoder step is entropy decoding. This step achieves additional lossless compression by encoding the quantized DCT-coefficients more compactly based on their statistical characteristics. The JPEG proposal specifies two entropy coding methods:

- Huffman coding

- Arithmetic coding

The principle of Huffman coding is to assign shorter code words (less number of bits) to the values occurring often, and longer code words to the values occurring less frequently. This strategy clearly results in less total data compared with coding every sample by a fixed word length. Because Huffman coding encodes variable code lengths to the samples, no code can be a subset of another one. This is to be able to separate the codes when traversing sequentially the concatenated words. Huffman coding is an optimal way of coding with integer length code words.

Arithmetic coding is not used in the baseline system, in addition it is patented by IBM. Therefore arithmetic coding is not discussed in this document.

2.4.4 Decoder Structure

Decoding is the inverse process of encoding. Consequently, the three logical steps in the decoder process are:

- Entropy decoder to decode the Huffman codes
- Dequantization
- Inverse DCT-transform

The basic parts of the decoder is depicted in figure 2.7.

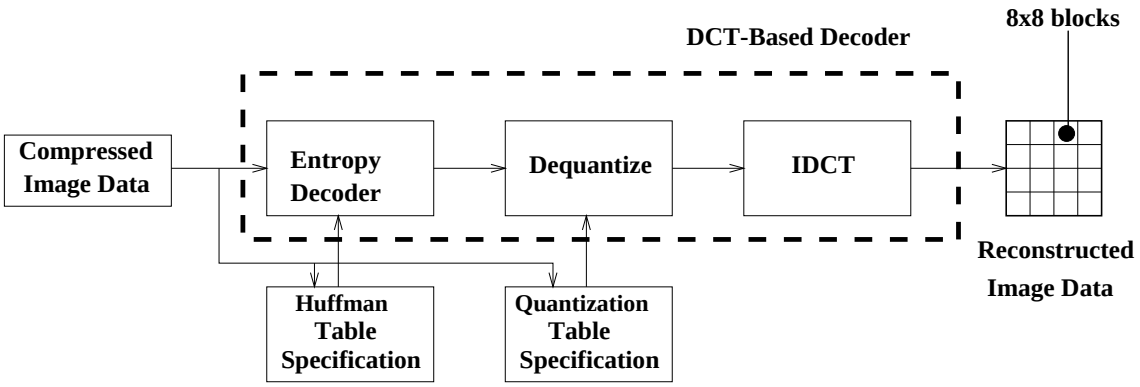

Figure 2.7: The decoder process.

The output of the entropy decoder is the quantized DCT-coefficients. The entropy decoder must use the same Huffman tables that were used in the encoder. These tables must be predefined to the application, or they can accompany the compressed image data.

Dequantization is the inverse function of quantisation, and means that every coefficient is multiplied with the corresponding element in the quantization matrix. This process retrieves the normalized DCT-coefficients appropriate for input to the IDCT step. Note, however, that coefficients which were quantized to zero, and rounding errors in the encoder, can not be corrected in this step. Quantization is the main reason why JPEG is a lossy technique (and the reason for its high compression rates, as well). Similar to the entropy decoding step, the tables used when dequantizing must be the same as the ones used when quantizing.

The last step in the decoding process is transforming the DCT coefficients back to image data. The IDCT reverses the FDCT process by taking 64 DCT-coefficients and reconstructs a 64 point (8x8 block) image samples. The equations below are the mathematical definitions of the IDCT:

$$f(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v) * cos\frac{(2x+1)u\pi}{16} cos\frac{(2y+1)v\pi}{16}\right]$$
$$\text{where: } C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u,v = 0 \\ 1 & \text{otherwise} \end{cases} \quad (2.3)$$

2.4.5 Compression and Picture Quality

For continuous colour images, the baseline technique typically produces the following levels of picture quality for the indicated ranges of compression [Wallace91]:

- 0.25-0.5 bits/pixel: Moderate to good picture quality.
- 0.5-0.75 bits/pixel: Good to very good quality.
- 0.75-1.5 bits/pixel: Excellent quality, sufficient for most applications.
- 1.5-2.0 bits/pixel : Usually indistinguishable from the original image.

Be aware that these levels are only guidelines, quality and compression can vary significantly according to the source image characteristics.

2.5 JPEG Shortcomings

The JPEG standard has a couple of shortcomings. The most obvious one is that the values of the q-factor *are not standardized*. This means that different applications may use different interpretations of q-factors. This is only a problem when the Quantization Tables are not stored within the image file, so called *abbreviated images*, and implicit Quantization Tables are assumed. To avoid confusion in this report, *highest* q-factor will denote lowest compression ratio, while *lowest* q-factor denotes the highest compression ratio. For instance, if the q-factor range is [0..100] [IJG94], q-factor 100 denotes virtually no compression, while q-factor 0 denotes the highest possible compression ratio.

The JPEG standard does not define a *file format* to store JPEG compressed images, either. A de facto standard called *JPEG File Interchange Format* (JFIF) has developed [JFIF94]. This suits most purposes, but also has some shortcomings; for instance, colormaps can not be specified within an image. Recently, the JPEG committee has published their Committee Draft version of the JPEG standard Part 3 , which will eventually become **IS 10918-3**. This draft specifies a file format called *Still Picture Interchange File Format* (SPIFF), which is functionally more general than JFIF. It remains to be seen which of the two formats that becomes the most popular.

2.6 Speeding up JPEG

When playing digital video the frames are displayed rapidly. It is therefore very important to be able to decompress the JPEG compressed data sufficiently fast. The straight forward

implementation of the JPEG algorithm is bound to work slowly, but there are ways of speeding up the decoding. The following four methods are examined in this section:

- Using hardware
- Implementation shortcuts
- Fast IDCT algorithm
- Trading off quality versus speed

2.6.1 Hardware Decoding

Designated hardware will always be faster than software for various purposes. JPEG decoding involves a lot of mathematics, and a hardware board designed to perform this job would be fast enough. The previous video player utilized an XVideo Parallax [Parallax91] board to execute the decoding. This board is devoted to JPEG decoding. The project is reviewed in section 3.10.1.

Another solution that is growing popular and common is using a Digital Signal Processor (DSP). A DSP is quite inexpensive, and is suited to perform such computations which JPEG decoding demands. However, this project aims to do the decoding without any special purpose hardware, and this subject will not be discussed any further.

2.6.2 Implementation shortcuts

The implementation of the system is significant for the performance and speed. It is perfectly possible to implement a system without considering the speed of the decoding. However, a good and effective implementation requires that the implementor is aware of parts that use the majority of time or are frequently visited. The trick is to optimize these parts. Often there are bottlenecks which delays the other processing. If these can be identified and removed, more effective code results.

The compiler may also be important. Some compilers are it optimizing ones, or have a optimizing option. These are to gain speed, but often they do not, unfortunately. The programmer may have to go down to assembly coding himself. This solution is especially applicable when bottlenecks must be eliminated, and when tuning frequently used parts.

2.6.3 Fast IDCT Solutions

The IDCT is the part of the JPEG decoding that demands most time. Mathematics show that there are faster ways of calculating the IDCT, demanding less operations. This subsection gives an overview of the most efficient algorithms currently known. I will not go into detailed descriptions of the algorithms as this is done in other texts, and is not the purpose of this work. Rather, I list the results from the algorithms and give references to the texts.

The most straightforward way to implement the DCT is to follow the theoretical equations, see section 2.4.4. The equations are *separable*, meaning they can be constructed from products of a horizontal 1-D DCT and a vertical 1-D DCT. The 2-D DCT transform done in this way would require 1024 multiplications and 896 additions for an 8x8 block.

The most popular and effective algorithms known are:

- Arai, Agui and Nakajima [Arai88] has developed a separable 1-D approach that uses 80 multiplications and 464 additions, which is competitive with other approaches. This algorithm has properties that are advantageous for scaled DCT implementations.
- Vetterli [Vetterli88] describes a 2-D DCT that requires 104 multiplications and 460 additions.
- Feig [Feig90] describes a 2-D solution that uses 54 multiplications and 462 additions and 6 multiplications by 1/2 (bitwise right shift).

These are all very good solutions, and the video player developed in this project is based on the first algorithm.

2.6.4 Quality vs Speed Tradeoff

It is possible to obtain faster decoding if poorer picture quality is tolerable. The obvious solution is to use low quantization factor, resulting in high compression ratio and low picture quality. The decoding will go faster compared to using higher quantization factors. It is also possible to scale down the original image, gaining speed in this way.

Using fast IDCTs often means poorer quality of pictures. Fast IDCTs are approximations to the original IDCT, and of course lowers quality. The user must decide if this compensates for the speed-up.

Walmsley et.al. [Walmsley94] describe a way to prune the IDCT process. Their argument is that when there usually are so many zeros in the DCT coefficients, why bother doing the whole 8x8 IDCT transform. Sufficient quality can be achieved by doing a NxN transform, where $N < 8$. This technique obviously saves computation efforts. Actually a speed up of over 40 percent can be achieved with this technique. Again, the user must decide if the picture quality is sufficient.

2.6.5 Summary

When hardware devices can not be utilized, other techniques have to compensate. Because the majority of time is used in the IDCT, using a fast IDCT is a demand. As stated previously, the user has to determine if the quality is sufficient for the quality vs speed tradeoffs. In addition speed can be gained by clever implementations and tuning of the code. Combinations of the techniques listed is possible in order to gain the desired speed.

2.7 Motion JPEG

The JPEG standard is defined only for still images. Digital video utilizing JPEG compression and decompression therefore consists of a sequence of independent frames, called *intra* frames. Digital video by means of JPEG is frequently referred to as *Motion JPEG* or M-JPEG. A Motion JPEG frame sequence is depicted in figure 2.8.

Note that M-JPEG is not standardized. M-JPEG is simply the application of JPEG compression to individual frames of a video stream. Various vendors have made their own design choices, and in the absence of any recognized standard, the products differ. The resulting video files are not compatible across different vendors. As far as I know, there is no current standardization going on for a motion JPEG video.

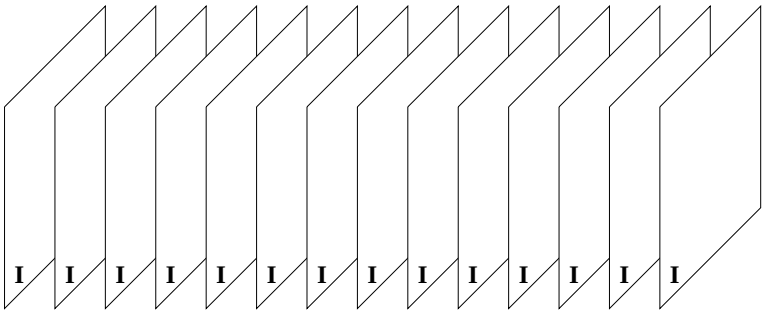

Figure 2.8: Motion JPEG frame sequence

As mentioned in section 2.3.3, MPEG is the recognized standard for motion picture compression. It uses many of the same techniques as JPEG, but adds inter-frame compression to exploit the similarities that usually exist between successive frames. The advantages of M-JPEG compared to MPEG are:

- MPEG requires much more computation to generate the compressed sequence, since detecting visual similarities is hard for a computer.
- M-JPEG is more flexible than MPEG. The frames are independent of each other, and this makes it easy to move from frame to frame. In addition, M-JPEG is not as wounderable as MPEG when frames are lost, for instance during network transmission.
- It is difficult to *edit* an MPEG sequence on a frame-by-frame basis, since each frame is intimately tied to the ones around it. It is this latter problem that has made M-JPEG methods rather popular for video editing products.

On the other hand, MPEG is superior to M-JPEG on other areas:

- MPEG achieves higher compression ratios than M-JPEG because of the inter-frame compression. Consequently, M-JPEG video files are larger than MPEG files.
- Decompression of M-JPEG frames takes longer time compared to MPEG.
- M-JPEG does not define any methods for representation or compression of audio data.
- As mentioned above, there exists no M-JPEG standard.

M-JPEG usually demands dedeicated hardware boards in order to perform video playback in real time. It is generally recognized that software decompression of JPEG images can not be performed in real time on most of today's computers. However, as hardware tends to become cheaper and faster, hardware decompression devices may be included in computers. This would promote use of M-JPEG video as a de facto standard for certain application areas, for instance editing of video.

Obviously, there should have been a standard for M-JPEG video; M-JPEG is not likely to disappear even though MPEG is growing more popular. Nevertheless, as long as there is no standard for M-JPEG video, applications should be made as general as possible. An M-JPEG application should not be dependent of the file format; this makes it easy

to rewrite only the file part of the application in order to change to another file format. However, this is not really relevant for commercial products, because the user do not get a chance to modify the source code. Thus, motion JPEG keeps floating among various vendors and implementations.

Chapter 3

Network Transmission and Protocols

An objective of this project is to investigate a network solution to send video data from a remote host to a video client. Compression is important for network transmission as well. Transmission of raw image data might produce several Mbytes of data per second (see section 2.1), and most networks do not have such a capacity. Consequently, compressing the video data is required prior to transmission.

The main issue of this chapter is the choice of a network protocol. Section 3.1 contains a discussion of general requirements to the protocol for transmission of *continuous media* (CM), especially video and audio data. Subsequently some protocols are briefly described and evaluated to see whether they are suited for transmission of video.

It is important to learn what others have accomplished in the field of digital video. This chapter is concluded with an overview of some prior and related work, with respect to network transmission and the digital video issues discussed in the previous chapter. Especially, the *VoDoo* video system, which this work is based on, is described here.

3.1 Introduction

After capturing video it is usually stored on a physical device, for instance a hard disk. It is desirable that video clients not directly mounted to the remote device, are able to receive the video data and play video. Network communication is necessary for video clients to receive audio and video frames transmitted from a remote host.

A network protocol must be defined in order to communicate through the network. The protocol defines the rules for data transmission, and the choice of protocol influences the performance of the system significantly.

This section examines some general demands for protocols which are used for transmitting video. This is followed by a discussion of relevant protocols and whether they are suited for video data transmission.

3.2 Transmitting Video Data

Transmission of video data should be as *transparent* as possible to the user. This means that the transmission of data is not visible to the user, it should seem like data is fetched

from the local hard disk. The user should only specify where the data remains, ie. the name of the host computer. However, the transparency objective is hard to fulfill, because network transmission is not as reliable an fast as receiving frames from the local hard disk. Figure 3.1 depicts the network transmission process.

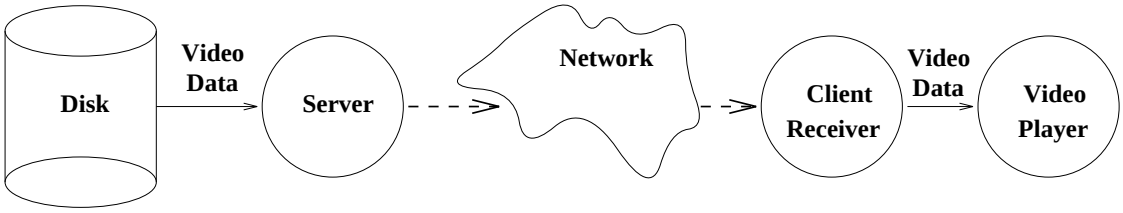

Figure 3.1: General network transmission

When playing video, frames must be received at a similar rate which they were recorded to convey the original meaning from the sequence. Thus, if frames were recorded at 20 frames/sec, the video client must receive 20 frames/second. If it is impossible to maintain the desired frame rate, or the client can not handle the speed due to processing speed, the frame rate must be lowered. A solution is to skip frames in the sequence. For instance, every second frame could be skipped, decreasing the original frame rate to the half. The frames then have to be displayed for double the time to maintain the original meaning and speed. This results in more visible 'jumps' in the motion of the sequence.

Network transport imposes delays; time is spent to physically transfer data from one computer to another. Delays would not be a serious problem if they were constant. Unfortunately, delays vary during transmission. *Delay jitter* is the variation of the delays with which packets travelling on a network connection reach their destination [Ferrari91]. Delay jitter occurs of numerous reasons:

- Network traffic load
- Losing frames
- Errors in frame data

Network traffic load is determined by all the concurrent transmissions on the actual network, and can not be controlled by a single user. If a lot of other people are using the network, the time spent to transport data increases. Similarly, if there is little traffic on the network, transmission delays will decrease. Networks have a limited *bandwidth*, meaning that there is an upper limit for how much traffic that can be handled.

If a *retransmission* protocol is used, losing frames and errors in frame data also causes delays. When a frame does not arrive at the client side, the server retransmits the frame. Consequently, this frame arrives later than it should. Errors in frame data could also require a retransmission, and a delay is imposed. However, image data are error tolerant. A few errors in the pixel values have very little effect on the total picture. Retransmissions on account of frame errors should be performed only when errors have a serious impact on the picture.

For good quality of reception, continuous media (audio, video, image) streams require that delay jitter be kept below a sufficiently small upper bound. An objective is to reduce the number of retransmissions, and the network protocol should help accomplishing this. Note that the protocol can not control the network traffic except the transmission the application is doing.

Buffering is a way to compensate for delay jitter. Buffering means that frames are transmitted from the host and stored at the client, before the client really needs the frames. Instead of receiving frames directly from the host, the video player uses the frames that are held in the buffer. Buffering relies on that the server can predict the sequence of frames that the player needs. The server is one step ahead of the player, and thereby the delay jitter should be balanced.

Playing video is an interactive process. The user can play, reverse, rewind, fast forward, step frames etc. As long as the user stays in one mode (for instance *play* mode), the server knows the sequence of frames, and buffering works well. When the user changes mode, the frames in the buffer becomes inappropriate. The server has to fill the buffer with frames that is necessary for the new mode. The time interval from the user changes mode until the buffer is filled with the appropriate frames, delays have an impact on the playing process. There are two strategies to handle this problem:

- *Optimistic strategy:* Playing starts as soon as the first frame in the new frame sequence arrives, hoping the buffer will be filled succeedingly.

- *Pessimistic strategy:* Playing suspends until the buffer is filled with a sufficient number of frames of the new frame sequence.

The first one should be chosen if the delays are short and delay jitter is small. The pessimistic strategy should be used when the network is unreliable.

The size of the buffer is determined according to the average delay of network transmissions. Longer delays and jitter demands a bigger buffer. Ideally, an infinite buffer should be utilized. However, the buffer size is also restricted by the storage capacity on the client side. The buffer size is therefore determined according to both the network delays and the storage space at the client.

As described above, the protocol should reflect the type if the network. Networks can be classified as *Local Area Networks* (LAN) and *Wide Area Networks* (WAN). A LAN is generally more reliable and faster than a WAN. Consequently, delay jitter is smaller and errors occur less frequently on LANs. If the data is transmitted through a WAN, the protocol has to be more reliable compared to a LAN protocol.

The video data consists of two separate streams: one image frame stream and on audio frame stream. Audio and video frames belong together as pairs. The syncronization of audio and video frames are significant to convey the meaning of the video and for the sequence to be pleasant to watch. Audio frames are 'master' frames when playing, the image frames have to synchronize with them. The priority is put on audio data, they have to be sent continuously. Image frames can be discarded, but is is much more annoying to loose audio frames. On account of this, different protocols for audio and video might be worth considering.

Reliability demands more data transmitted per frame. A bigger frame spends longer time to be transmitted. Obviously you have to trade off overhead of transmission against speed. Also, if one uses a protocol applying retransmissions and handshaking (ACK/NACK), the network load increases. Transmission speed vs reliability overhead has to be considered when choosing the protocol.

3.3 Protocols

Network protocols define the rules for data transmission. ISO[1] has defined a standard reference model. The reference model, called *Open Systems Interconnection* (OSI), has adopted a layered approach [Halsall92]. The complete communication subsystem is broken down into a number of layers, each of which performs a well defined function. The OSI model is shown in figure 3.2.

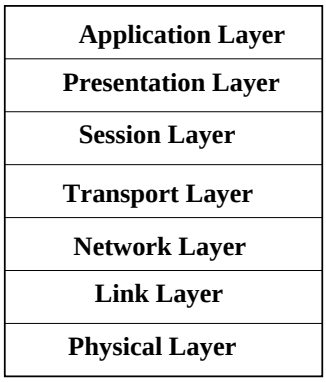

Figure 3.2: The ISO Reference Model for Open Systems Interconnection

Each layer has a well defined interface between itself and the layers immediately above and below. Consequently, the implementation of a particular layer is independent of the implementation of the other layers, it is only dependent on the interface of the lower layer. Each layer provides a defined set of functions to the layer immediately above.

This work is concerned with layer four (transport layer) to seven (application layer). The transport layer provides end-to-end message transfer, error control, fragmentation and flow control. The application layer provides the user interface, like file transfer, document and message interchange, job transfer and manipulation. The presentation and session layers are not discussed further.

Internet is a non-commercial network that connects many universities, organisations, companies and other people all around the world. The Internet protocol standard is the TCP/IP protocol suite, discussed in the next subsection. TCP/IP is a best-effort protocol suite, meaning that no guarantees are given with respect to timely delivery. Delays wary according to the network load and routing decisions made by the protocol. It is therefore generally agreed that a new generation of protocols are necessary for transmission of Continuous Media (CM) [Shepherd91], [Wolfinger91]. Typical examples of CM are video and audio data. Characteristic of CM data is that upper bounds are imposed on delay and delay jitter. In addition, it is often possible to predict which data which is to be transferred.

A lot of research is currently being performed on transmission protocols for CM. The next subsection examines the Internet standard TCP/IP protocol suite. Section 3.5 describes the Real-Time Transmission Protocol, a protocol under development by an Internet Work Group. The University of Berkeley has proposed a protocol suite for transmission of CM, and this is described in section 3.6. Finally, the Experimental Stream Protocol version 2 (ST-II) is reviewed in section 3.7.

[1] International Standardization Organization

3.4 The TCP/IP Protocol Suite

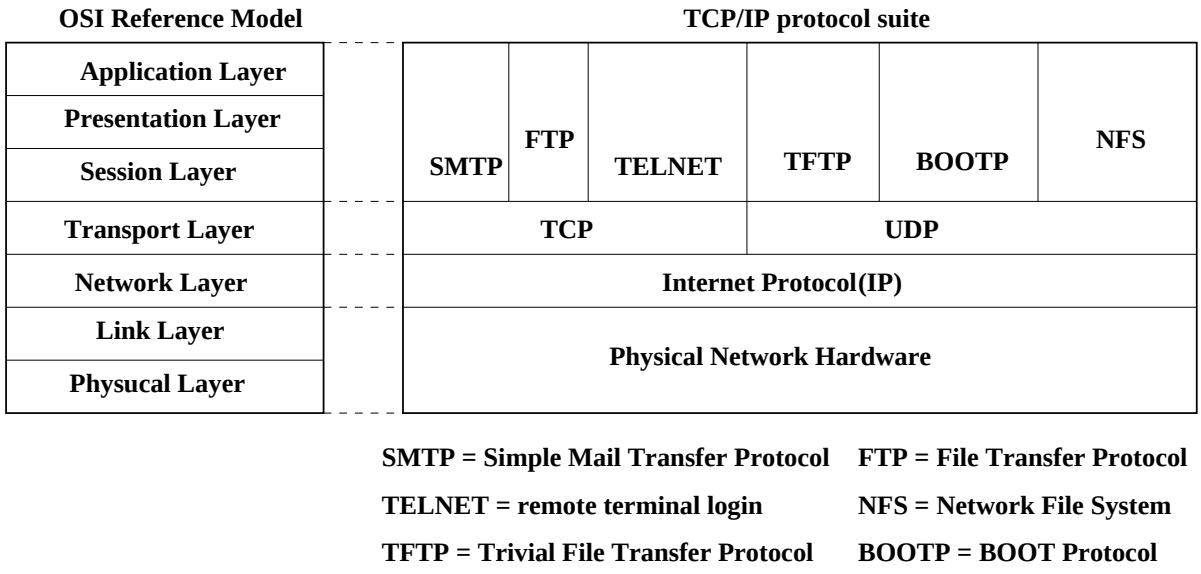

Figure 3.3: The TCP/IP protocol suite

The protocol suite used with the Internet is known as *Transmission Control Protocol/Internet Protocol* (TCP/IP). The TCP/IP protocol suite relates to the OSI model as shown in figure 3.3 [Washburn93]. In TCP/IP, all protocols above the transport layer are application protocols. The transport layer contains two layers:

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

TCP and UDP provides the interface for application protocols. In addition they can be used for transmissions themselves. The next two subsections discusses TCP and UDP as protocols for transmission of video data.

3.4.1 The UDP protocol

The User Datagram Protocol provides a connectionless unreliable service [Washburn93]. It allows data to to be transmitted to a computer without the need to establish a connection. Single datagrams are sent to a remote node without any requirement for responses to indicate that the datagram has arrived. In other words, no *handshaking* is provided (no acknowledge). UDP provides no error detection or correction; the reliability of service is solely dependent of the basic error performance of underlying protocols. No flow control is offered, the application using UDP has to supervise the retransmissions and transmission speed.

Clearly, an application utilizing the UDP protocol for video data transfer must add some basic functionality:

- Flow control: Error detection, retransmission, acknowledge.
- Transmission speed.

This functionality must be achieved by a control protocol in the opposite direction of the UDP protocol.

The advantages of UDP are:

- Little overhead data per datagram.
- No network bandwidth used for flow control, error correction and detection, and acknowledge.

The disadvantages are:

- The service is unreliable; there is no guaranty that datagrams arrive at the receiver or correct data.
- The application must do the flow control and any error detection and correction.

3.4.2 The TCP protocol

The TCP protocol is the second protocol in the transport layer of the TCP/IP protocol suite. Contrary to UDP, TCP provides a connection-oriented service [Washburn93]. A connection is like a data pipe that runs between two points. TCP has all the features to necessary to provide a reliable service between two computers. In achieving reliability, TCP adds a significant amount of overhead to manage flow control, timers and connection management facilities. It has more overhead than UDP in terms of both the processing power required and the size of the protocol headers it uses. In addition, TCP also uses more of the network bandwidth than UDP.

TCP has all the features needed for transmitting video data. The data is correct, and flow control is used to prevent a transmitter over-running a receiver due to resource limitations. The question is whether TCP provides sufficiently high data rates. This is discussed in the next subsection.

3.4.3 Data Rates of TCP and UDP

In order to choose between TCP and UDP, the requirements for data transmission rates must be considered. Generally, UDP is fast and unreliable, while TCP is slower but reliable.

A project at the University of Mannheim had performed some research on network protocols [Lamparter91]. The project implemented a system for storing, transmitting and presenting digital movies in a computer network. The project is more thoroughly reviewed in section 3.10.

The transmission protocol used in the project is called *Movie Transfer Protocol* (MTP). In one experiment, TCP was used for digital movie transmission. The throughput was about 150 kbits/s. Utilizing UDP instead, they measured a throughput of 2Mbit/s. Here approximately 10 % of the frames were lost in the network. The conclusions are that the TCP protocol is not suitable for movie transmission. In particular the sliding window flow control and the go-back-n error recovery in TCP are inappropriate. The transmission of digital movies requires protocols that have a very low delay jitter and provide high throughput.

Another project at Berkeley, University of California, did some research on network protocols as well [Rowe92]. They came up with the same conclusions as the Mannheim project: TCP is generally too slow to handle the requirements of continuous media.

3.5 Real-time Transport Protocol

The *Real-time Transport Protocol* (RTP) is an application layer protocol with reference to figure 3.2. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data [RTP94]. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services.

The data transport is augmented by a control protocol, the *Real-time Transport Control Protocol* (RTCP), designed to provide minimal control and identification functionality. The *RTP control protocol* provides two functions:

- Monitoring the distribution of data, which is performed by the RTCP sender or receiver of report packets.
- Conveying minimal session information, to provide support for 'loosely controlled' sessions

RTP and RTCP are designed to be independent of the underlying transport and network layers. For further information about RTCP, see [RTP94].

Note that the protocol is still under development by the *Audio/Video Transport working group* within the *Internet Engineering Task Force*. Consequently the protocol is likely to change.

RTP is suitable for continuous media, like audio and video. However, RTP itself does not provide any mechanism to ensure timely delivery or provide any other *QoS*[2] guarantees. Rather, it relies on lower-layer services to do so. The protocol does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence.

The sequence numbers included in RTP allow the end system to reconstruct the sender's packet sequence. RTP is designed to run on top of a variety of network and transport protocols, for example, IP, ST-II or UDP. RTP transfers data in a single direction, possible to multiple destinations is supported by the underlying network.

3.5.1 RTP Data Transfer Protocol

The RTP data transfer protocol has a fixed header and a possible extension. The fixed header format is depicted in figure 3.4. The total length of the header varies from 96 bits to 576 bits, depending on the number of *CSRC*s (see below).

The fields have the following meaning:

- **Type (T)**: Identifies the type of the RTP packet.
- **Padding (P)**: If the padding bit is set, the packet contains one or more padding bits at the end. The very last octet gives the number of padding octets.

[2] Quality of Service

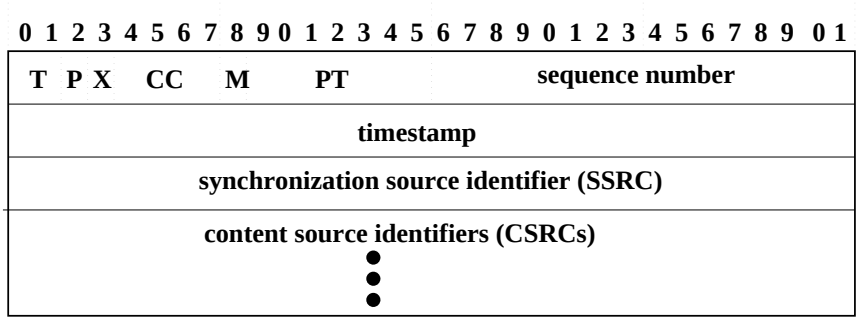

Figure 3.4: The RTP fixed header format

- **Extension (X)**: The header is followed by a header extension (see below).
- **CSRC Count (CC)**: The number of CRCSs.
- **Marker (M)**:The interpretation of this bit is defined by a profile.
- **Payload Type (PT)**: The type of the data (*payload*) in the packet.
- **Sequence Number**: The sequence number increments by one for each packet sent.
- **Timestamp**: The timestamp is incremented by a clock frequency.
- **SSRC**: Synchronization Source Identifier. The value is chosen randomly, with the intent that no two synchronization sources within the same channel have the same SSRC value.
- **CSRC**: Zero or more Content Source Identifier. The number of CSRCs is given by **CC**. CSRC are inserted only by bridges.

The *RTP header extension* has a format pictured in figure 3.5. The extension is provided for extending the RTP data packet header in an application-specific or profile-specific way. If the *X* bit in the fixed RTP header is set, the RTP header is followed by a variable-length header extension. The header extension contains a length field which counts the number of extension octets, including the 32 bits extension header.

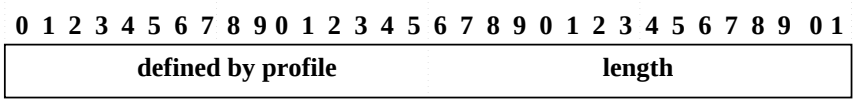

Figure 3.5: The RTP header extension format

3.6 The Tenet Protocol Suite

The Tenet Group of University of California at Berkeley has done, and is currently doing, a lot of research on networks and transmission of CM. The *Tenet* protocol suite for CM is designed, implemented and tested [Banerjea94]. The Tenet protocol suite proposes a new type of service, called *guaranteed performance service*. The suite includes:

- Real-Time Internet Protocol (RTIP) for real-time transmission (corresponding to IP in TCP/IP).
- Real-Time Message Transport Protocol (RMTP), for real-time transmission of messages.
- Continuous Media Transport Protocol (CMTP) for transmission of CM.
- Real-time Channel Administration Protocol (RCAP) for setup and administration of RMTP and CMTP connections.

Figure 3.6 shows how the protocols relate to each other and how they relate to TCP/IP.

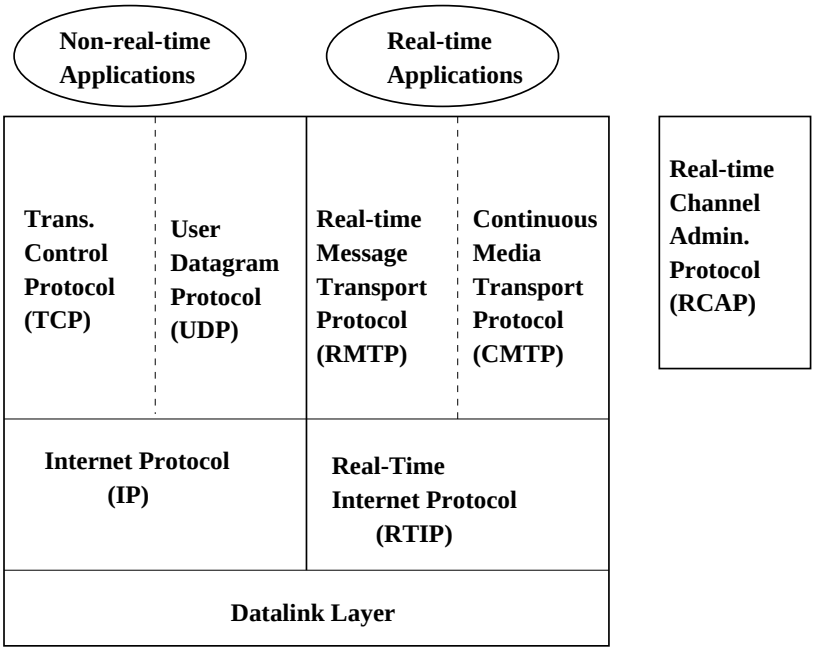

Figure 3.6: The Tenet protocol suite

RTIP is the network layer data transport service. RTIP performs rate control, jitter control, packet scheduling, and data transfer functions [Zhang92]. It provides a host-to-host, simplex, sequenced, unreliable, and guaranteed-performance packet service. Note that a packet is not guaranteed to be delivered. It may be dropped for two reasons: the packet may be corrupted during transmission, or there might not be enough buffer space to store the packet. The client data is not checksummed;they may get corrupted due to transmission errors.

When setting up a RTIP connection, four performance parameters can be requested by the client:

- Delay bound
- Delay jitter bound
- Delay violation probability bound
- Buffer overflow probability bound

In addition, a traffic description is described by means of specifying some traffic parameters:

- Minimum packet inter-arrival time
- Average packet inter-arrival time
- Averaging interval
- Maximum packet size

These two sets of parameters are used to set up a network connection. If the network is able to satisfy the parameters, the protocol guarantees that the performance guarantees requested by the communication client are satisfied as long as the client obeys the restrictions implied in its traffic description.

In order to give the performance guarantee, the protocol uses a reservation-oriented architecture. This means that bandwidth is reserved along all nodes on the connection path. Consequently, the CM connection is not affected by other transmission on the network path, and performance guarantees can be provided. [Ferrari92] gives a description of the design and implementation of RTIP.

RMTP is one of the transport layer protocols of the Tenet protocol suite. It is *message-oriented*, ie. responsible for such end-to-end functions as message fragmentation and reassembly, regulation of traffic according to traffic specifications and optional checksumming. Is based on RTIP, and therefore provides a simplex, end-to-end, unreliable, in-order, and guaranteed-performance message service. More information on RMTP is given in [Ferrari92].

The other protocol on the transport layer of the Tenet suite is *CMTP*, It is *stream-oriented*, meaning that it is designed to transport a continuous sequence of data units. As CMTP is also based on RTIP, it provides the same guaranteed-performance as RTMP.

When setting up a CMTP connection, the client must provide *Quality of Service* parameters (bounds) and traffic characteristics. CMTP processes are set up both the receiver(CM_r) and sender (CM_s) side. The sender (C_s) writes data to a shared buffer with CM_s. CM_s sends the data through the network to CM_r, which writes the data into a buffer. This buffer is shared with the receiver (C_r), which reads data from the buffer [Ferrari92b]. This scheme is depicted in figure 3.7.

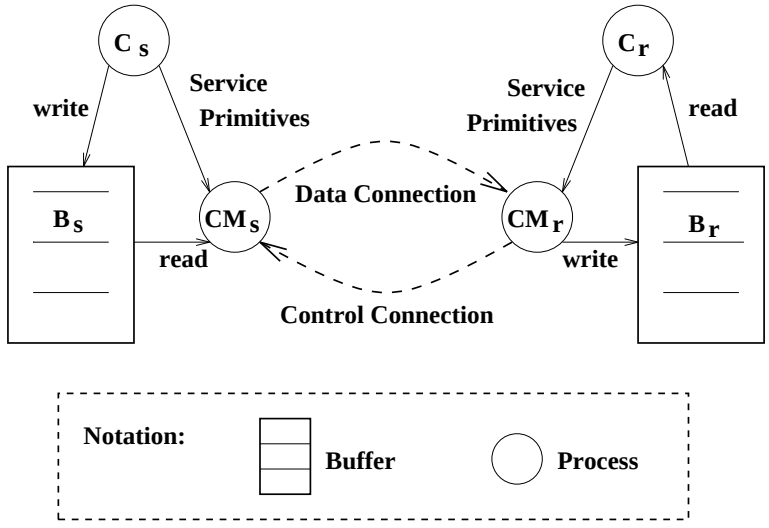

Figure 3.7: The CMTP communication process

The *RCAP* protocol is responsible for setting up the network connection for CMTP RTMP.and RTIP. RCAP provides control and administration services for the Tenet suite. It is actually RCAP which performs the reservation when a network connection is set up.

The RCAP services are:

- Channel establishment
- Channel teardown
- Status inquiry for RTIP, RTMP and CMTP.
- Managing resources along the channel path.
- Admission control to enable the network to provide end-to-end performance guarantees.

The reservation approach is based on worst-case analysis. Consequently, RCAP provides hard guarantees. [Banerjea92] describes the RCAP protocol.

3.7 ST-II

The *Internet Stream Protocol, Version 2* (ST-II or ST2) is a connection-oriented internetworking protocol that operates at the same level as connectionless IP (the *network* layer in the OSI model). The protocol document is an Internet-Draft, a working document of the Internet Engineering Task Force, and the protocol specification might therefore change in subsequent versions [ST-II94]. ST-II is still not standardized or widely used, but may become an Internet standard in some years.

ST-II has been developed to support the efficient delivery of data streams to single or multiple destinations in applications that require guaranteed data throughput and controlled delay characteristics. The main application area of the the protocol is the real-time transport of digital audio and video packet streams across internets.

ST-II can be used to reserve bandwidth for multimedia streams across network routes. This reservation, together with appropriate network access and packet scheduling mechanisms in all nodes running the protocol, guarantees a well-defined quality of service to ST-II applications.

The ST-II relation with the OSI model and the TCP/IP suite is shown in figure 3.8. ST-II consists of two protocols:

- Stream Protocol (ST) for the actual data transport.
- Stream Control Message Protocol (SCMP) for all control functions, mainly resource reservations.

ST is simple and contains only one Protocol Data Unit (PDU) that is designed for fast and efficient data forwarding in order to achieve low communication delays. SCMP is quite complex with functions for setting up connections, messages for maintaining connections, and functions for closing connections. ST and SCMP have a similar relationship as IP and ICMP in the TCP/IP protocol suite.

A connection in ST-II is called a *stream*, which forms the core concepts of ST-II. Streams are established between one *origin* (sending node) and one or more *targets* (receiving

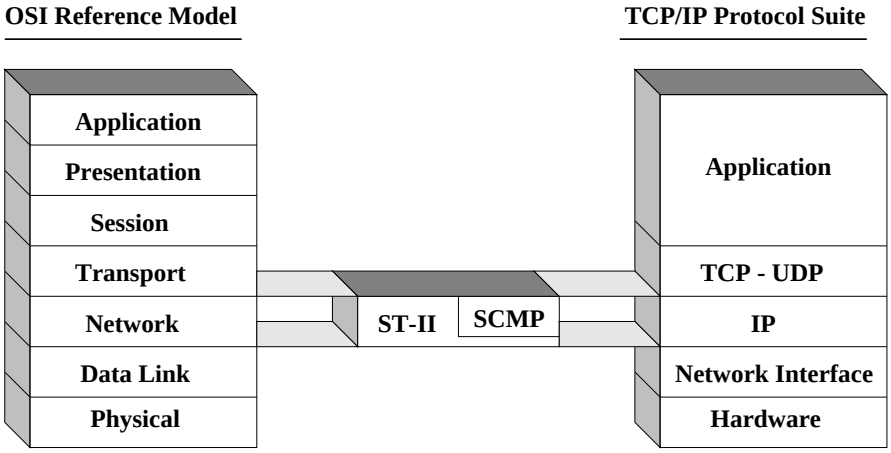

Figure 3.8: ST-II protocol, related to OSI and TCP/IP

nodes) during stream setup to transmit application data. An ST-II stream may be built over many intermediate nodes to reach the targets and has the form of a directed acyclic graph. Nodes in the tree represent so-called *ST Agents*, which are entities executing the ST-II protocol. Figure 3.9 illustrates the concept of streams.

Data transfer in ST-II is simplex. Data transport through streams is very efficient; ST-II puts only a small header in front of the user data. Efficiency is also achieved by avoiding fragmentation an reassembly on router nodes. A maximum transmission unit for data packets is negotiated at establish time; the upper protocol layers must provide data packets of suitable size to ST-II.

Streams are maintained using SCMP messages. Typical SCMP messages are:

- *CONNECT* and *ACCEPT* to build a stream.
- *DISCONNECT* and *REFUSE* to close a stream.
- *CHANGE* to modify the stream characteristics.
- *NOTIFY* messages to inform ST agents of changes.

As a part of establishing a connection, SCMP negotiates quality-of-service parameters for a stream. These parameters form a *Flow Specification* (FlowSpec) which is associated with the stream. Different versions of flow-specs exist, but typically they contain (similar to RTIP, section 3.6:

- Average throughput
- Maximum throughput
- Maximum end-to-end delay
- Maximum delay jitter

Depending on the success of the reservations on the ST-Agents along the stream, a stream is either established or refused. Negotiation is used to check whether the FlowSpec can be

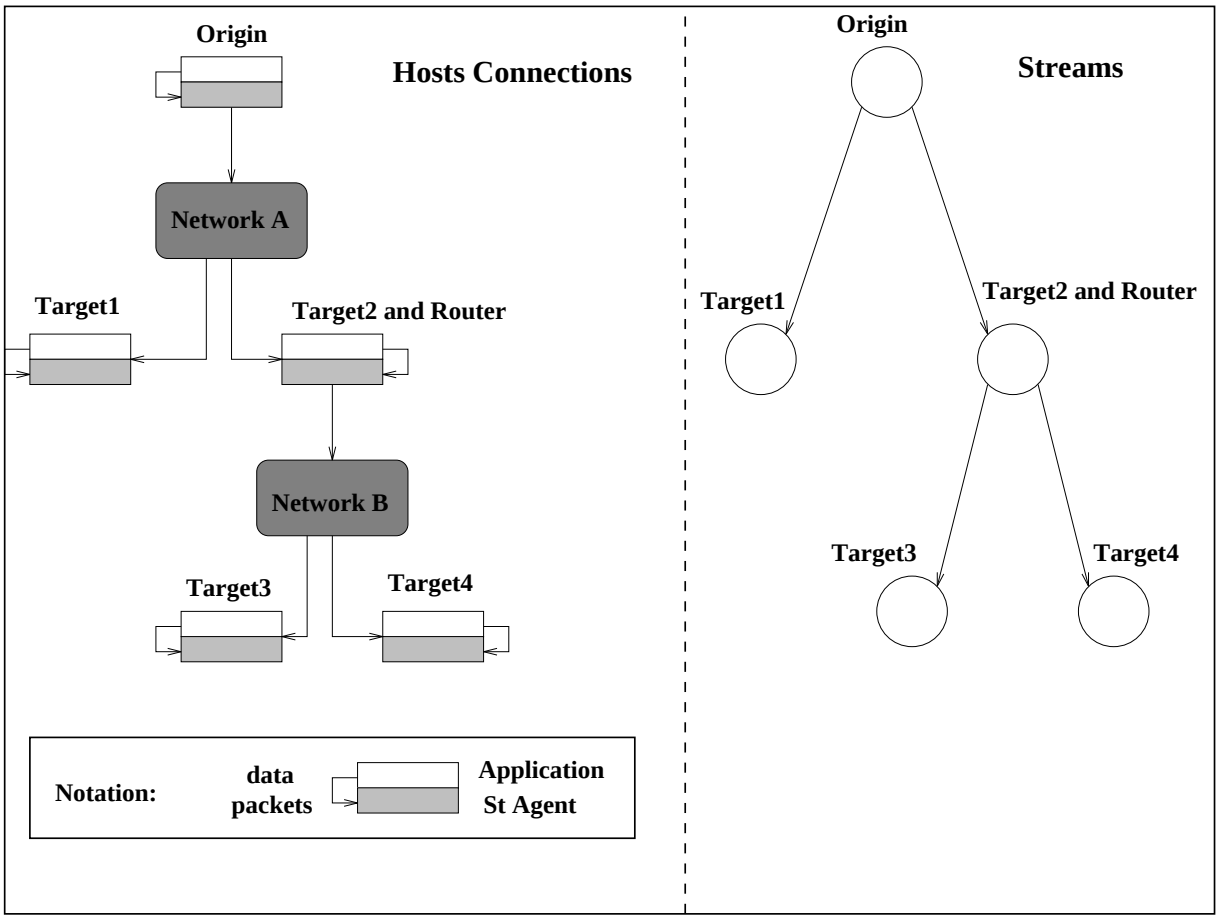

Figure 3.9: The Stream Concept [ST-II94]

accepted along the stream. If a stream is established, the parameters are guarantee the performance of transmission.

ST-II is an unreliable protocol, meaning that packets do not always arrive at their targets, and data may be corrupted during transfer. If reliability is necessary, upper layer protocols must perform any error checking and retransmissions. However, video and audio are *error tolerant*, and some errors might be acceptable. As ST-II is a network protocol, transport protocols are built on top of it for the transport of data (similar to TCP and UDP being on top of IP, see section 3.4). The *Heidelberg Transport Protocol* (HeiTP) is an example of such a transport protocol built on ST-II [Delgrossi92]. The Heidelberg project is reviewed in section 3.10.

Finally, it should be mentioned that ST-II is oriented to the Internet architecture. Both IP and ST-II employ the IP addressing schemes to identify different hosts. Thus, ST-II co-exists with TCP/IP, and can be used parallel to TCP/IP. ST-II is *not* a replacement for TCP/IP, but a supplement for transmission of continuous media data.

3.8 Asynchronous Transfer Mode

Asynchronous Transfer Mode (ATM) is a relatively new, fast packet switching technology. ATM is developed to support the requirements broadband applications. The data rates achieved with ATM ranges from 1.5 Mbps to 150 Mbps. This section briefly describes

some important properties of ATM in relation to transmission of Continuous Media. This section is entirely based on the diploma thesis of Johnsen [Johnsen94].

3.8.1 The ATM Concept

The ATM concept is based on two well-known techniques:

- Packet switching [Halsall92]
- Statistical multiplexing [Halsall92]

Some modifications has been applied to the packet switching scheme:

- No error control or float control is performed by the links in the ATM network.
- Connection-oriented on the lowest level; all the information is transmitted in a *virtual circuit*.
- The ATM packet (called cell) length is short (predefined length), so no fragmentizing or re-assembly of packets is necessary.
- The cell header contains a minimum of information.

The length of an ATM cell is specified to 53 bytes, containing a 5 byte header and 48 bytes payload; this is shown in figure 3.10. The main function of the cell header is to route the cell from sender to receiver. Other information in the cell header includes type, priority and checksum.

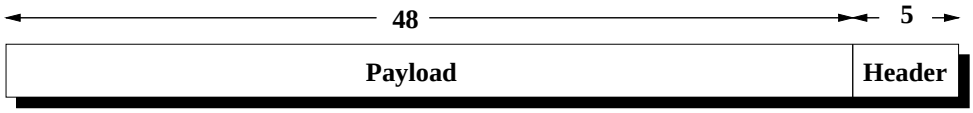

Figure 3.10: The ATM cell

3.8.2 The ATM Reference Model

Similar to the OSI model, there is a reference model for ATM, which is also layered. In addition the model is divided into multiple planes. The ATM Protocol Reference Model (PRM) is depicted in figure 3.11.

The three planes considered are:

- The *user plane* transports user information.
- The *control plane* takes care of control signals.
- The *management plane* is responsible for maintenance of the network and operational functions.

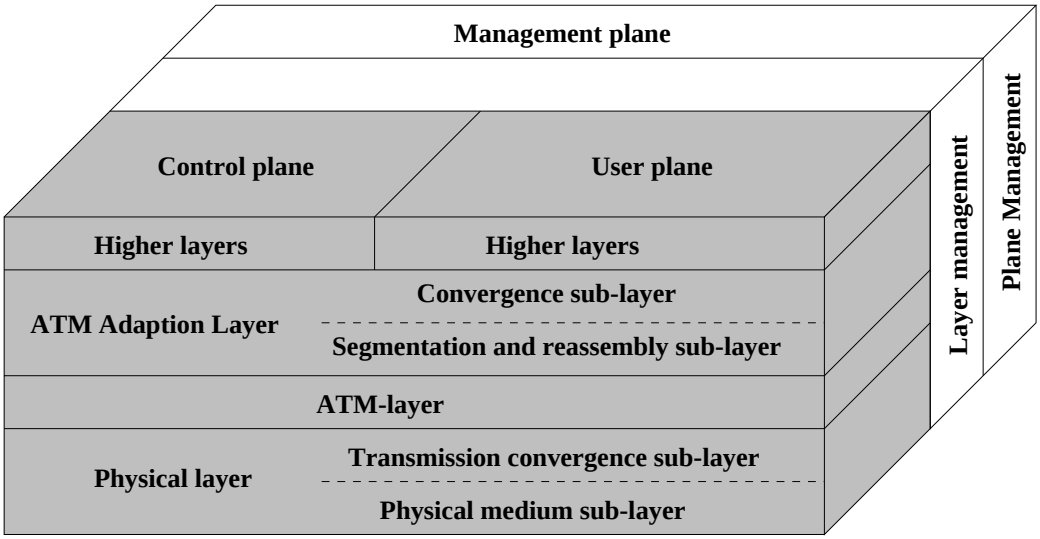

Figure 3.11: ATM Protocol Reference Model.

In addition there is a third dimension called *plane management* which is responsible for administration of the various planes.

The physical layer of PRM is responsible for correct transmission of data on the physical device. Consequently this layer is dependent of the physical medium (optical, electronic).

The ATM Layer supports transparent transmission of ATM layer Service Data Units (ATM SDU). Prior to transmission of data, a connection has to be established, and a *traffic contract* is negotiated. A traffic contract consists of a QoS-class, traffic parameters, and other parameters. The network traffic through the ATM nodes should be in accordance with the traffic contract.

The *ATM Adaption Layer* (AAL) is the interface between the higher layer services and the ATM layer. The main purpose of AAL is to extend the services which the ATM layer provides. Since ATM is supposed to support a broad selection of of applications, and four service classes are identified. This is shown in figure 3.12.

	Class A	Class B	Class C	Class D
Time relation between sender and receiver	Necessary		Not necessary	
Transmission rate	Constant	Variable		
Connection mode	Connection-oriented			Connection-less

Figure 3.12: ATM Adaption Layer classes

For transmission of digital video, class B is particularly interesting. The characteristics of class B are:

- There is a time relation between sender and receiver.

- The transmission rate is variable.
- Connection oriented.

The ATM technology is still quite new, and research is still being performed. However, ATM seems suitable for transmission of CM data; ATM provides QoS-parameters and traffic parameters, variable transmission rate, time relations between sender and receiver, and high transmission rates. ATM is not very common outside research environments today, but in the near future, use of ATM will grow heavily. It should also be noted that the TCP/IP protocol suite can run upon the ATM-layer, which promotes a co-existence of ATM and the TCP/IP standard.

3.8.3 Protocol Summary

The TCP/IP protocol suite is the current standard on the Internet. However, TCP/IP does not provide support for continuous media, and is generally not sufficient when requirements such as fast transmission and low delay jitter are necessary. The UDP protocol has been used in som multimedia projects, and proved faster than the TCP protocol. Thus, achieving an acceptable speed is possible, but jitter and loss of frames is still a problem. It is clear that the network community needs new protocols to handle continuous media properly.

The two protocols RTIP and ST-II are both implemented and tested, and they both seem to be well-suited for continuous media transmission. RTIP and ST-II are based on the same concepts:

- Resource reservation
- Quality of service parameters and negotiation
- Guaranteed performance
- Specially for CM transfer

But none of the protocols are standard on the Internet for CM transmission. It seems reasonable to think that ST-II is going to become a new CM transmission standard, since it is currently moving in the Internet standardization process.

The RTP protocol does not provide QoS parameters as do RTIP and ST-II; RTP relies on lower-layer services. However, because RTP is independent of underlying layers, it could easily be ported to run on top of protocols which provide QoS guarantees. The *VoDoo video system*, reviewed in section 3.9, utilizes the RTP protocol for data transfer.

ATM technology is also a growing standard, and ATM has many properties in commom with RTIP and ST-II. ATM is actually going to be installed on computer belongning to the Database Group in the near future. This gives the opportunity to experiment and test ATM technology in connection with CM applications.

3.9 The VoDoo Video Player

In a project at *The Norwegian Computing Center*[3] during 1994, a digital video player called *vshow* has been developed [VoDoo94]. In my work I have used this player as a basis

[3]Norwegian:Norsk Regnesentral

for implementation (see appendix A for the copyright conditions). This section briefly describes the VoDoo video player system.

VoDoo is a simple "Video on Demand" system for workstations. It consists of a viewer application and a video server. The video images are stored as JPEG compressed frames, and vshow relies on a hardware board in order to decompress video frames in real-time. The hardware board used is an XVideo Parallax board [Parallax91] which compresses and decompresses JPEG images "on the fly", as required. The VoDoo system is also capable of capturing video (recording video played by an analoge video player) by means of a *video grabber* process. However, as recording of video is not relevant for this project, I will not describe this function.

3.9.1 User Interface

The user interface of vshow is mostly similar to common analoge video players. It consists of a window where the video stream is displayed, and a control panel where the user controls the playing of video beneath. The user interface is shown in figure 3.13.

Figure 3.13: The user interface of vshow.

The buttons of the control panel consists of, from left to right:

- **Play/Pause**: Play or pause a video film. When vshow is in play mode, the pause symbol is shown. Similarly, when vshow is in pause mode, the play symbol is displayed.
- **Step Reverse**: Step one frame in the reverse direction. Double-clicking on this button puts vshow in a "slow reverse" mode.
- **Step Forward**: Similar to the above, except that the direction is forward.
- **Fast Reverse**: Fast rewind without audio, various speed.

- **Fast Forward**: Fast forward without video, various speed.

In addition the control panel contains a *slider device* (scale) which allows the user to jump directly to some part of the video film. It also moves while playing, ie. it shows the relative position within the video film continuously during playback.

Keyboard characters can also be used to control the playback. The following keys are recognized:

- '**q**': Quit.
- '**p**': Play/Pause.
- '**f**': Fast Forward.
- '**r**': Fast Rewind.
- '**s**': Step Forward.
- '**b**': Step Reverse.
- '**t**': Toggle control buttons.
- '**a**' or '**T**': Toggle time display.
- '**e**' or '**S**': Open control window.
- '**w**': Print current location.

The vshow application is usually invoked with a single command line argument. The filename of the video file is given as an input parameter, and vshow will try to open this file locally. If a *hostname* is specified as well, vshow will try to connect to the remote host, and video data is transferred over the network by the vodoo protocol (see the next section).

3.9.2 VoDoo Architecture

As described above, vshow either reads video data from a local disk or receives the data from a remote host. Figure 3.14 shows the architecture of the system with a local connection. The input to the player consists of one data stream which contains both audio data and video frames. The video frames are transferred to the Parallax board, which decompress the JPEG frames and displays them on the display window. Audio data is written to the audio device output buffer.

A *client-server* architecture is used when transmitting frames through the network. The vshow application starts a *video server* on the remote host, which sends data to the vshow client. This is depicted in figure 3.15. The internal structure of vshow is a little different in this case in order to receive frames transmitted by the video server. A network protocol has been designed to control and transmit data, called the *VoDoo protocol*. Data is sent by the RTP protocol [RTP94], and control is accomplished by a custom ASCII protocol. For further information about the VoDoo system, see [VoDoo94].

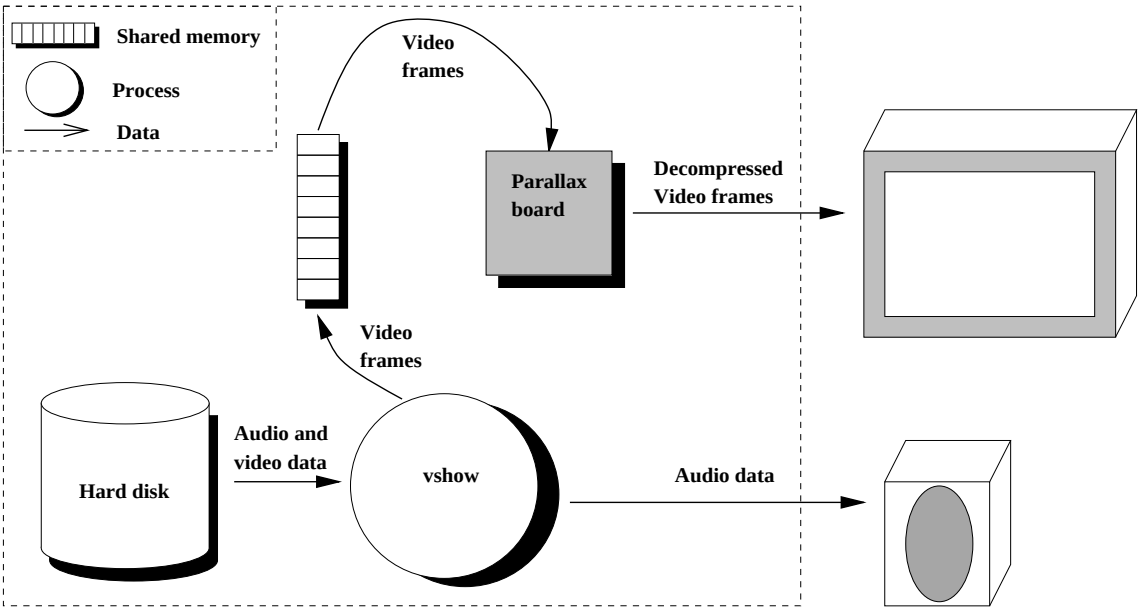

Figure 3.14: Local architecture of the VoDoo system.

3.10 Related Work

A lot of research has been performed on digital video issues, including capture and playback, efficient transmission of video data and compression. The previous section presented the VoDoo system; this section briefley reviews some more work related to this diploma thesis.

3.10.1 show_video

Sigurd Skeide developed a video system in his diploma thesis in autumn 1993 at NTH [Skeide93]. This system is related to the VoDoo system described above; both systems are based on the same code and uses a Parallax board to compress and decompress JPEG frames.

The system is capable of:

- Record video and store it on hard disk.
- Play video, accessing frames either stored on local disk or accessed through a network connection. The player is called *show_video*.

The user interface of show_video is analogue-like, ie. it has functions like play, pause, fast forward, fast reverse and step functions. The video player is currently being used in cooperation with a video description database at the Database Group at NTH. A family of tools have been developed to facilitate annotations on video, video browsing and search in the video description database. Further information about these systems can be found in [Holte94] and [Bekkadal94].

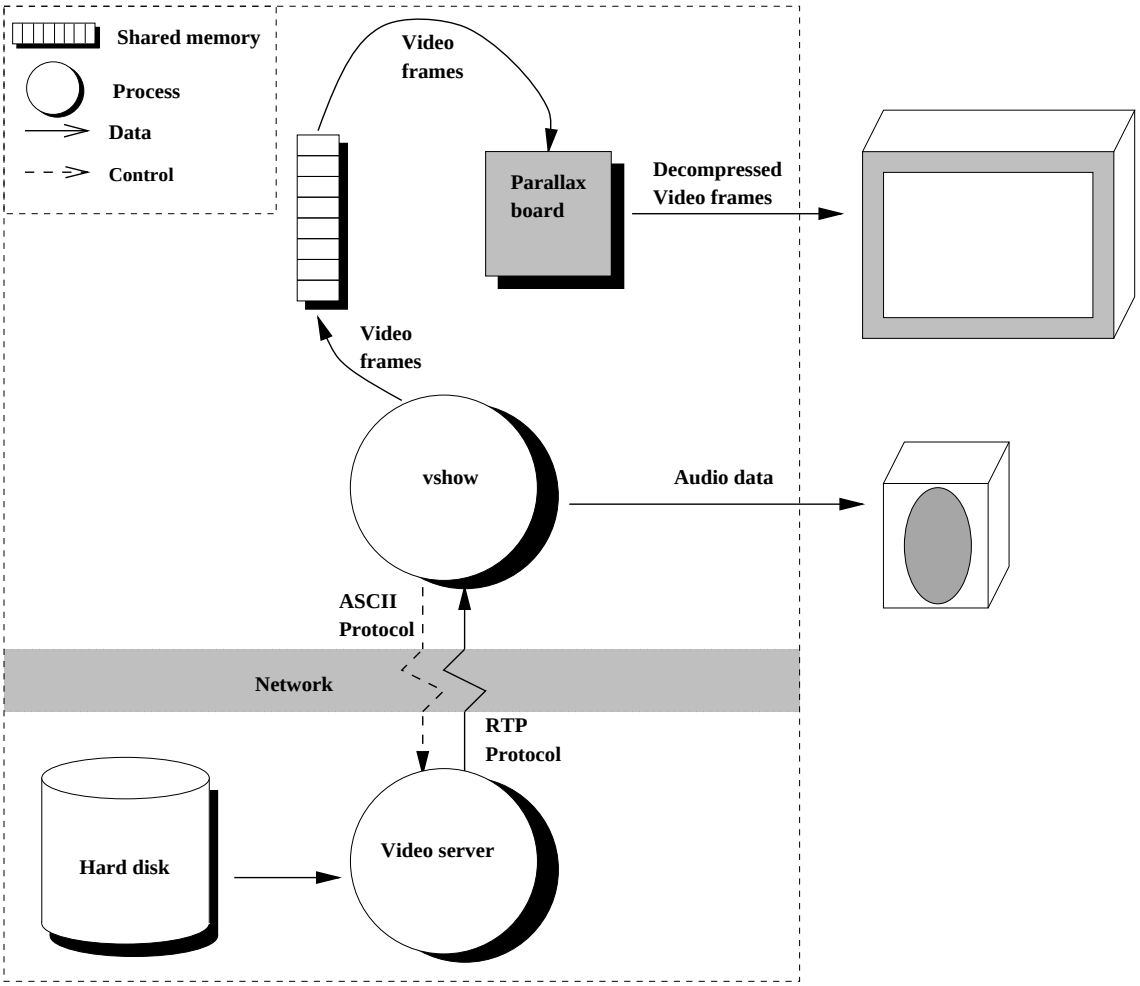

Figure 3.15: Network architecture of the VoDoo system.

3.10.2 Berkeley Plateau Multimedia Research Group

The *Berkeley Plateau Multimedia Research Group* is doing research on systems infrastructure and applications of continuous media (digital video and audio). Some of their projects are:

- MPEG-1 video decoder and parallel encoder. The well-known *mpeg_play* application is freely distributed, and is the de facto non-commercial player of MPEG movies. The decoder decompresses and plays video streams of 160x120 in real-time (30 frames/second) on current generations RISC architecture workstations. The system does not handle real-time synchronization or audio streams. Information about the MPEG-1 video decoder and encoder can be found in [Rowe92] and [Rowe93].

- The *Continuous Media Player* (CMPlayer) is a network playback system that handles Motion JPEG video or MPEG video stored on a file server. MJPEG decoding is performed by a hardware chip, for instance the Parallax board [Parallax91], while MPEG video is performed by the MPEG-1 decoder mentioned above. The CMPlayer can the RTIP protocol described in section 3.6 or a best-effort protocol implemented on UDP and TCP (see section 3.4) for data transmission. A frame rate of 24 frames/second and image size of 320x240 has been achieved over an Ethernet

or the Internet. The synchronization is implemented by synchronizing the system clocks on the server and the client using the *Network Time Protocol*.

This group has performed a lot of research on continuous media, and some recent reports can be found in [Rowe94a], [Rowe94a] and [Rowe94b]. They also cooperate with the *International Computer Science Institute* in developing the *Tenet Real-Time Protocol Suite*. The Tenet protocol suite is reviewed in section 3.6.

3.10.3 Heidelberg Transport System

At the IBM European Networking Center, the *Heidelberg Transport System* (HeiTS) has been designed and developed. The goal of HeiTS is to transport multimedia data over heterogeneous networks in accordance with their real-time requirements. HeiTS uses the ST-II protocol described in section 3.7.

HeiTS consists of a family of protocols and systems which support transmission of multimedia data. A resource manger called *Heidelberg Resource Administration Technique* (HeiRAT) guarantees delivery parameters like throughput, delay and error handling. The *Heidelberg Buffer Management System* (HeiBMS) is used to manage buffers needed for data transport minimizing data copies. ST-II makes use of *Heidelberg Data Link* (HeiDL) to send and receive packets over the different networks, while the *Heidelberg Transport Protocol* (HeiTP) is used within HeiTS as the transport protocol on top of ST-II. In addition, a special memory manager (MEMMGR) and the *Heidelberg Operating System Shield* (HeiOSS) are included in the system. This is all summarized in figure 3.16.

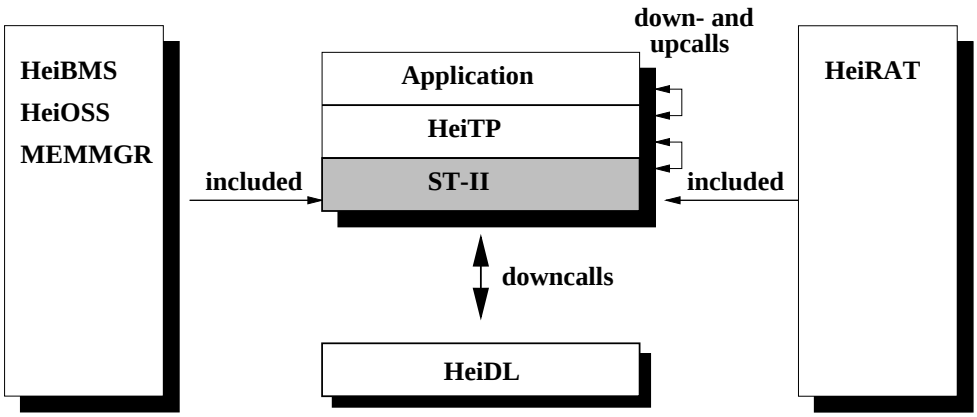

Figure 3.16: Overview of HeiTS components.

More information about the Heidelberg projects can be found in [Delgrossi92].

3.10.4 XMovie

As mentioned in section 3.4.3, a research group at University of Mannheim, Germany, is doing research on continuous media. A system called *XMovie* has been developed. XMovie is capable of transmitting and receiving audio and video data through a network connection, and play video streams on the client side. A protocol called *Movie Transport Protocol* (MTP) is put on top of IP/UDP to transport data. A *Forward Error Correction* mechanism and packet priority is used to prevent packet loss and delay jitter during network transmission. order to display images, the X-Server of the X Window system [Young90]

has been extended; the X Window System does have any support for real-time display of images. Further information can be found in [Lamparter91] and [Keller93].

3.10.5 SoftPEG

Paradise Software has developed UniFlix, a software toolkit for decompresion and display of digital video. UniFlix is designed to be used with Motion JPEG video sequences, either by means of dedecated hardware or software only. The *SoftPEG* tool supports software decompression and playback of MJPEG files at frame rates of 5-30 frames per second. SoftPEG contains a very specialized JPEG decompression algorithm which is capable of performing over 144000 8x8 DCT transforms per second on a SPARCstation 10. To obtain frame rates of 30 frames per second, SoftPEG must run on a SUPERSPARC computer with grayscale output. Color output produces from 3 to 15 frames per second, dependent of the computer. One audio track can be played in synchronization with the movie. UniFlix also supports network transmission of video using a client-network architecture.

Note that Paradise Software is a commercial company, and therefore the UniFlix Toolkit is a commercial product as well. Consequently it is impossible to investigate the internal design and implementation of UniFlix and SoftPEG. The above information about UniFlix can be found in [Paradise94].

Chapter 4

System Requirements and Discussion

The previous chapter investigated the basics of JPEG compression and decompression, and essential protocol and network subjects. Prior to presenting system interface and design, some general and functional system requirements with respect to a software video player are determined. Fundamental subjects concerning system design has to be discussed and reasoned.

This chapter first looks at the genral requirements for a digital video player. Subsequently, issues like displaying images, determination of the video frame rate, the network protocol and synchronization are discussed. Also, some post processing techniques necessary after the JPEG decompression of images, are presented.

4.1 General Requirements

The purpose of the digital video player is to play video sequences on the display screen of a computer. Audio data associated with the video stream must be played in synchronization with the video stream. Audio and video data are read from files of predefined format.

Generally the video playback frame rate should be kept constant. This is, however, not possible for a software based video player, as explained in section 4.3. Consequently, a requirement is that the frame rate should be allowed to vary within defined bounds. The frame rate should not increase or decrease very often during playback.

The playback of video must be performed with the same frame rate of which the video was captured. A software video player can not usually cope with real-time playback of video. The *skip frame* approach presented in section 4.3 is suggested to handle this problem. Skipping of frames should only be performed on video data; audio data should be played continuously.

The video player should accept various q-factors (see section 2.4). This means that the video player must be able to scale the quantization matrices to suit the compressed video frames. All frames of video must be compressed using the same q-factor. In addition, the video player should be able to display different video frame sizes.

The above are the general requirements for a software digital video player. The subsequent sections discuss some specific problems and issues concerning a digital video system.

4.2 Drawing Images

Digital video consists of a sequence of digital images which are drawn on the screen succeedingly. When drawing images on a display screen, several system-specific problems have to be investigated. I have worked under the *X Windows system*, using the *XLib* [Young90] and *Motif* [OSFPROG91] libraries. The discussions in this section is based on this environment, but other systems are likely to have similar properties.

This section first discusses color, and the *visual* concept is introduced. Image formats, ie. how images are represented in order to be displayed, are also presented. Finally, the speed of drawing images on a display screen is discussed.

4.2.1 Visuals and Colormaps

A *visual* is an XLib concept, used to specify the color handling for a drawable[1]. Every window has a an associated visual, usually inherited from its window parent. The purpose of a visual is to hide the hardware specific details of the display hardware, and thereby support portability of applications.

Important data carried by a visual are:

- Depth, ie. the number of bits that represents an individual image pixel.
- The order of RGB values and number of bits representing each R, G and B value. These are only relevant when using DirectColor or TrueColor visuals (no colormapping).
- Visual class (explained below).

A *colormap* is a color lookup table, which is used to translate image pixel values into the visible colors on the screen. Colors are specified by their RGB values as explained in section 4.6.1. However, instead of specifying the RGB values directly, an *index* into the colormap is used. This is depicted in figure 4.1.

There are two properties that distinguish colormap types:

- Colormaps can be either *read-only* or *read-write*. A read-only colormap has predefined color values which can not be changed by the user, only reading is allowed. Read-write colormaps has entries which can be modified by the user in addition to reading. Consequently, users can define their own colors when read-write colormaps are supported.
- Colormaps are characterized by either having a *single* index or *decomposed* index for RGB values. A decomposed index means that image pixels are broken down into three separate colormap indices, one for each primary color (R,G and B). A single index into the the colormap means that one image pixel indexes one RGB triplet in the colormap.

The number of *planes* of a display screen determines the number of colors the screen can display *simultaneously*. Common display hardware today have 8 planes, providing 2^8=256 different colors. As memory hardware is becoming cheaper, 24 planes display hardware grows more popular. With 24 bits per pixel, it is possible to display over 16 million distinct

[1] An XLib drawable is either a window or a pixmap.

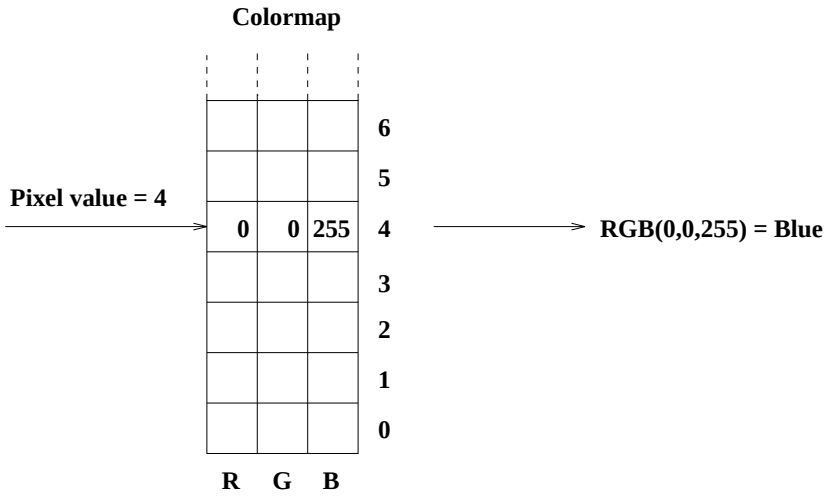

Figure 4.1: Pixel value to RGB mapping with a colormap

colors (2^{24}=16777216). A colormap this big can not be handled, therefore a decomposed index into the colormap is used. This means that each primary R,G and B value has its own colormap, resulting in three distinct colormaps.

The number of colors available to an application is dependent of the number of entries into the colormap. Even though a display screen has 24 planes, an application using a 256-entry single index colormap can display only 256 colors. Consequently, it is perfectly possible to simulate an 8-plane display screen on a 24-plane display screen.

Most computers support only one *hardware colormap*, that is, the colormap that remains in memory and is currently available for applications to use. Obviously the hardware colormap is a scarce resource; if two or more applications require all the colors in the colormap, and they want different sets of colors, only one application gets what it wants. A common work-around is to *share* colors, meaning that several applications use the same colors. However, when an application has specific color requirements, this is not a perfect solution. Therefore, XLib supports *virtual colormaps*. A virtual colormap is created by the application, belongs only to the application, and the application can define the colors it needs in this colormap. The window manager swaps the virtual colormaps in and out of the hardware colormap as appropriate. This means that when a specific application is active, its associated virtual colormap is swapped into hardware. When the application is not active anymore, the virtual colormap is swapped out, and the old colormap is swapped in again.

Now we return to discussing visuals. A visual belongs to a *visual class*; the classes are distinguished by their colormap type. XLib provides six visual classes, listed in table 4.1.

When using a visual with decomposed colormap, the image data is stored with the R,G and B indexes separated. The computer imposes restrictions on the sequence of this data, eg. that data is stored *... 0BGR 0BGR 0BGR...*, where the initial 0 byte is for alignment to computer word size. The application must process the image data such that the format is according to the computer specifications.

For an application to be portable to various display hardware, several visual classes must be supported. The appropriate visual should be selected by the application, hiding this from the user. On the other hand, the user may want to force a particular visual class, and this should be possible as well. Therefore the selection of visual must cover both the

Visual Class	Read/Write	Indexing
DirectColor	Read/Write	Decomposed
TrueColor	Read only	Decomposed
PseudoColor	Read/Write	Single index
StaticColor	Read only	Single index
GrayScale	Read/Write	Single index
StaticGray	Read only	Single index

Table 4.1: Visuals supported by XLib

display hardware requirements and the user demands. Naturally the user can not obtain a visual that is not supported by the display hardware.

Ideally, applications should use the default hardware colormap, avoiding swapping virtual colormaps in and out of the colormap hardware. However, to achieve high quality display of colors, custom virtual colormaps must be used. A custom colormap is the best solution regarding the range of colors. Unfortunately, the swapping of colormaps would cause a "flickering" on the screen every time the colormap is swapped in and out of the hardware colormap (unless the computer supports multiple hardware colormaps).

4.2.2 Drawing speed

XLib provides standard functions to draw image on the screen. The `XPutImage()` function is used to display image data contained in user memory on the display screen. `XPutImage` is a general function that works with all visual classes. A possible problem is that the drawing is slow, particularly when large images are to be displayed. Fast drawing of images is important when we are working with digital video. There are two negative effects that may occur when drawing of images is slow:

- The frame rate is low.
- Watching the video stream is not pleasant, because the drawing is done in a line-by-line manner. When drawing is slow, this 'partial updating' of the image is visible to the user.

MIT[2] provides the *Shared Memory Extension to X* (MIT-SHM) [MIT-SHM]. This is designed for fast drawing of images. MIT-SHM uses shared memory between the X client and the X server to achieve higher speed. The function `XShmPutImage` is the equivalent to `XPutImage`. MIT-SHM is perfectly appropriate for digital video, providing a significant speed-up. There are however two drawbacks with this solution. (1) Not all computers support MIT-SHM, and (2) the extension only works on a local connection (not on remote logins). Consequently, a digital video application should support both conventional XLib functions and MIT-SHM, using the latter when possible.

The difference in speed between conventional XLib calls and MIT-SHM varies, but in general, the difference increases when the size of the image increases. In section 8.1 experiments are performed to measure the time spent for the two methods for various image sizes.

[2]Massachussets Institute of Technology

4.3 Frame Rate

When playing video films, the play speed usually is measured in frames per second (fps), called the *frame rate*. The original frame rate is a property of the particular video film, and is determined when the film is captured. The rate of which the individual frames of a video sequence is captured becomes the original frame rate. Real-time video should have a frame rate of 20-30 fps.

The *vshow* application (see section 3.9) uses a hardware board to decompress JPEG compressed frames in real time, and the board also handles the drawing of decompressed images on screen in real-time. These two functions are essential to achieve real-time performance of digital video.

A software based video player is bound to be slower than a hardware based one (at least on most of today's computers). As established above, the two processes that are more time consuming in a software implementation are the decompression of JPEG compressed frames, and the displaying of decompressed frames (images). This inherent property leads to *skipping of frames*. Instead of displaying all frames in a video sequence, only some of the frames are displayed; frame sequences are skipped with a defined interval count.

The resulting frame rate is calculated according to the following equation:

$$\text{Frame Rate} = \frac{\text{Original Frame Rate}}{\text{Skip Frames} + 1} \tag{4.1}$$

In order to illustrate the skipping of frames, an example is appropriate. A video film's original frame rate is, say, 20 fps. If the decompression of frames uses 200 ms, it is only possible to achieve a frame rate of 5 fps. Consequently, the video player displays every fourth frame, skipping the frames in-between. This is shown in figure 4.2. It is naturally possible to play all the frames, but real-time synchronization would be lost, displaying 20 adjacent frames during 4 seconds (compared to 1 second in real-time). The visual result is slow-motion film, and this is not what we are looking for.

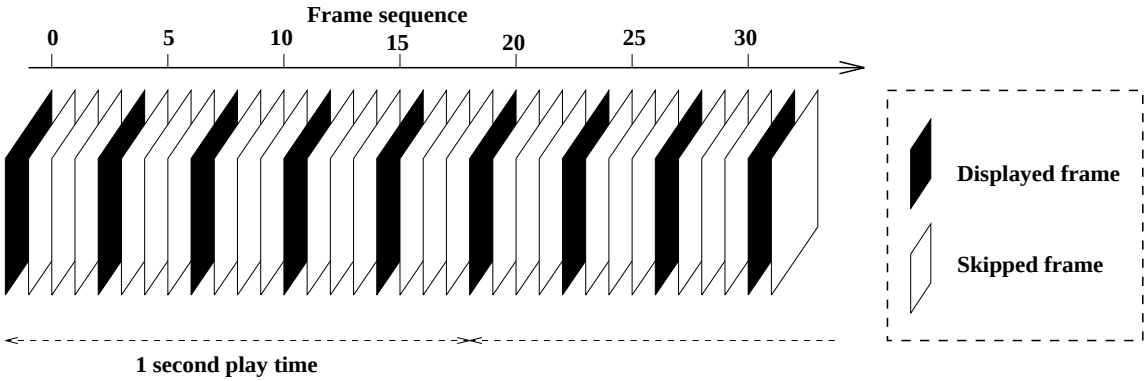

Figure 4.2: Skipping of frames (Original frame rate: 20 fps. Video player frame rate: 5 fps. Skip frames: 3 frames)

Naturally the skip frame approach does not produce as high video quality as the real time equivalent. It is however the only solution if audio-video and real time synchronization is the aim. Synchronization is discussed in section 4.4.

The major obstacle with the skip frame approach is determining the number of frames to skip. This number must be computed from two factors when reading the video stream

from a local connection:

- **Decompression delay**: The average time spent decompressing frames.
- **Display delay**: The average time spent displaying frames.

When a network connection is used to transmit frames, an additional factor has to be taken into account:

- **Transmission delay**: The average time spent transmitting a frame.

The algorithm for determination of the skip frame value is presented in section 5.2. However, the problem is not yet resolved. We also have to consider the variations in the delays, called *delay jitter*. There are three types of delay jitter for a video player:

- *Decompress delay jitter*:
  The software decompression time is not constant; it varies mainly because of the following reasons:
  - JPEG compressed frames in a video stream have differing sizes and complexities, and this influences the decompression time.
  - On a multi-tasking computer, other processes may execute simultaneously. This increases the CPU load, and the decompression time increases.
- *Display delay jitter*:
  The display time is not constant because of the CPU load variations and the jitter of the X protocol used when transmitting images from client memory to the X server.
- *Transmission delay jitter*:
  As explained in section 3.2, the network transmission time is not constant either. This is due to the nature of todays network protocols. Transmission delay jitter is investigated in section 3.2.

Alas, the video player never knows the total time used to process a frame beforehand. Still, we have to base the frame rate on the frames processed so far. In the following I devise an approach to handle this problem.

First, in the calculation of the number of frames to skip, we have to add a *jitter duration*. The purpose of this is to incorporate the delay jitter in advance so frames that take longer time to process does not arrive too late. Adding the jitter duration leads to a lower frame rate, but now less frames arrive too late. It is a kind of 'safety margin' in case a frame uses more time processing than the average. The jitter duration time may be determined by means of statistics (standard deviation) or determined empirically by testing.

A way of varying the frame rate while playing a movie is necessary as well. If the CPU load or network load suddenly changes, the frame rate has to be increased or decreased accordingly. This can be measured by counting the number of frames arriving too late. If this number increases above a defined threshold, the number of frames to skip has to be increased. Similarly, if the number of frames arriving too late is low, and the idle time of the system is high, the number of frames to skip is decreased. The algorithm actually used in the video player is described in section 5.2.

This dynamic variation of the frame rate has to be handled with care. It is very annoying for the user if the frame rate increases and decreases frequently; instead it is better to

maintain a constant, low frame rate. The rule of thumb is therefore to lower the frame rate relatively quickly when the number of frames arriving too late increases, and be more careful increasing the frame rate.

An approach to handle frames arriving too late has to be devised as well. Frames arrive too late when processing of the frame used more time than expected, for instance when decompression of a frame is not finished when it should be displayed. There are generally three different approaches for handling frames that arrive too late.

1. Discard the current frame and continue with the next one.
2. Wait for the delayed frame and discard the next frame.
3. Display the part of the delayed frame that is decompressed, and continue with the next one.

Strategy 1 does not disturb the synchronization of the video stream, but the work spent on decompressing the skipped frame is worthless. Strategy 2 does not waste the processing time spent on decompressing, but the synchronization will be disturbed because the frame will arrive too late. The third strategy is the best one concerning decompression processing and synchronization, however, it is not pleasant at all for the user to watch only parts of frames now and then.

4.4 Synchronization

Digital video is a synchronous medium, ie. video frames have the same *period*. In addition, when multiple streams exist in the system, the streams have to be synchronized. A typical example of multiple streams in digital video is when both video and audio streams are present. The below presentation is based on [Skeide93].

Synchronization can be categorized by *location* and *structure of the data sources*:

- **Local single source**: Data arrives from one single stream; the stream is stored on local disk. The necessary synchronization is achieved by reading data at the correct time (isochronous data stream), and rout it to the correct device.
- **Local multiple source**: Data arrives through multiple streams, which are stored on local disk. In this case all the streams have to be inter-synchronized in addition to keeping the streams isochronous.
- **Distributed single source**: Data arrives from one single source located remotely; data is transmitted through a network connection. Synchronization is similar to the local single source case.
- **Distributed multiple source**: Data arrives through multiple streams which are located remotely; data is transmitted through a network connection. Synchronization is similar to the local multiple source case.

Synchronization *methods* are also usually divided into different categories based on different views of the methods. The following classifications are introduced:

- **Synchronization classes**: Synchronization methods may be separated into *parallel synchronization* and *serial synchronization*. The latter deals with the data rate in

a single stream, while parallel synchronization take care of time scheduling between multiple data streams (ie. audio and video streams). It is possible to transform parallel synchronization into serial synchronization by *merging* the data streams.

- **Synchronization scope**: We distinguish between *point synchronization* and *continuous synchronization*. Point synchronization requires that one field in an entity corresponds a field in another entity, for instance time stamps of video frames. Continuous synchronization does not have this discrete time control, instead synchronization of activities continuously during a time period is required.
- **Synchronization master**: A synchronization master is in control of the synchronization of streams; the master controls the total synchronization. Sometimes a global system clock may be used a the master.
- **Synchronization precision**: Synchronization of streams might not require exact precision; ie. some variations are allowed. Synchronization precision indicates the error that can be allowed. For instance, *lip synchronization*[3] requires a precision of 0.1 - 0.001 second, while caption requires a precision of 0.1 - 1 second.

Another factor to consider is *where* in the process hierarchy the synchronization is performed. There are generally three different levels where synchronization is performed:

- **Controller based synchronization**: In this case synchronization is performed by a separate device controller. The advantages are very fast and flexible ways to read, synchronize and write information, without disturbing other parts of the system. However, control devices are often limited to specific systems, and this limits the use of controller based synchronization.
- **Operating system based synchronization**: The synchronization is performed in the core of the operating system. This is the most efficient type, and is well suited for local multiple source synchronization. Today only a few operating systems provide this possibility.
- **Application based synchronization**: This is the most flexible model for synchronization. The synchronization is handled by the application program only. The cost of the flexibility is loss in efficiency, due to overhead of system calls.

One of the most important issues concerning playback of digital video streams is to keep the data streams *isochronous*. This means that frames arrive at a constant rate, ie. the distance in time between adjacent frames in a stream must not vary too much. Watching video when frames arrive at a non-constant rate, is not pleasant. Maintaining an isochronous data stream is a matter of serial synchronization; each frame frame is synchronized with respect to the other frames in the stream.

4.5 Network Transmission

Chapter 3 discussed network transmission, and some protocols were presented. The chapter shows that a lot of work has been done already on network issues. This report will not contribute to the general problem of transmitting continuous media, rather than conclude that protocols providing Quality of Service parameters and guaranteed delivery are required.

[3] An audio stream synchronized with a video sequence of a person who is talking

However, *software processing* of video demands that special care is taken with respect to the network transmission. This section presents the specific requirements for network transmission to a software video player, and some solutions are suggested.

When using the approach of skipping frames when playing video, the video player is particularly vulnerable to frames arriving too late from the remote sender. A delayed or lost frame will naturally not be displayed, and this is very visible when frames are skipped. A video player displaying all the frames would simply proceed with the succeeding frame, and the loss of the frame frame is not so visible when the frame rate is for instance 20 fps. For a video player skipping 8 frames (ie. displaying every 9. frame), losing a frame would mean skipping 17 frames, and this is not very pleasant for the user to watch.

This problem could solved by *pre-transmission* of frames. The remote server must know the current position of the movie and the number of frames skipped during video playback. On account of this information, the server transmits frames to the video player before the player needs them. The frames are stored in a buffer at the video player side, and the video player accesses the frames from the buffer when they are to be displayed.

A digital video player is an interactive application; the user may change playback mode (play, wind, step, etc.) at any time. This means that the remote host must be kept updated on the current playback mode and position. Pre-sending of frames is most important when the video player is in play mode; when stepping and winding the video player can afford to loose a frame. The approach suggested is therefore:

- Always send the next *play* frame, no matter what the current playback mode is.
- When the video player enters play mode, the sender fills the buffer on the receiver side by pre-sending the frames to be played.

This solution requires advanced receiver mechanism and buffer handling on the video player side. Since the frame rate may vary during playback of a movie using a software video player, the remote server must know the current number of skip frames. This is necessary when calculating which frames to pre-send. It is the responsibility of the video player to feed this information to the remote server, in addition to receiving frames.

The pre-sending and buffering approach also opens for re-transmission of lost frames. Since the remote host is sending ahead of the video player, there is time to send a re-transmission request when a frame is lost in the network. Handling re-transmissions of frames adds to the complexity of the video; it must check which frames that arrive and determine when to send a retransmission request.

The solution described above may be called a *media-specific* protocol; the protocol is tailored to suit the current application with the requirements of the medium being transmitted.

4.6 Post processing of images

The JPEG standard only specifies compression of images. Typical applications of JPEG must add functions that is not specified in the JPEG standard. The most common functions are:

- Upsampling of components
- Colorspace conversion

- Color quantizing
- Dithering

Prior to discussing the post processing of images, a section describing the RGB and YCbCr colorspaces is necessary. This is followed by a description of the various post processing that may be necessary for JPEG compressed images.

4.6.1 Colorspaces

Image data is represented in various *color spaces*. A color space, also called a color model, defines how color images are interpreted. The trichromatic theory states that, ideally, three arrays of samples should be sufficient to represent a color image [Pennebaker93].

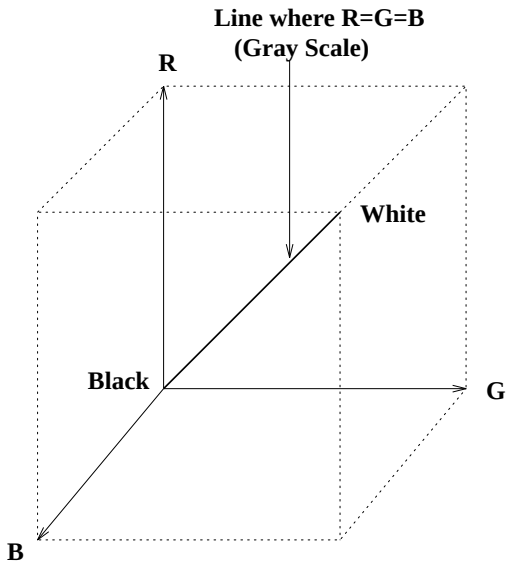

Figure 4.3: The RGB color model. Points close to (0,0,0) are dark, while those round (1,1,1) are bright.

Among various color spaces, the *RGB* and *YCbCr* colorspaces are particularly interesting with respect to JPEG. In the RGB (red, green, blue) color model colors are represented as positive combinations of the red, green and blue primaries. Each of the values can be varied independently, and we can therefore create a three-dimensional color space with the three components (R, B and G) as coordinates. Colors are represented as points in this space, as shown in figure 4.3. Shades of gray from black to white are found on the diagonal line where R=G=B. In general, pixels in a color space have information from the samples of each component, and the image is comprised of the two-dimensional arrays of the component samples. Further information is found in [Gonzalez92].

The YCbCr color model is a *luminance-chrominance* color space, and is particularly important to JPEG compression (and image compression in general). Luminance-chrominance color spaces have one luminance component (grayscale) and two chrominance components (color). The luminance provides a grayscale version of the image, and the chrominance components provide the extra information that converts the grayscale image to a color image. The mapping between RGB and YCbCr is given by the following equations:

$$\begin{aligned} Y &= \tfrac{3}{10}R + \tfrac{6}{10}G + \tfrac{1}{10}B \\ V &= R - Y \\ U &= B - Y \end{aligned} \tag{4.2}$$

Further information is found in [Gonzalez92] and [Haugen94].

4.6.2 Upsampling

The JPEG standard is designed to be 'color blind', ie. JPEG makes no assumption about the particular color model used. However, the human eye is not as sensitive to high-frequency color as it is to grayscale. Consequently, you can afford to lose a lot more information in the chrominance components than in you can in the luminance component. On account of this, JPEG compressed images are often compressed in the YCbCr colorspace, utilizing a technique called *downsampling* of image components. Downsampling means averaging together groups of pixels, and thereby achieve higher compression. Downsampling is obviously a lossy process.

The luminance component is not downsampled, it is left at full resolution. The chrominance components are usually reduced 2:1 horizontally and either 2:1 or 1:1 (no change) vertically. JPEG refers to these downsampling ratios as *2h2v* and *2h1v* sampling, respectively, and sometimes *411* and *422*. The downsampling effectively reduces the data volume by one-half (2h2v) or one-third (2h1v), while having nearly no impact on perceived image quality. The 2h2v and 2h1v downsampling is illustrated in figure 4.4.

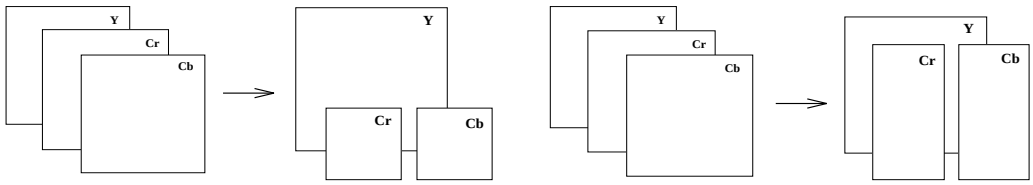

Figure 4.4: 2h2v downsampling to the left, 2h1v downsampling to the right.

Upsampling is the inverse process of downsampling. Upsampling restores the chrominance components to full size, and must be applied after the basic JPEG decompression. After the upsampling process, the image is not similar to the original image because of the averaging when downsampling the image. It is also possible to discard the color components and get the plain grayscale (only luminance component) image. Upsampling is not applied in this case.

4.6.3 Colorspace conversion

As stated in the previous section, JPEG images are compressed in the YCbCr colorspace. However, most display hardware require images to be represented in the RGB color model. Consequently, prior to displaying an image on the screen, *colorspace conversion* from YCbCr to RGB has to be performed. The mapping from YCbCr to RGB is given by equation 4.2 page 49. For grayscale images the colorspace conversion is a no-op; the luminance component is simply extracted from the YCbCr image.

4.6.4 Color quantization

Color images which are JPEG compressed use 24 bits per pixel, 8 bits for each of the YCbCr components. This yields a total of 16.7 millions (2^{24}) different colors. Unfortunately, most users do not have 24 bit per pixel (full color) display hardware. Typical display hardware stores 8 or fewer bits per pixel, so it can display 256 or fewer distinct colors at a time. To display a 24 bits image on this kind of display, an appropriate set of representative colors has to be chosen, and then map the full color image into these colors. This process is called *color quantization*, and must not be confused with the downsampling process described in section 4.6.2 or the frequency component quantizing described in section 2.4.3. Clearly, color quantization is a lossy process.

Color quantization requires that a *colormap* is specified. A colormap is a lookup table used when reducing the full color image, and the output image is written in *colormapped* form. This means that each pixel is an index in the colormap, which specifies the RGB values for each pixel value that is selected in the colormap. Colormaps are also discussed in section 4.2.1.

It is important that the colormap is representative for the image, meaning that the colors that are selected are as similar as possible to the original image. Naturally, a color quantized image will not be of the same high quality as the original. Ideally each single image should have its own custom colormap. Different images have different colors, and there is no such thing as a *global* colormap that is perfect for all images. This fact is due to the restricted number of colors available. However, when dealing with digital video, it is not convenient to have separate colormap for every image. This is explained further in section 4.2.1. When color quantization is performed on images in a digital video stream, a single colormap is utilized. This colormap must have a selection of colors to cover the whole color spectrum.

4.6.5 Dithering

When performing color quantization, a predefined colormap is used. This usually produces images of poor visual quality; to achieve the best possible quality a custom colormap should be produced for each image as explained in the previous section. An image processing technique called *dithering* may be used to improve the image quality. Dithering 'fools' the human eye to believe that there are more colors in the image than there actually is. This section gives a brief introduction to *ordered dither* and *Floyd-Steinberg dither*.

In order to understand *ordered dithering*, it is helpful to look at a bilevel image display (only two colors, usually black and white). Instead of letting each image pixel represent an output pixel directly, each image pixel is intensified according to a *dither matrix*. The dither matrix is of size $N\ x\ N$, and produces N^2+1 possible intensities on a bilevel display. Each matrix cell has a defined value; the guidelines for selecting matrix values is referred in [Foley91]. A typical 3x3 dither matrix, producing ten intensity levels, is listed below.

$$\begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}$$

Whether or not to intensify a pixel at point (x,y) in the image depends on the desired intensity S(x,y) at that point and the dither matrix. First two indexes into the dither matrix, i and j, are computed:

$$\begin{aligned} i = &\quad x \quad \text{modulo} \quad N \\ j = &\quad y \quad \text{modulo} \quad N \end{aligned}$$

Then if

$$S(x, y) > D_{ij}^{(N)}$$

the point at S(x,y) is intensified, otherwise, it is not.

A display with more than two intensity levels (multi-level displays) can also use ordered dithering. This requires additional dither matrixes, defined by the number of display intensities and the matrix size chosen. A size of NxN of the dither matrixes and I intensity levels produces $(N^2 * (I - 1)) + 1$ different intensities. Consider a display with two bits per pixel and hence four intensity levels. A matrix size of two allows us to display (4x3)+1 = 13 intensities. For color dithering, each component is dithered separately. See [Schumacher91].

Floyd-Steinberg dithering is a technique that usually produces satisfactory visual results at the cost of slower processing compared to ordered dithering. *Error diffusion* is used to propagate the error (ie. the difference between the exact pixel value and approximated value actually displayed) to the four image-array pixels to the right and below the pixel in question. This has the effect of spreading, or diffusing, the error over several pixels in the image array, see [Floyd75].

Dithering is also discussed in section 5.1.2

To sum up this discussion, dithering techniques is only used when quantizing colors. There are a lot of dithering techniques proposed. In this project I only look at Floyd-Steinberg and the faster ordered dithering. For instance, the `mpeg_play` [Rowe92] video player has a lot more dither algorithms available. Dithering is only necessary when images do not have a custom colormap and/or the number of available colors is few. To read more about dithering, see [Schumacher91].

4.6.6 Blocking Artifacts

The JPEG technique of segmenting images into 8x8 blocks may lead to *blocking artifacts*; visible discontinuities between adjacent blocks (see [Pennebaker93]). An effect called *ringing* may occur as well. The JPEG standard does not define a way to suppress these effects, rather it *suggests* a technique called *AC prediction* in an appendix. AC prediction is, as the words implies, a way of predicting the AC coefficient of a DCT block when decompressing an image. This technique will not be further investigated in this paper; it is outlined in the JPEG specification, appendix A [JPEG94].

Blocking artifacts occur most often when compressing with low q-factors (high compression ratio). Digital video requires a certain image quality, and it is therefore not beneficial to use so low q-factors such that blocking artifacts occur. Alternatively, *smoothing* techniques could be applied to remove these effects. However, as I see it, the best solution is to stick with a high enough q-factor and avoid this problem. The dithering technique discussed above also contributes in the suppressing of blocking artifacts.

4.6.7 Postprocessing summary

From the above discussion, one understands that post processing is necessary after decompressing JPEG compressed images. However, the user must decide which post processing techniques to use, based on the image quality desired. This may also be a tradeoff between speed and quality; the higher quality desired, the longer time is spent on processing the image. Figure 4.5 shows a summary of the above post processing techniques, in the pipelined sequence they usually are applied. In this project I will not discuss the blocking artifact effects further.

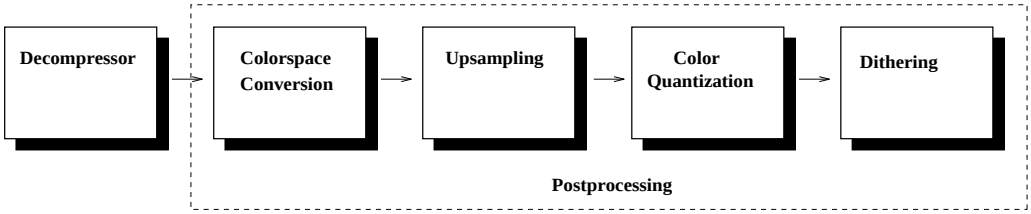

Figure 4.5: Postprocessing of decompressed JPEG images.

Chapter 5

Functionality of the Video Player

A software Motion JPEG video player, called *jpegvid*, has been developed in this project. This chapter describes the functionality of the video player, and also serves as a user's guide. The user interface is presented, and design choices are discussed and reasoned.

In the first section, the user interface of jpegvid is described, including the user controls and visual aspects. Second the input parameters to jpegvid are listed and described. Determining the frame rate and discarding of frames is discussed in the following section. The fourth section discusses the color handling of jpegvid, and the chapter is concluded by summarizing the limitations of jpegvid.

5.1 User Interface

The user interface of jpegvid is pictured in figure 5.1 and figure 5.2. As can be seen, the player controls are in a separate window from the display window. The main reason for this is an implementation decision, see chapter 7. In spite of this solution appearing a little odd and unusual at first, some advantages follow as well. The display window can be resized as much as you like, and still not affect the control panel. This is an beneficial property when scaling movies, discussed below and when the original video frame size is small. In addition, images have various *depths*[1]. A grayscale image usually has a depth of 8, while a color image sometimes uses 24 bits to describe RGB values (8 bits per red, green and blue, respectively). Image depth is further described in section 4.2.1. The depth of the images in the video stream played with jpegvid only has an impact on the display window; the control window always has the same colors and bitmaps.

5.1.1 Player controls

The player controls include the usual controls known from analoge video players:

- **Quit:** The quit button terminates jpegvid.
- **Play/Pause:** This button controls the playing of a video film. When jpegvid is in play mode, the button shows the pause symbol. Similarly, when the player is in pause mode, the button shows the play symbol.

[1] The depth of an image is the number of bits per image pixel.

Figure 5.1: The video display window of jpegvid

Figure 5.2: The video controls of jpegvid

- **Step Forward:** Pushing this button makes the video jump one frame forward, no audio is played. Double clicking on this button makes the player enter a 'slow forward' mode.
- **Step Reverse:** Similar to the one above, except that the direction is reverse.
- **Fast Forward:** Play the video faster than the play speed. On account of the inherent slow decoding of frames with jpegvid, this mode will only skip a larger number of frames, ie. the frequency of displaying frmaes is not increased. Pushing this button additional times speeds up the playing in a linear scale.
- **Fast Reverse:** Similar to the one above, except that the movie winds reverse.

In addition to this, the player has a slider device, which is not usually found on analoge players. The slider has two functions:

- When the video is played, the slider moves its position according to the current position of the film. The slider shows how much of the film which is played and how much is left on a percentage scale.

- It is possible to grab the slider with the mouse pointer, and move it to a desired position. The video will jump to a frame in the video sequence according to the percentage scale of the slider. Frames are not displayed while moving the slider, the display window is updated after the slider is set.

At the boundaries of the video sequence, jpegvid enters pause mode. This means if the user attempts to access a frame less than the first frame number, the video enters pause mode displaying the first frame. Similarly, when a frame larger than the last frame number is accessed, jpegvid pauses and displays the last frame. This functionality is attractive particularly when winding forward or backward, since the boundaries may be reached faster than expected, due to the skip frame approach.

It is possible to control jpegvid by means of keyboard characters. This a convenience for frequent users of the system. In addition this facility is an advantage because the display window has its own custom colormap.

The character controls are:

- '**q**': Quit.
- '**p**': Play/Pause.
- '**f**': Fast Forward.
- '**r**': Fast Rewind.
- '**s**': Step Forward.
- '**b**': Step Reverse.

5.1.2 Input Parameters

The jpegvid application may be started with some input parameters. The syntax of the input parameters is as follows:

```
Usage: jpegvid [-(color|grayscale)] [-depth d] [-scale 1/n]
               [-idct (int|fast|float)] [[-p <port>] <host>:] <filename>
       jpegvid -v
```

The only mandatory parameter is the `<filename>`. When this is the only parameter specified, jpegvid will select appropriate defaults. The current defaults are:

```
jpegvid -color -scale 1/1 -depth 32 -idct fast
```

Naturally, jpegvid can neither display color video images on grayscale display hardware nor use a depth larger than the display hardware allows. In this case, jpegvid selects parameters appropriate for the display hardware as close to the requested settings as possible. This selection algorithm is described in section 5.3.

Specifying `jpegvid -v` simply outputs the version of the current implementation of jpegvid; no video playing is executed

The `-(color | grayscale)` parameter specifies whether images are displayed in color or grayscale. Color output requires that the video film is stored in color format. On the

other hand, it is possible to force grayscale output from a color image. This is explained in section 4.6.1. Setting the `-grayscale` parameter has this effect. The advantage of forcing grayscale output is that decompression of frames runs faster, resulting in a possible higher frame rate. The price to pay is the decreased quality of grayscale video film compared to color video.

The `-depth d` parameter is used to specify the depth of the output images in the video sequence. In cooperation with the `-(color | grayscale)` parameter, the depth is used to select the *visual* of the display window. Visuals are described in section 4.2.1, and the actual selection of the visual in jpegvid is presented in section 5.3. This option is also used to specify when color quantizing (see section 4.6.4) must be applied;

```
jpegvid -color -depth 32 ...
```

demands no color quantizing, while

```
jpegvid -color -depth 8 ...
```

requires that colors are quantized into 256 (2^8) different colors. Color quantizing makes decompression run slower. On the other hand, 8 bits images are displayed faster on the screen than for instance 24 bit images. Currently, only quantizing into 256 colors is possible in jpegvid.

When quantizing colors, dithering may be applied, see section 4.6.5. Different types of dithering may be added as an option in subsequent versions, but for now, jpegvid applies *ordered dithering*. Floyd-Steinberg dithering has been tested, but proved too slow, and the image quality was not very much better than ordered dithering. No dithering at all was also tested. This is faster than ordered dithering, however it produces far too low image quality. Figure 5.3 shows the three types of dithering mentioned above. Note the poor image quality when no dithering is applied, and the small difference between ordered dither and Floyd-Steinberg dither[2].

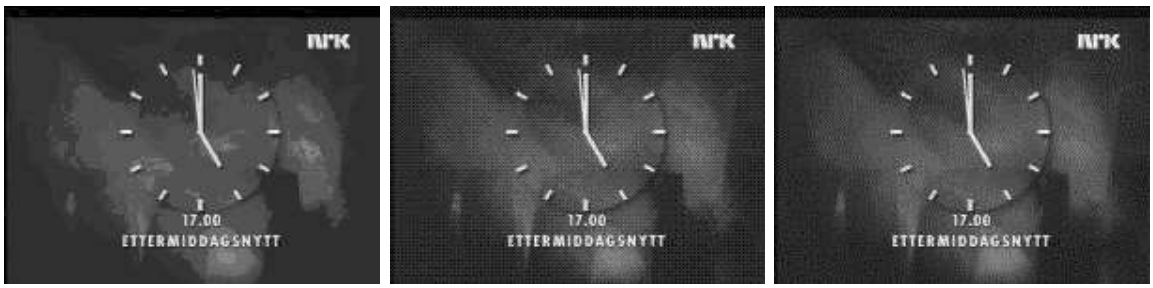

Figure 5.3: No dither to the left, ordered dither in the middle and Floyd-Steinberg dither to the right.

Another feature of jpegvid is the possibility of scaling down the original size of the video frames. Setting the `-scale 1/n` parameter scales the original frames by *1/n* both horizontally and vertically. The current options for `1/n` in `-scale 1/n` is `1/1`, `1/2`, `1/4` and `1/8`. If the user inputs a different value than those listed above, jpegvid selects the closest one. Down-scaling of frames also has the effect of speeding up decompression, at the cost of the lower image quality produced by the smaller frame size.

As described in section 2.4.4, the IDCT[3] algorithm is not standardized by JPEG. This opens for three choices of IDCT algorithm in jpegvid. The alternatives are:

[2]Unfortunately, due to Grayscale and Postscript converting, and paper print quality, the differences are not sufficiently illustrated in the figure. When running jpegvid, the differences are much more visible

[3]Inverse Discrete Cosines Transform

- `-idct int`: An integer based, slow but accurate algorithm, based on the equations listed in section 2.4.4.

- `-idct fast`: A fast, inaccurate algorithm. The algorithm is based on [Arai88], and is the default of the `-idct` option.

- `-idct float`: An algorithm based on floating point operations, opposed to the `-idct int` option. This algorithm is accurate, and may be fast on computers with a fast floating point processor. On computers without this kind of processor, the integer point option is recommended instead, because integer arithmetic is usually faster than floating point.

As the above parameters, the `-idct` option is used to gain speed of decompression. However, the image quality is not drastically decreased when using the `-idct fast` option; therefore this is recommended for all but the most quality demanding users.

The above input parameters may be combined to achieve various video output quality and frame rates. The setting that produces the highest frame rate is

```
jpegvid -grayscale -depth 8 -scale 1/8 -idct fast <filename>,
```

resulting in a small grayscale display window. A user searching for the highest output quality on a 24-plane display screen would specify

```
jpegvid -color -depth 24 -scale 1/1 -idct int <filename>
```

This will, however, result in a significantly lower frame rate compared to the selection above.

5.1.2.1 Network Connection

In the parameter syntax above, the `-p <port> -<host>`: option specifies a remote host and a port number. These parameters are used when a server on a remote host transmits audio and video data to jpegvid (instead of reading data from a local disk). *This is not implemented in this project.* I have focussed on the software decompression of JPEG compressed images and the playing of digital video. Because of time constraints, the network part is not implemented. Therefore, when the user specifies the above parameters, the message

```
Sorry, network connection is not implemented.
```

is printed out.

5.2 Frame Rate

The jpegvid application is executed as a software process, and therefore the time spent on decompressing and displaying frames varies. Section 4.3 discusses this effect. The jpegvid video player has to determine the frame rate according to the current computer on which it is running and the video which is played. The frame rate may have to be changed during the playback of a video as well. An approach to handle frames arriving to late is specified as well.

It is important not to confuse the following expressions:

- **Original frame rate**: This is the frame rate of which the video was captured when recording.

- *Produced frame rate*: The playback frame rate, ie. the speed of playing a video sequence. jpegvid skip frames in order to maintain the data isochronous, and usually the produced frame rate is less than the original frame rate

When jpegvid starts, the decompression time and display time for the first few frames is measured. The maximum average processing time is used to compute the initial number of frames to skip, according to the following equation:

$$\texttt{SkipFrames} = \frac{max(averageDecompTime, averagePutvideoTime)}{Interval} \tag{5.1}$$

where $Interval$ is the original duration between two djacent frames.

While jpegvid is running, the number of frames to skip is calculated by means of a *sliding window* technique. The size of the sliding window is 10 seconds. The window keeps count of the number of frames displyed and the number of discarded frames continuously during the latest 10 seconds. If the number of discarded frames increases above 15 percent of the total number of frames displayed during the window interval, the number of skip frames is increased by two. This means that the frame rate is decreased. On the other hand, if the number of discarded frames decreases below 2 percent of the total number of displayed frames during the window interval, the number of skip frames is decreased by one. This increases the frame rate.

To avoid rapid increase and decrease of the produced frame rate, a lock mechanism is applied. When the frame rate is increased, it can not be increased again for some time, in order to see if the frame rate is stable on this level. Similarly, when the frame rate is decreased, the frame rate is locked for decrease on this level to check whether the frame rate is stable.

The sliding window mechanism provides a way of dynamiccally varying the frame rate. This is not really a desirable property of a video player; the frame rate should be kept constant. However, because the processing of video frames varies, this technique is applied. The alternative would be to select a constant number of skip frames, and maintain this during the complete playback time of the video. This approach has two main disadvantages:

1. It is difficult to calculate a skip frame number which can be applied to the complete video sequence.
2. If the CPU-load increases, processing of frames slows down, and a lot of frames would be lost. Similarly, if the CPU-load decreases, the approach does not take advantage of the free processing capacity.

In addition to the an algorithm regulating the number of frames to skip, an approach to discard frames has to be devised. A frames is discarded either if it arrives too late from decompression, or displaying of the previous frame is not finished. In the latter case, the new frame is simply discarded, and the succeeding frame is displayed next time. For frames arriving too late from decompression, the frame is *not* discarded. Rather, the frame is displayed next time, and the frame that should have been displayed is discarded instead. This is referred to as strategy 2 in section 4.3. This strategy causes disturbance in synchronization, but on the other hand, the decompression processing is not wasted. For the jpegvid video player, decompression processing is a precious resource, and the synchronization disturbance can be tolerated.

For a video player using dedicated hardware for decompression, like the vshow video player (see section 3.9), the decompression of frames is not that precious. Consequently, simply

discarding the frame that arrives from decompression may be applied in this case. This approach is referred to as *strategy 1* in section 4.3. This approach is actually applied in vshow.

5.3 Visuals, colormaps and portability

Section 4.2.1 explains the concept of visuals. In order to support portability, jpegvid supports several types of visuals. The selection of visual depends on the current display hardware, the visual classes supported by X Windows, and any input parameters provided by the user. Since jpegvid always uses a application-specific colormap, we do not have to separate between read-write and read-only visual types (see section 4.2.1). This means that it is indifferent whether we are using for instance a `DirectColor` or a `TrueColor` visual; jpegvid selects either if it exists.

The selection of visual follows a predefined order, from the highest quality output to lowest. If a visual class and depth can not be matched, the subsequent visual class and depth in the hierarchy is tested. The selection hierarchy is shown in figure 5.4.

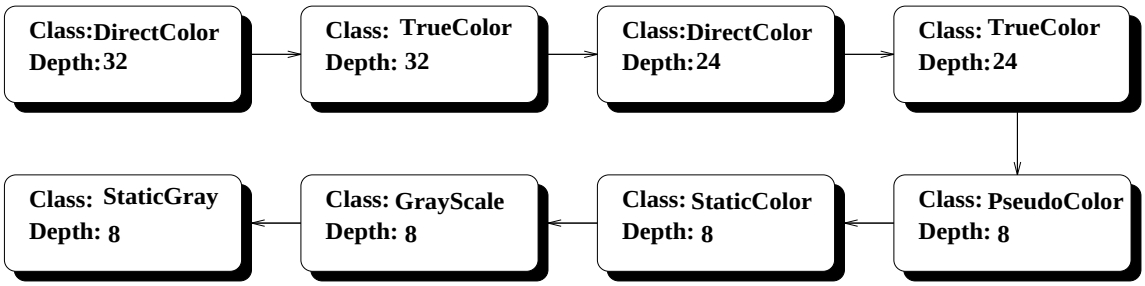

Figure 5.4: The visual selection hierarchy for jpegvid.

Both visuals of depth 24 and depth 32 use three bytes (24 bits) to specify RGB values. A visual of depth 24 adds an extra byte in order to align to 32 bits boundaries, and a visual of depth 32 has one unused byte. Consequently, the 24 and 32 depths can be used indifferently; however, all may not be supported by the the current X server. Therefore jpegvid has to traverse all possibilities, starting from the highest quality output.

When the user specifies the `-(color | grayscale)` and/or the `-depth d` parameters, jpegvid starts from the corresponding entry in the selection hierarchy. Note that when `-grayscale` option is selected, jpegvid actually starts from the `PseudoColor` entry, because `PseudoColor` and `StaticColor` visuals can be used to simulate grayscale output by only including grayscale colors in the colormap.

A *Colormap* has to be utilized when the images are color quantized, ie. when specifying the `-color -depth 8` parameters. The jpegvid application defines a colormap with an acceptable selection of colors which is used in this case. As described in section 4.2.1, color quantizing produces images of poorer quality compared to full color output. To improve the quality of the color quantized images, *ordered dithering* is applied (see section 5.1.2).

Section 4.2.1 explains the concept of *virtual colormaps*. The jpegvid video player defines a virtual colormap when quantizing colors or when grayscale output is requested. For computers which have only one hardware colormap available, the virtual colormap is swapped in and out of the hardware colormap dependent of whether jpegvid is the active application on the screen (ie. the mouse pointer is inside the display window). The swapping

causes a *flickering* the screen. In addition, when jpegvid is active, other applications on the screen get their colors disturbed. Similarly, if jpegvid is not the active application, the images in the video display window are disturbed. This is a problem when using jpegvid in connection with other tools, because only one application have the correct colors at any time.

Figure 5.5: A black and white Floyd-Steinberg dithered image.

As you see from figure 5.4, jpegvid requires at least an 8-plane display (having visuals of depth 8). The next step on the hierarchy would be 1-plane display screens, providing 2 colors, usually black and white. One could argue that it makes no sense playing video on a white-black screen; however, *dithering* could be used to achieve very low quality output images. This is not implemented in jpegvid, but to illustrate the possibility, a black and white dithered image is shown in figure 5.5.

5.4 Video Data Files

The jpegvid video player reads audio and video data from two separate files. In addition there are *offset files* which index the two data files. The offset files are required for jpegvid, and are heavily used when skipping frames and jumping to positions by means of the slider. The alternative to using offset files, is to run through the video films sequentially, and this may be more time consuming and keeps the disk busy more frequently.

The purpose of using four files instead of gathering everything into one single file is that the video data is viewed from a *database perspective*. In this perspective, audio and video data are different entities. Consequently, putting them into separate files makes sense. From a practical point of view this is also profitable, particularly when playing only either video or audio streams. Keeping the indexes in the offset files is also consistent with the database perspective.

To keep track of which files that belong together, naming conventions have been determined. A video film originally named *ExampleVideo* has the four files shown in table 5.4 [Skeide93].

5.5 Summary of Limitations

This chapter is concluded by summarizing the limitations of jpegvid:

ExampleVideo.jvid	File with JPEG compressed video frames
ExampleVideo_lyd.jvid	File containing audio data
ExampleVideo.dat	Offset file for ExampleVideo.jvid
ExampleVideo_lyd.dat	Offset file for ExampleVideo_lyd.jvid

Table 5.1: File name conventions

- When jpegvid runs on a remote host, using NFS[4], audio is not supported. In addition, MIT-SHM (see section 4.2.2 is not used in this mode.
- It is not possible to scale frames up to be larger than original frame size.
- The jpegvid application requires that UNIX shared memory is supported. For instance, if a full color image of size 400x300 is to be displayed, a shared memory segment of size 480 Kbytes is used.
- Ideally jpegvid should be the only CPU-intensive application running on the host in order to achieve highest possible frame rate and minimize delay jitter.
- jpegvid requires at least 8-plane display hardware.
- The network connection client-server mode is not implemented.

[4]Network File System

Chapter 6

System Description

Previous chapters have covered the basic information concerning a software motion JPEG video player. This chapter provides an system description of the jpegvid application. An overview of the system architecture, both for a local connection and a network connection is provided, and the responsibilities of the modules are established. Note that the network part of the system is not implemented. The next chapter will examine the system design more closely.

6.1 System overview

The jpegvid player consists of three separate modules or processes. These are:

- `jpeg_control`
- `decompress`
- `jpeg_display`

When a network connection is used to transmit video frames, two additional processes are necessary:

- Server
- Receiver

The basic jpegvid player architecture is the same for both local and network connections. The only difference is the reading of video files. When jpegvid has a local connection, the files are read directly from the disk. A network connection requires that video data is sent by the server process to the receiver, which puts the data in the appropriate buffer.

The architecture of the player using a local connection is described in the next section. Subsequently the proposed architecture of the system utilizing a network connection is described, and the differences compared to a local connection are pointed out.

6.2 Using a Local Connection

When jpegvid reads video data from a local connection, this means that audio and video frames are read directly from the disk where the video data is stored. This disk is either a

local disk connected directly to the computer, or data is read by the *Network File System* (NFS).

The jpegvid architecture is depicted in figure 6.1. The `jpeg_control` process controls the other processes and is responsible for synchronization, flow control and user input; this is the "master" process. `decompress` does as the name indicates; it decompresses JPEG compressed frames. The task of `jpeg_display` is to display decompressed frames on the display screen.

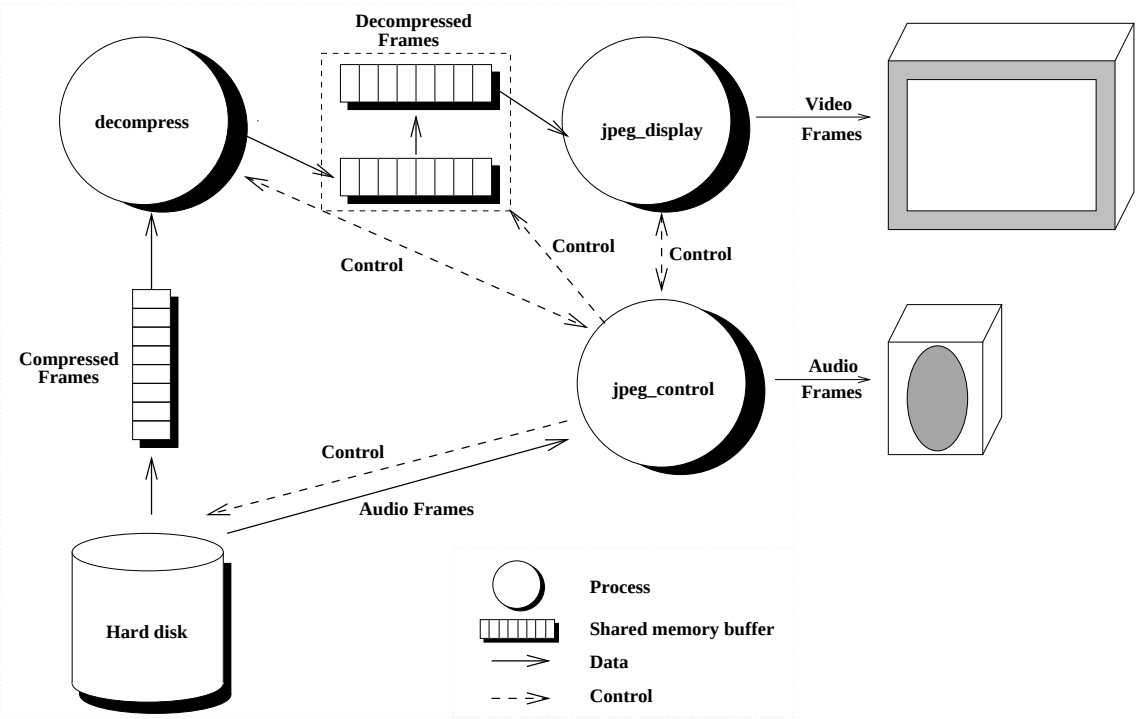

Figure 6.1: Architecture of jpegvid using a local connection.

The logical functions of `jpeg_control` are:

- Parse input parameters.
- Open video, audio and offset files for reading.
- Set up the control panel
- Execute `decompress` and `jpeg_display` processes.
- Create and initialize the communication channels between the processes, and create the two shared memory buffers.
- Feed compressed video frames to the shared memory buffer which `decompress` is attached to.
- Feed audio data to the audio device.
- Respond to user interactions in the control panel.
- Synchronization of audio and video data; this includes controlling the `decompress` and `jpeg_display` processes.

- Increase and decrease the frame rate as appropriate.

`decompress` is devoted only to decompression of JPEG frames. The logical functions are as follows:

- Initialize the decompression, based on frame size, q-factor, scaling etc.
- Connect to the communication channel and attach to the shared memory buffer.
- Read frames from the shared memory buffer.
- Perform colorspace conversion, also color to grayscale extracting.
- Perform possible scaling.
- Perform possible quantizing.
- Put data in the output shared memory buffer in the correct RGB format.

`jpeg_display` performs the drawing of images on the screen. The logical functions are:

- Create an appropriate display window.
- Determine whether to use MIT-SHM or conventional XLib calls.
- Connect to the communication channel and attach to the shared memory buffer.
- Read decompressed frames from the shared memory buffer and display them on the display screen. Displaying is trigged by `jpeg_control`

6.3 Proposed Network Connection

When jpegvid receives video data by means of a network connection, two more processes are present. A `Server` process is responsible for reading audio and video frames from disk, and sending them through the network connection. On the receiving side of the network connection, a `Receiver` process is responsible for receiving the data transmitted by the `Server`. The `Receiver` is connected to `jpeg_control`, and puts video data in the shared memory buffer which `decompress` is attached to. Figure 6.2 shows the system architecture.

In this set-up, only the `jpeg_control` process is modified slightly compared to the description in the previous section. `decompress` and `jpeg_display` are not changed at all.

The changes in jpeg_control are as follows:

- Execute the `Receiver` process and create a communication channel to it.
- The audio and video data are not read from disk; instead `jpeg_control` supervises the `Receiver` process.
- Receive the offset files and store them in a local, temporary file.
- Send messages to `Receiver` about frame rate, display mode etc.

The logical functions of `Receiver` are:

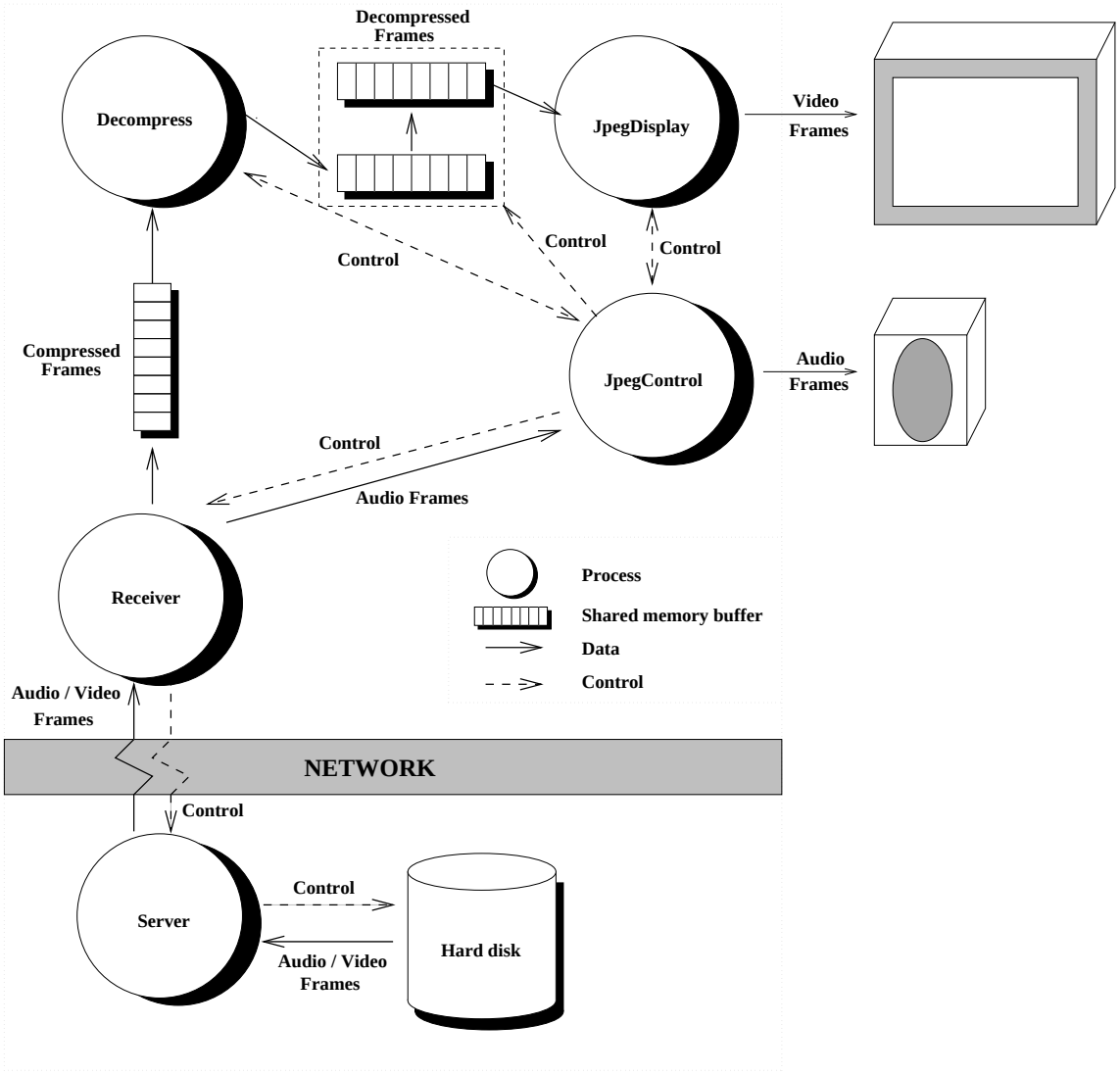

Figure 6.2: Architecture of jpegvid using a network connection

- Connect to the communication channel and attach to the shared memory buffer.
- Create a network connection, and start a `Server` process on the sender side.
- Send messages to the `Server` process about frame rate, display mode etc.
- Receive audio and video data on the network connection, and put the in the shared memory buffer.
- Perform any error checking and assembling sub-packets.
- Ask for retransmission of packets if necessary.
- Perform possible buffering of data.

The logical functions of the server are:

- Initialize and connect to the network connection.
- Receive control data from `Receiver`.

- Determine which video and audio frames to transmit based on the control data sent by `Receiver`.
- Read audio and video data from disk and transmit then on the network connection.
- Receive retransmission requests and retransmit data through the network connection.

6.4 Summary

The architecture of jpegvid is quite complex, involving three or five processes. The communication between the processes is therefore essential to performance and synchronization. On the other hand, we see that the processes have distinct functions, improving simplicity and understandability.

By separating the network task in a `Receiver` process, the changes in the total system are minimal. The main change is that data is not read from disk by `jpeg_control`; this is now `Receiver`'s responsibility. The network architecture also promotes a *client-server* approach.

This chapter provided an overview of the architecture of the system. A more thorough description and discussion follows in the next chapter. Note again that the network architecture is not implemented in this project.

Chapter 7

Construction and Implementation

The previous chapter outlined the total architecture of the jpegvid system. This chapter provides a more thorough description of the construction and implementation, in order to show how the system works.

The structure of this chapter is:

- Overall system description
- Subsystem description
  - `jpeg_control`
  - `decompress`
  - `jpeg_display`
- Description of the communication between the subsystems

7.1 Overall System Description

The overall architecture of the system is depicted in figure 6.1 page 64. The `jpeg_control` subsystem is responsible for reading audio and video frames from the video file. `jpeg_control` also allocates the shared memory buffers used to exchange video frames between the subsystems, and it creates the communication channels. Synchronization of video and audio, producing an isochronous data stream (see section 4.4) and determination of the number of frames to skip (as described in section 4.3) is also the responsibility of this module. In addition, `jpeg_control` handles user input from the control panel or the keyboard, and performs the appropriate actions.The `decompress` and `jpeg_display` subsystems can be viewed as *slaves* of `jpeg_control`. They are both executed as child processes of `jpeg_control`.

The `decompress` process does exactly what the name inidcates: It reads uncompressed JPEG frames from one shared memory buffer and writes decompressed images into another shared memory buffer. All the information `decompress` needs for decompression, ie. where to read and write data and various decompression parameters (size, scale values, color/grayscale, idct method, etc.), are received from `jpeg_control` at start-up.

Similarly, the `jpeg_display` module also has only one single responsibility: To display the decompressed images in a window on the screen. `jpeg_control` copies the uncompressed data into a shared memory buffer which `jpeg_display` is attached to. Each time a frame

is to be displayed, jpeg_control trigs jpeg_display. jpeg_display reads the data from the shared memory buffer and displays an image in the display window. Similar to the initialization of decompress, jpeg_display receives the necessary display parameters at start-up.

The overall operation of jpegvid is summarized in the diagram of figure 7.1. In subsequent sections, the subsystems are described separately.

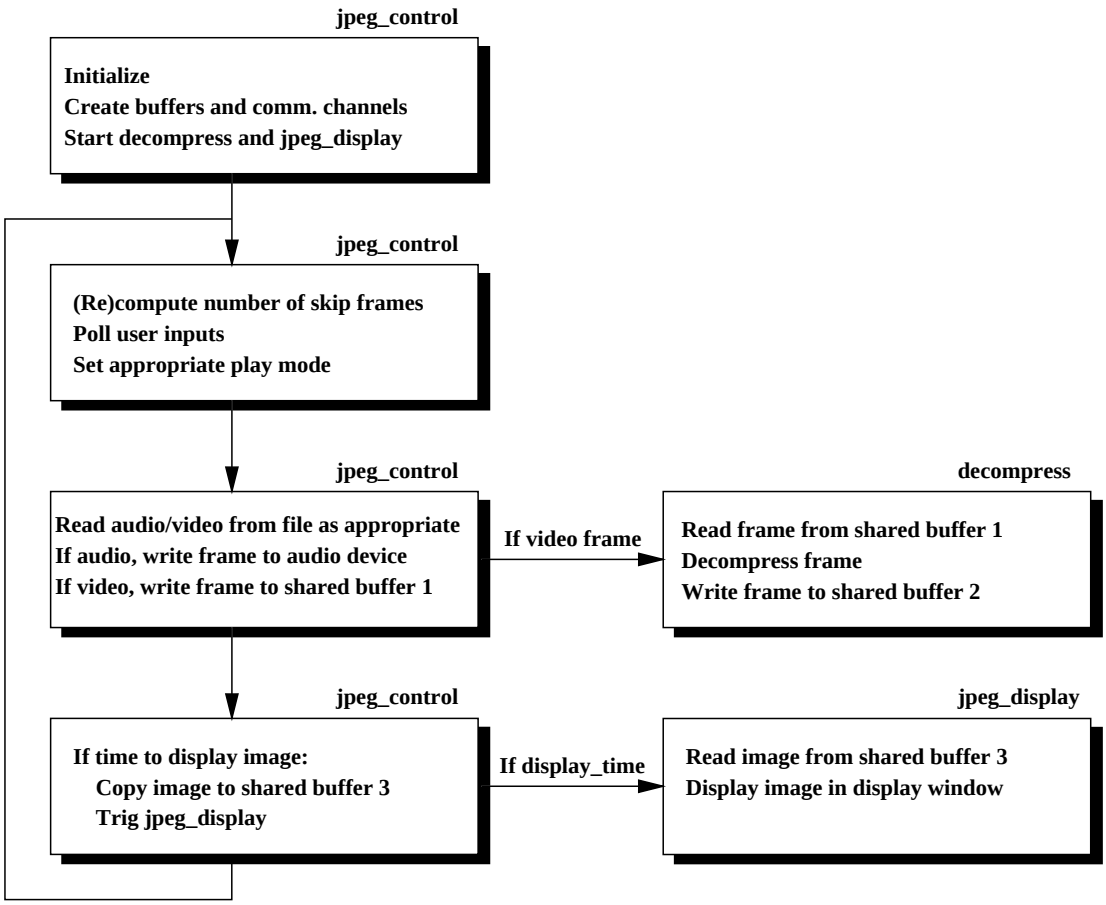

Figure 7.1: The overall operation of jpegvid.

7.2 The jpeg_control Subsystem

The jpeg_control module is the one which controls the other subsystems. It is therefore the most complex module, due to the large number of tasks that it must perform.

jpeg_control is implemented in an object oriented manner. It is based on the *libmain* framework [VoDoo94] for event driven applications. libmain defines classes like TimedObject and StreamObject which can be derived in order to receive callbacks based on time events or on streams that are ready for reading or writing. The mainloop object manages all these callbacks. The libmain package also contains the xwin object (class XMain) which incorporates X11 event handling into mainloop.

7.2.1 Internal Structure

The internal structure of `jpeg_control` is shown in figure 7.2. Objects from libmain are not shown in order to add clarity and readability. Class names are used as labels in the figure.

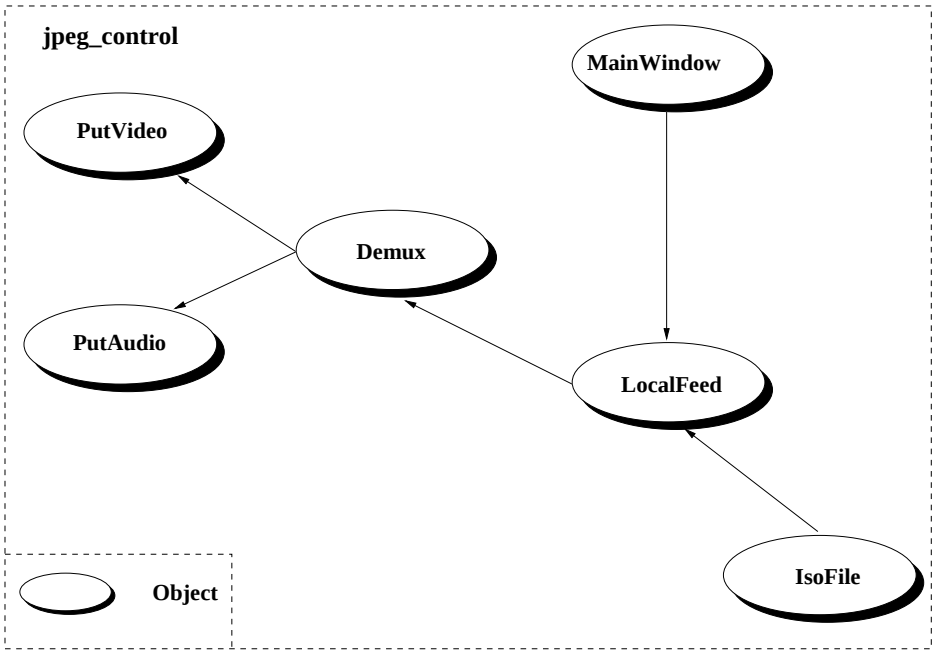

Figure 7.2: The internal structure of `jpeg_control`.

The `MainWindow` object handles all aspects of the Motif based user interface. All Motif widgets are hidden within this object. The `MainWindow` controls an `IsoFeeder` object, in this case a `LocalFeed` object. Button presses and other other control actions result in calls to the appropriate methods defined in `IsoFeeder` (`LocalFeed`).

The `LocalFeed` controls access to an `IsoFile` object. The `IsoFile` object hides all details regarding the actual file format. All reading from files is performed in the `IsoFile` object, and written into shared memory in case of a video frame. Audio frames are written in a local buffer, and given to the `Demux` object (see below).

A *receiver* object registers with the `IsoFeeder`. The `IsoFeeder` will then call the `Deliver()` method of the receiver when every a video frame is to be displayed, or audio data should be played. In the case of figure 7.2, a `Demux` object has been registered as the receiver of the data from `LoacalFeed`. `Demux` relays data to the `PutVideo` and `PutAudio` objects as appropriate. In the case of a video frame, no actual data is written to `PutVideo`. Only the information about the video data is sent; this is explained further below. Audio data, on the other hand, are given directly to the `PutAudio` object.

`PutAudio` receives audio data from `Demux`, and simply writes the data to the audio device of the computer.

The `PutVideo` object is in control of displaying images. It reads data from the shared memory buffer written by the `decompress` process, and copies it to another shared memory buffer that the `jpeg_display` process is attached to. Subsequently `PutVideo` trigs `jpeg_display`, telling it to draw the image placed in the shared memory buffer, on the display window. The `PutVideo` object is also responsible for deciding when frames are

discarded.

The class hierarchy of jpeg_control is depicted in figure 7.3. Objects derived from the libmain framework are gray.

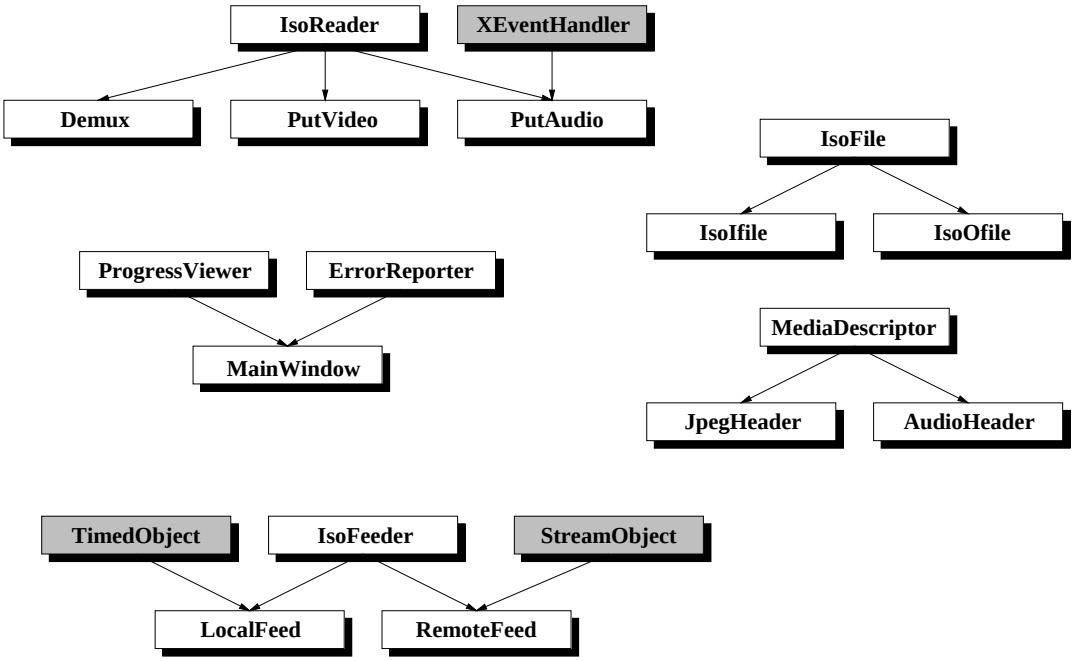

Figure 7.3: The class hierarchy of jpeg_control.

7.2.2 The Mainloop

The Mainloop object of jpeg_control is the synchronization tool. Mainloop maintains a linked list of TimedObjects, in this case instances of LocalFeed (the LocalFeed class is a subclass of TimedObject, see figure 7.3). Every time a video or audio frame is read from file, the TimedObject is linked into the list which Mainloop maintains. The TimedObjects have an associated *TimeValue* which determines when the frame should be delivered to the Demux object. This TimeValue is called ActionTime, and is assigned a value based on a *BaseTime* and a TimeValue stored in the frame header of the video frame. The file format and frame header is described in section 7.6.

The BaseTime is the reference time to the very start of the video. When jpegvid enters pause mode, step mode or wind mode, the BaseTime is updated accordingly in order to keep it correct with respect to the start time of the video. When a frame is read, the associated TimedObject is linked into the MainLoop list after assigning a ActionTime by the following equation:

$$ActionTime = BaseTime + Frameheader.TimeValue \tag{7.1}$$

A TimedObject is linked out of the Mainloop queue when current time equals ActionTime. When this happens, the TimeCallback() method of the TimedObject is called. Within TimeCallback(), two operations are performed:

1. The data carried by the TimedObject is delivered to Demux (remember that the TimedObject is actually a LocalFeed object).

2. A new frame is read from the file by the `IsoFile` object and linked into the `Mainloop` list.

Whether to read from the audio or video file in the `TimeCallback()` method depends on the number of video frames which are to be skipped, counted in the variable `SkipFrames`. If no video frames are to be skipped, audio and video frames are read alternately; ie. the type of the frame to read equals the frame which is linked out of the list. When the number of video frames to skip is greater than zero, video frames are skipped and audio frames are read and linked into the list instead. A variable called `SkippedFrames` counts down the number of skipped video frames. `SkippedFrames` is initialized to `SkipFrames` and decreased by one every time a video frame is skipped. When `SkippedFrames` reaches zero in a call to `TimeCallback()`, a video frame is read linked into the list, and `SkippedFrames` is re-assigned to `SkipFrames`. This process is described in figure 7.4.

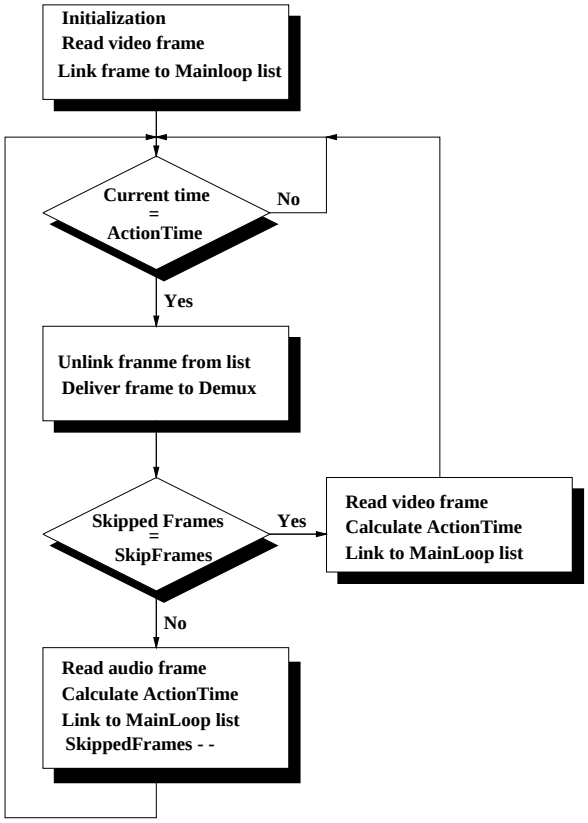

Figure 7.4: The synchronization process.

7.3 The `decompress` Subsystem

The `decompress` subsystem is responsible for decompressing JPEG compressed frames into the appropriate image format. `decompress` is based on the *Independent Jpeg Group* (IJG) software JPEG library [IJG94]. The library provides a framework for compression and decompression of images, of which the decompression part is utilized in jpegvid. The IJG library is, as far the author of this report is aware, the fastest freeware implementation of JPEG compression and decompression. The implementation of the library is perfomed in *C*, which is linked to the C++ code of `decompress` at the object level.

The reason for doing decompression in a separate process is that the time spent on decompression varies from frame to frame, as explained in section 4.3. If decompression were included in the `jpeg_control` process, synchronization would be disturbed. A frame which spent a longer time decompressing than expected, would cause delay or impose errors in the `Mainloop` object. On the other hand, the cost of the selected solution is the additional overhead due to context switches and communication between the processes.

The main object in `decompress` is the `Decompress` object. In addition there are `SharedMemory` and `MessageQueue` objects to handle the shared memory buffers and communication, respectively. These objects are merely convenience object encapsulations, and are described in section 7.5.

The `decompress` process is a child process of `jpeg_control`, and the necessary parameters, like size, q-factor, scale ratio, color/grayscale, the addresses of the shared memory buffers are given as input parameters from `jpeg_control`. At initialization `decompress` reads the default Quantization Tables and Huffman Tables (see section 2.4.3). The Quantization Tables are scaled by the q-factor if this is not the default. The Huffman Tables are the same for all videos, and are not modified. Subsequently the appropriate decompression parameters are set based on the input parameters from `jpeg_control`.

The actual decompression of JPEG frames is performed in the `DoDecompress()` method. This method relies on a IJG library function called `jpeg_read_scanlines()`, which takes decompressed JPEG data as input and produces raw RGB data, colormapped data or grayscale data output, dependent of the requested format. The complete JPEG decompression process, including post-processing, as described in section 2.4 and 4.6, is encapsulated by `jpeg_read_scanlines()`. All preferences are determined by setting appropriate decompress parameters at initialization.

As illustrated in figure 6.1 page 64, `decompress` uses two shared memory buffers; one input buffer for compressed data and one output buffer for decompressed image data. In order to read data, a `source manager` is registered with `jpeg_read_scanlines()`. The source manager is responsible for feeding `jpeg_read_scanlines()` with data. The output from `jpeg_read_scanlines()` is copied directly into the output buffer.

The `DoDecompress()` method is called every time `jpeg_control` puts a new frame into the input shared memory buffer. `decompress` does not interfere in which frames are to be decompressed. This means that only `jpeg_control` supervises synchronization; data flow and synchronization is *transparent* to the `decompress` process.

The operations of `decompress` is summarized in figure 7.5.

7.4 The jpeg_display Subsystem

The `jpeg_display` process draws images in the display window on the screen. Image data are read from a shared memory buffer, written by `jpeg_control`. As for the `decompress` module, all synchronization and data flow is transparent to `jpeg_display`. The reason for doing the displaying in a separate process is the same as for `decompress`; a delay in drawing would delay synchronization or corrupt jpegvid. This is explained in section 4.2.2.

At startup, `jpeg_display` is given the displaying parameters by `jpeg_control`. During initialization, the following tasks are performed:

- Selection of the appropriate visual. The *visual selection hierarchy* is presented in section 5.3.

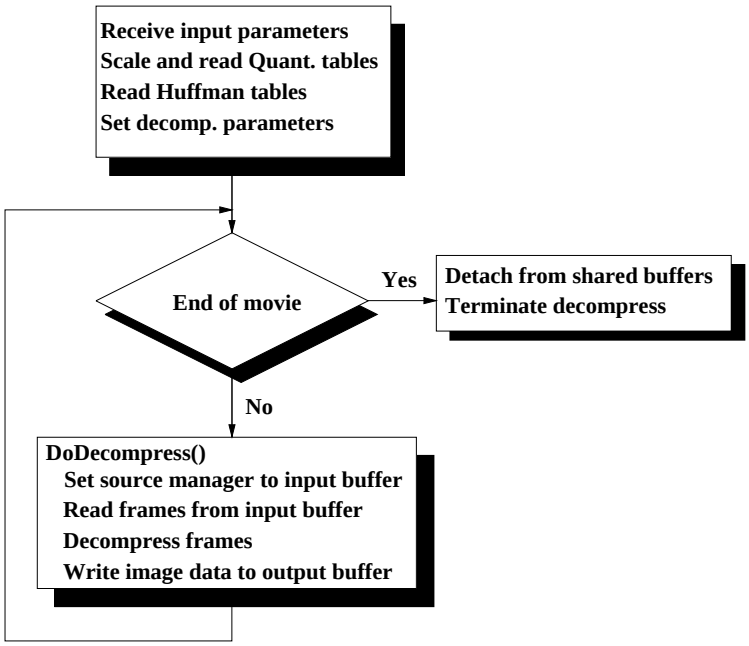

Figure 7.5: The `decompress` process.

- Creation and installation of the colormap. The colormap is different for the various visuals; 8-bits GrayScale uses a grayscale colormap (R=N, B=N,C=N for 0<N<256). 8-bits PseudoColor uses a default colormap which is used for all video sequences when color quantizing is applied. A 24 bits TrueColor does not require the colormap to be set to specific values. A discussion of colormaps and visuals is given in section 4.2.1.

- Determine whether to use Shared Memory Extensions to X (MIT-SHM) or conventional XLib functions. MIT-SHM facilitates faster drawing than the latter. This is discussed in section 4.2.2.

After the initialization and creation of the display window, `jpeg_display` enters an X Mainloop and waits for `jpeg_control` to trig the drawing of an image. `jpeg_control` trigs `jpeg_display` by using *XProperties*. This is further explained in section 7.5.

`jpeg_display` uses *XImages* [Oreilly90] and the XLib or MIT-SHM associated functions `XPutImage()` / `XShmPutImage()` for drawing images on the screen. In the XLib case, the XImages are stored in client memory, and copied to the X Server when drawing. When using MIT-SHM, the client and X Server shares a memory segment, and thereby facilitating fast display of images.

Section 5.1.1 described the keyboard character control alternative in jpegvid, giving the user the opportunity to control jpegvid by keyboard characters when the mouse pointer is inside the display window. These control events are transferred to `jpeg_control` by using XProperties, as well.

Figure 7.6 summarizes the operations of the `jpeg_display` module.

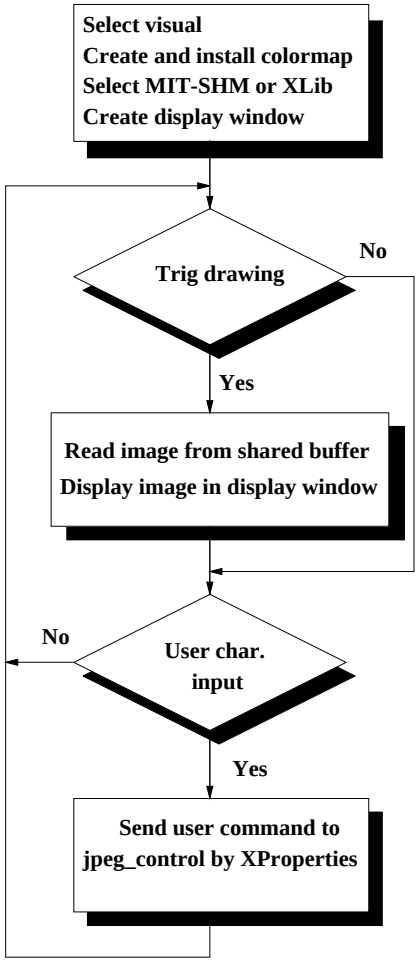

Figure 7.6: The `jpeg_display` process.

7.5 Communication Between Processes

The jpegvid application utilizes three methods for communication between the three processes:

- *Message Queues* to exchange small data volumes.
- *Shared Memory* to exchange image data.
- *XProperties* to send XEvents and small data volumes, using the event handling capabilities of X Windows.

7.5.1 Message Queues

Message Queues are used for exchanging message packets between processes using a queue mechanism. The data in packets arrives in the same order as it was sent, however, the receiver can determine in which order the distinct *packets* are received. Message Queues are a part of UNIX System V IPC[1].

[1] Interprocess Communication

jpegvid uses two Message Queues to exchange data between processes, a queue between each pair of processes. A class called `MessageQueue` has been defined, and two methods, `SendMessage()` and `GetMessage()` has been created to send and receive messages. A message consists of a `type` and a variable-length character string. In order to exchange messages, a simple `protocol` has been defined.

The message types are used by processes to identify messages. `MessageQueue` defines the following message types:

- **INIT_MESSAGE_ONE**:
  initialization messages
- **INIT_MESSAGE_TWO**:
  initialization messages
- **SHOW_FRAME**:
  message from `jpeg_control` to `jpeg_display` to tell that a frame is ready to be drawn.
- **FRAME_SHOWED**:
  message from `jpeg_display` to `jpeg_control`, telling that `jpeg_display` is finished drawing an image.
- **READ_FRAME**:
  message from `jpeg_control` to `decompress`, telling that a new frame is put in shared memory for `decompress` to read.
- **WRITE_FRAME**:
  message from `jpeg_control` to `decompress`, telling that decompress can write decompressed data. in the buffer.
- **RELEASE_FRAME**:
  message from `decompress` to `jpeg_control`, telling that a frame is read from shared memory.
- **FRAME_DECOMPED**:
  message from `decompress` to `jpeg_control`, telling that a frame has been decompressed.

The reason for using two initialization message types is that both processes may send initialization messages, and a process should not receive its own message.

Further information about Message Queues can be found in [Curry91].

7.5.2 Shared Memory

Shared Memory is also a part of UNIX System V IPC. Shared Memory permits two or more processes to share a segment of virtual memory, and use it as if it were actually a part of the process. This is the fastest way for processes to exchange large data volumes, and suits the exchange of compressed and uncompressed images between the three processes.

A class called `SharedMemory` is encapsulating the UNIX Shared Memory primitives. Creating a `SharedMemory` object involves creating a shared memory (if it has not been created already) and attaching to it. When a `SharedMemory` object has been instantiated, two methods may be called: `GetShmid()` to get the identifier of the shared memory segment and `GetShmaddr()` to get the start address of the segment.

jpegvid uses three `SharedMemory` objects to exchange data, as shown in figure 6.1 page 64:

- A segment exchange compressed JPEG data between `jpeg_control` and `decompress`.
- A segment to exchange decompressed image data between `decompress` and `jpeg_control`.
- A segment to exchange the image data between `jpeg_control` and `jpeg_display`.

Further information about Shared Memory can be found in [Curry91].

These Shared Memory segments must not be confused with the Shared Memory that MIT-SHM uses to exchange data between the X Client and the X Server, described in section 7.4.

7.5.3 XProperties

When using the X Window system, applications are *event driven*, ie. applications are controlled by sending and receiving various types of *XEvent*s. XProperties may be used to send events *and* exchange data.

Every window of an X Client has an associated set of XProperties in the X Server. Data is exchanged between applications by changing or setting an XProperty, by means of the `XChangeProperty()` function. `XGetWindowProperty()` is used to read XProperties. When an XProperty is changed for a window, a *PropertyNotify* event is generated for the window. The PropertNotify event may be caught by a window, and appropriate action is taken in a *callback function*.

The jpegvid application uses XProperties for communication between `jpeg_control` and `jpeg_display`. `jpeg_control` changes an XProperty in the `jpeg_display` display window every time a new image is ready for drawing, and thereby *trigs* the `jpeg_display` process. Similarly, `jpeg_display` changes an XProperty in the `jpeg_control` control panel window when character controls are used to control jpegvid, as described in section 5.1.1.

In order for applications to change XProperties of other windows, the window identifier (ID) must be known. This is achieved by storing the window IDs of `jpeg_control` and `jpeg_display` in XProperties of the *root window*. All applications of running on the same X Server knows the ID of the root window; therefore information may be exchanged by changing XProperties of the root window. However, to catch the PropertyNotify events, XProperties of the actual window must be changed. Consequently, after reading the window ID from the root window, XProperties of the window that is to receive PropertyNotify events are changed. This scheme is depicted in figure 7.7.

The reason for using both Message Queues and XProperties for interprocess communication, is that they provide communication on different *levels*. Message Queues are low-level communication, and can not be used for event handling. XProperties are higher-level communication, and is used in connection with XLib functions and event handling. For instance, the trigging of `jpeg_display` by `jpeg_control` is not possible by means of message queues.

Xproperties, event handling and associated issues are described in [Young90] and [Oreilly90].

7.6 File Format

The file format for the audio and video files are shown in figure 7.8. Every frame has a *frame header*, which is succeeded by a the frame data. The frame header contains

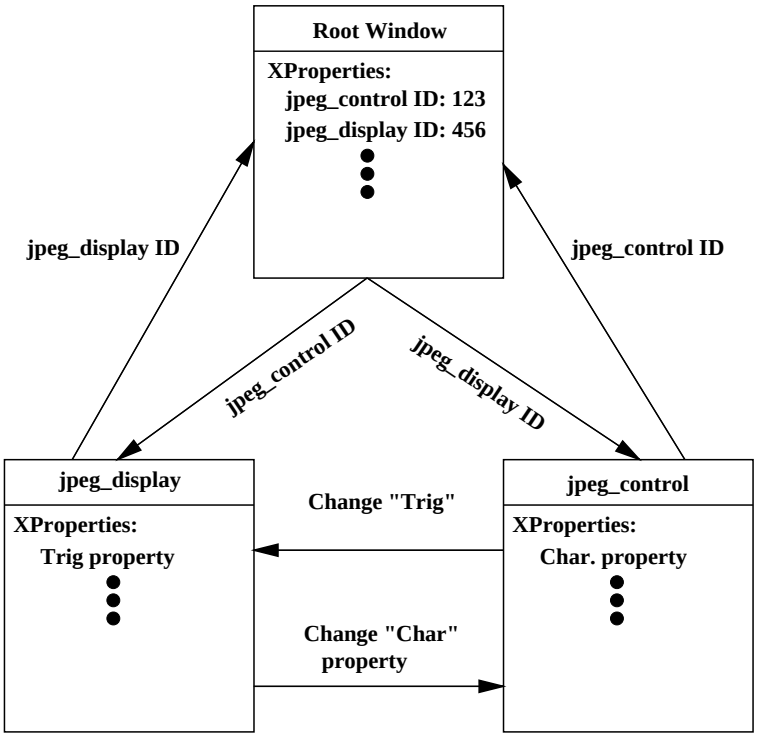

Figure 7.7: Communication by means of XProperties in jpegvid.

information for the succeeding frame:

- `Data type code`: Distinguishes between audio and video frames.
- `Time value`: The relative time value of the frame, ie. when it should be played. Consists of a `sec` (seconds) and a `usec` (micro seconds) part.
- `Sequence number`: The relative sequence of the frame in the frame sequence.
- `Data size`: The size of the succeeding frame data.

The first frame data of the video file consists of a *initialization packet*. This is not present in the audio file. The initialization packet contains the basic information of the video, namely:

- `qfactor`: The qfactor for JPEG compression.
- `width`: The width of the video frames.
- `height`: The height of the video frames.
- `interval`: The duration between two adjacent frames in the frame sequence.

The offset file format is illustrated in figure 7.9. An offset file simply consists of a mapping from sequence number to offset in the data file. This makes random access into the data file possible.

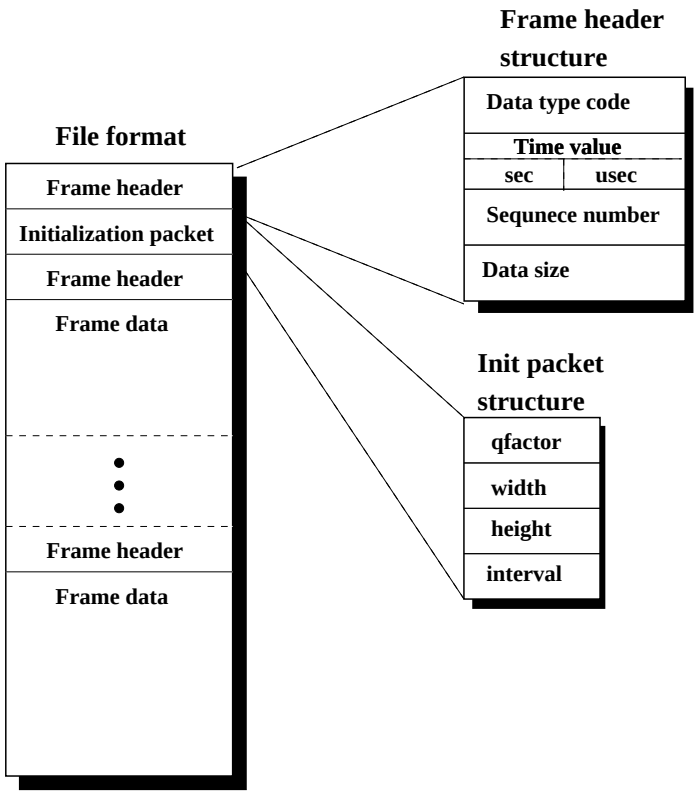

Figure 7.8: File format of data files.

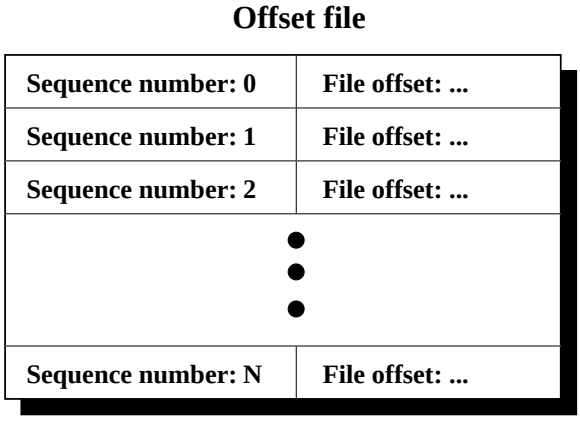

Figure 7.9: Offset file format.

Chapter 8

Experiments and Testing

In this project a software digital video player has been developed. Tests and experiments have to be performed in order to gather experience and point out important characteristics of the system. The jpegvid video player has some options which can be used to trade off image quality against higher frame rates. These options are tested in order to find the effects of setting various parameters. The key point of testing is the produced *frame rate*. Also, jpegvid varies the frame rate during playback, due to variations of frame complexity and CPU-load. This chapter describes the tests that has been performed on the *jpegvid* application, which is documented in previous chapters. Each test is put in a context, explaining which field of the system that is addressed and the motivation for the test. This is followed by the test results.

8.1 Frame Rate and Discarded Frames

The most important characteristic of a video player is the *frame rate* that is produced. The frame rate is discussed in section 4.3, and the *skip frame approach* is explained. The problems when determining the frame rate is that the processing time varies from frame to frame, and this leads to *delay jitter*, as explained in section 4.3. jpegvid uses a sliding window technique to vary the frame rate dynamically during playback of video, also explained in section 5.2.

The user may set some input parameters to jpegvid, in order to trade off image quality against speed. The input parameters are discussed in section 5.1.2. In the succeeding sections, the the result of setting varying the available parameters is investigated. The parameters are tested separately, ie. only one parameter is varied at a time.

All the tests in this section were performed on a *SPARC-LX* computer on a local login. This means that MIT-SHM is utilized (see section 4.2.2). Section 8.2 below tests running of jpegvid on other computers.

The experiments in this section were performed by playing video and gathering information about the playback. All experiments are presented by means of a table and a graph plot.

The graph plots show the variations in *number of skip frames* during playback of a video. The frame rate is inverse proportional to the number of skip frames with respect to the original frame rate (see section 4.3)[1]. That is:

[1] The original frame rate is 20 frames per second for all movies used in the experiments

$$\text{Frame Rate} = \frac{\text{Original Frame Rate}}{\text{Skip Frames} + 1} \tag{8.1}$$

For a video with original frame rate of 20 frames per second, a skip frame number of 20 of yields 1 frame per second, while skipping 5 frames means 4 frames per second.

The tables summarizes the information of the graph plots. The information listed in the tables is:

- **Avg Skip**: The average number of frames skipped during playback of the video. The average skip is calculated by *weighting* the individual skip numbers by their associated duration (ie. the duration a frame rate was maintained before changing).
- **Avg fps**: The average frame rate in frames per second. This number is calculated by equation 8.1 and the average skip.
- **Max Skip**: The maximum number of skip frames during movie playback.
- **Min Skip**: The minimum number of skip frames during movie playback.
- **Disc** Number of discarded frames, ie. the number of frames which were not decompressed or displayed in time (see section 4.3).
- **Disc %**: The percentage of discarded frames with respect to the number of frames that should have been displayed.

8.1.1 Scaling the Frames

Scaling down frames to a size less than the original frame size results in faster decompression of frames. The videos available all have a original size of $400x296$, and scaling down by all possible values have been tested for full color and grayscale output. Full color means that the `-color -depth 32` parameters were set; grayscale means specifying `-grayscale -depth 8`[2]. The IDCT algorithm was set to `-idct fast` in all the tests.

Table 8.1 shows the results of the tests.

Full color output

Scale	Size	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
1/1	400x296	18.8	1.0	25	17	15	6.1 %
1/2	200x148	13.3	1.4	19	9	18	5.2 %
1/4	100x74	9.5	1.9	17	4	27	5.4 %
1/8	50x37	6.2	2.7	14	2	38	4.7 %

Grayscale Output

Scale	Size	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
1/1	400x296	9.3	1.9	14	8	41	20%
1/2	200x148	7.6	2.3	12	6	38	6.6%
1/4	100x74	6.0	2.8	11	2	60	7.7%
1/8	50 x37	4.6	3.5	10	11	78	7.7%

Table 8.1: Various scale sizes with full color and grayscale output.

[2]Actually, jpegvid automatically selects a depth of 8 when grayscale output is wanted.

The tests show that scaling down frames produces a higher frame rate, as expected. The $400x296$ size image produces the lowest frames rate; 1 frame per second for full color and 2 frames per second for grayscale output. Note that scaling down the images to the half does note double the frame rate. For example, for color output, the difference is only 0.4 frames per second. This partially because Huffman entropy decoding must be performed in any case, and some of the processing is always performed.

The maximum frame rate obtained was when running grayscale video scaled by 1/8; 3.5 fps was achieved. This size is too small to be useful for watching video, unfortunately. It is possible to get an impression of what is being played, but reading text etc. is not possible.

Figure 8.1 shows the variations of the skip frames along the y-axis, and the frame number along the x axis. For color output, the differences between the various frame sizes can easily be seen. Note that the form of the curves are very similar. From this we can conclude that frames around 3000 to 3500 are relatively easy to decompress, while the frames around 3700 are complex and heavy to decode.

For grayscale output the lines are not so easy to see. This emphasizes the point that the variations in processing times of frames are unpredictable. For instance, the $100x74$ plot starts with the highest frame rate (lowest number of skip frames), but is heavily increased after ca. 1000 frames.

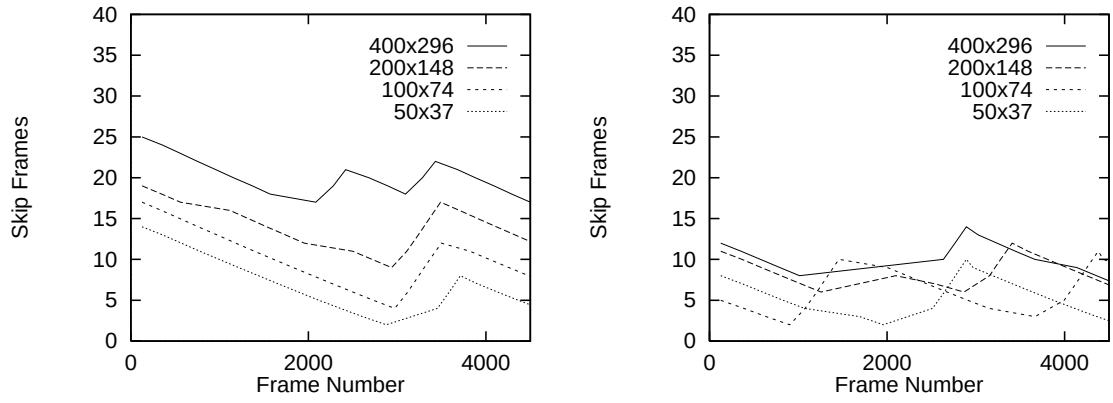

Figure 8.1: Variation in skip frames for various sizes. Color output to the left, grayscale to the right.

8.1.2 IDCT Algorithm

There are three possible IDCT algorithms available in for jpegvid - `integer`, `fast` and `float` - as described in section 5.1.2. Generally, the `fast` option is recommended. Table 8.2 shows the results of playback with the three algorithms for full color and grayscale output. All tests were performed with the original size ($400x296$) of the movies.

As expected, the `fast` algorithm produced the highest frame rate. However, differences are not very large, as seen from the table. For full color, the frame rate differs only by 0.1 frames per second, while for grayscale output, the gain by using the `fast` algorithm was 0.5 fps.

Note particularly that `int` and `fast` options produce virtually the same frame rates. This means that the floating number calculations on the computer are not very much more expensive than integer calculations.

Full color output

IDCT	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
INT	20.6	0.9	26	20	19	8.4%
FAST	18.8	1.0	25	17	15	6.1%
FLOAT	19.9	0.9	25	21	20	9.5%

Grayscale output

IDCT	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
INT	12.5	1.5	20	9	25	6.8%
FAST	9.1	2.0	13	7	41	8.5%
FLOAT	13.6	1.4	21	10	22	6.4%

Table 8.2: Varying the IDCT algorithm, full color and grayscale output.

Figure 8.2 shows the graph plots of the tests. The plots suggest also show that the `fast` option is only a little faster than the two other options.

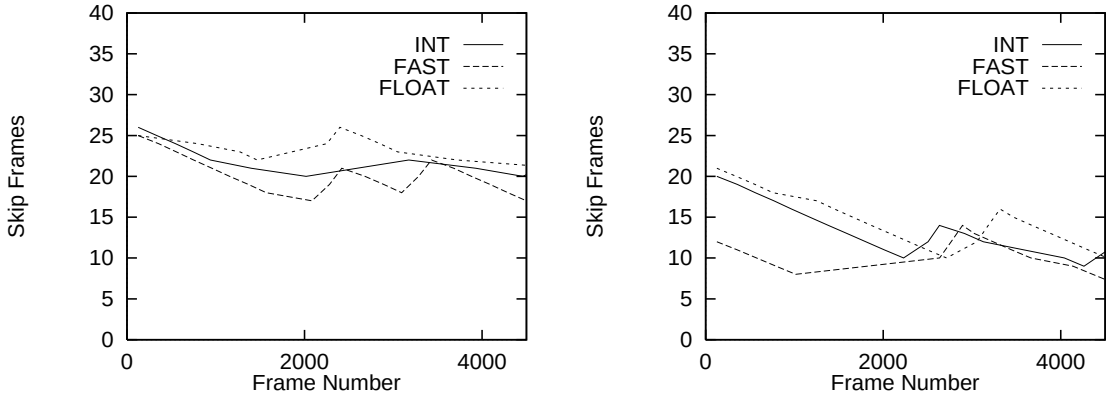

Figure 8.2: Variation in skip frames for various IDCT algorithms. Color output to the left, grayscale to the right.

8.1.3 Varying Visuals

Setting the `-color`, `grayscale` and the `-depth` switches determines (in cooperation with the machine hardware) which visual is selected for jpegvid. The visual selection algorithm is displayed in section 5.3. Grayscale output is generally faster than full color output, according to the previous tests. In this section the above parameters are explicitly tested. Note that the `-color -depth 8` setting requires color quantizing (see section 4.6.4).

In this test, the idct algorithm was assigned to `fast`. Two image sizes were used: $400x296$ and $100x74$. The results are found in table 8.3.

As seen from the tables, using a 8 bits PseudoColor visual produced the lowest frame rates for size $400x300$. This is expected, because *color quantization* is performed in this case. However, the difference between TrueColor and PseudoColor is not very large. This suggests that color quantizing is not very expensive.

Grayscale output produces the highest frame rate. This is also expected, because only one of the three image components is processed. Grayscale output doubles the frame rate, both for size $400x296$ and size $100x74$. The reason why it is not increased by a factor of

Size = $400x296$

Visual	Dpt	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
TrueColor	32	18.9	1.0	25	17	15	6.1%
PseudoColor	8	22.8	0.84	28	21	23	11.1%
GrayScale	8	9.1	2.0	14	7	41	8.5%

Size = $100x74$

Visual	Dpt	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
TrueColor	32	9.5	1.9	17	4	27	5.4%
PseudoColor	8	9.0	2.0	17	5	28	5.2%
GrayScale	8	6.0	2.8	11	2	60	7.7%

Table 8.3: Varying color visuals, size = $400x300$ and size = $100x74$.

three, is that Huffman decoding and some other work is performed in any case.

Figure 8.3 shows the graph plots for the tests. For size $300x400$, the differences between visuals are easy to see. The plot for size $100x74$ is more chaotic, but at least the grayscale graph is the one with the lowest skip number most of the time.

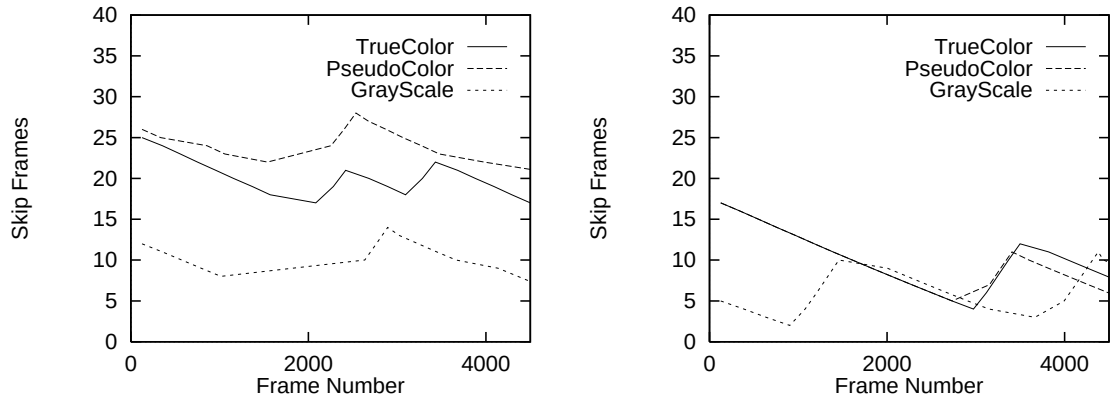

Figure 8.3: Variation in skip frames for various visuals. Size 400x300 to the left, size 100x74 to the right.

8.2 Different Computers

The above tests were all performed on a *SPARC-LX* computer. In this section, some experiments were performed on two other computers are presented. A *SPARC-SS20* running on *SOLARIS 2.3*, and a *SPARC-10/30* running *SunOS-4* were tested. Three experiments were performed, varying the `scale`, `color/grayscale` and `depth` parameters, on each of the computers:

1. 32 bits color (full color), size 400x296.
2. 32 bits color (full color), size 200x148.
3. 8 bits grayscale, size 200x148

The experiments were performed on remote logins, using NFS. This means that MIT-SHM was not used. Also the IDCT algorithm was set to `fast` in all experiments.

The results are listed in table 8.4.

Full color output, size=$400x296$

Computer	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
SPARC-SS20	21.5	0.88	29	9	60	25.3%
SPARC-10/30	23.3	0.82	30	14	33	15.9%
SPARC-LX	25.0	0.67	30	23	26	14.1%

Full color output, size=$200x148$

Computer	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
SPARC-SS20	6.4	2.7	11	6	49	7.4%
SPARC-10/30	7.3	2.4	11	6	36	6.0%
SPARC-LX	13.3	1.4	20	10	26	7.6%

Grayscale output, size=$200x148$

Computer	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
SPARC-SS20	3.7	4.3	9	1	82	6.4%
SPARC-10/30	4.6	3.6	8	3	62	6.7%
SPARC-LX	9.6	1.9	17	6	20	4.2%

Table 8.4: Comparison of computers

In all the tests, the SPARC-SS20 produced the highest frame rate as expected, followed by the SPARC-10/30 and the SPARC-LX computer.

For full color output and size $400x300$, the differences between the three computers are very small. This is due to the drawing of images on the screen. MIT-SHM is not used, and conventional XLib functions are utilized instead. The visual effect of this is that the screen is updated only one half at a time, in addition to the low frame rate.

The maximum frame rate was achieved by the SPARC-SS20 computer with grayscale output and a size of $200x196$, 4.3 fps was achieved. For both experiments with size $200x196$, the SPARC-SS20 doubles the frame rate produced by the SPARC-LX computer. The difference in frame rate between the SPARC-10/30 and the SPARC-SS20 is not significantly large for neither of the experiments. This suggests that the processing power of the two computers quite similar.

8.3 Discarding Frames

As explained in section 4.3, the video player accepts discarding of frames up to 15% during the latest 10 seconds. In all the experiments above, except one, the discarded frames percentage is below this limit; the percentage is stable around 5% to 8%. This means that the sliding window technique works well with respect to the discarding of frames. Note again that the discarded frames measure is relative to the frame rate, ie. it is calculated against the number that *should have been produced* with the current frame rate.

In table 8.4, full color output with size $400x296$ has notably higher percentage of discarded frames than the in rest of the experiments. This is because MIT-SHM is not utilized in this experiment, and XLib is more unstable for this output type. Consequently, the *display delay jitter* is higher than in the other experiments.

8.4 Fastest Setting

From the tests above, we can conclude that the jpegvid executes fastest with grayscale output, least size and the fast IDCT algorithm. In addition, the SPARC-SS20 computer proved to be the fastest. In order to achieve as high frame rate as possible, the fastest settings have been tested on the SPARC-SS20 computer. Scaling to 1/4 and 1/8 have been tested for both full color and grayscale output, while using the `-idct fast` parameter. The results are shown in table 8.5.

Full color output

Size	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
$50x37$	1.4	8.3	6	0	106	2.8%
$100x74$	3.0	5.0	9	1	104	6.6%

Grayscale output

Size	Avg Skip	Avg fps	Max Skip	Min Skip	Disc	Disc %
$50x37$	1.2	9.9	6	0	197	5.8%
$100x74$	4.1	3.9	8	0	66	5.3%

Table 8.5: The fastest execution results

The $50x37$ grayscale output produced an average frame rate of 9.9 fps, or half of the original frame rate of 20 fps. It is interesting to note that the frame rate was 20 fps (skip frames = 0) during parts of the movie, but such a high frame rate could not be maintained. The $100x74$ size grayscale image achieved a frame rate of 20 during parts of the video as well, however the average is much lower in this case.

For color output of size $50x37$ the frame rate was also producing 20 fps during parts of the playback. The average frame rate is almost similar to the corresponding grayscale video output. It must be emphasized that a frame size of $50x37$ is not appropriate for quality video playback; it is convenient for testing, though.

The comparison of grayscale and full color with frame size $100x74$ shows, surprisingly, that full color produces a higher frame rate than grayscale. The tests has been performed several times to remove coincidental results. This example merely shows the uncertainty of a software video player; the results vary continuously and you never know the frame rates beforehand. However, it is safe to say that, if the experiment were performed a sufficiently number of times, the full color output would produce a lower frame rate than grayscale.

8.5 Variations in Frame Rate

The sliding window algorithm of jpegvid makes the frame rate vary according to the number of frames the computer can process. A major goal is not to let the frame rate vary too much.

A general tendency is that the frame rate initially is set too high. In all the experiments, the frame rate decreases the first time.

Also, the frame rate varys a lot, as can be seen from the graph plots and the *Max Skip* and *Min Skip* values of the tables above. Generally, it seems like jpegvid decreases the frame rate until it reaches a limit, and then the frame rate is increased rapidly. This suggests two shortcomings of the system:

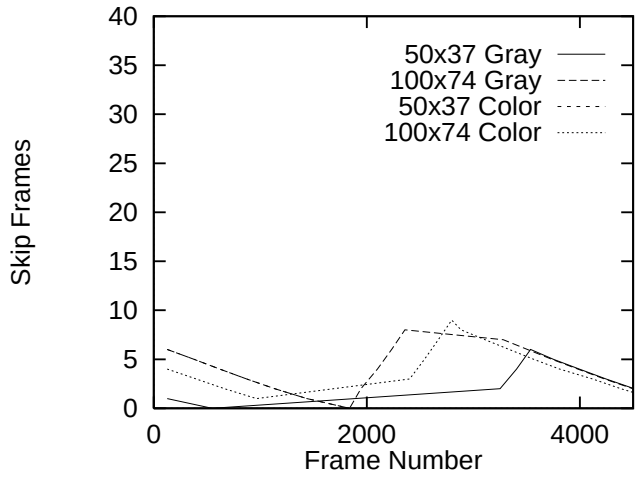

Figure 8.4: Fastest settings.

- The variable CPU load on the computer is difficult to take into account.
- The sliding window algorithm increases and decreases the frame rate too fast.

8.6 Test Summary

Generally, the frame rates produced in most of the experiments are too low if the requirement is high quality video playback. The frame rate is for almost all the tests so low that users easily see when a new frame is drawn. The test with the fastest settings above produced near real-time frame rates, but in this case the image quality is not very high. The viewer only get a shallow impression of what is going on on the display window.

Also, the frame rate generally varies too much. This partially due to the software processing of JPEG decompression. Other algorithms for varying the frame rates should also be tested, in order to find out whether the sliding window technique is well suited for jpegvid.

Chapter 9

Conclusions and further work

This project has investigated digital video in general, and particularly a software implementation of a JPEG video player. This chapter contains conclusions from the work, and suggests some areas for further work and study.

9.1 Conclusions

In this thesis digital video has been studied. Two main issues concerning digital video has been investigated; compression of video data and network transmission of continuous media, ie. audio and video data. These issues are highly significant with respect to digital video, and other research within these areas has been studied and presented.

Particularly, the JPEG standard for still image compression has been investigated for the purpose of using JPEG compression and decompression in a digital video player. This knowledge has been used when designing and implementing a software based Motion JPEG video player called *jpegvid*. The main strengths of jpegvid are:

- Dynamic variation of the produced frame rate according to the complexity of compressed frames and the capacity of the CPU.
- The jpegvid video player provides possibilities for trading off output image quality against produced frame rate. The options include color/grayscale output, scaling down the original frames, and three different options for IDCT algorithm.

The jpegvid video player has been used in experiments to explore the characteristics of JPEG decompression utilized in digital video. Generally, we can conclude that JPEG is not suited for software processing of digital video. Decompression is too slow, and the decompression delay jitter is too high. It is not possible to achieve high quality output with a reasonable frame rate. Only for the lowest quality settings of jpegvid, the frame rate was close to real-time playback, and even with these settings, it was not possible to maintain a constant high frame rate.

In spite of this, jpegvid is not totally useless. It can be used for browsing through video sequences if the user does not have too high requirements to playback quality. The user interface allows playback synchronized with audio, stepping, winding and random jumping within a video sequence. In addition, the advances in hardware technology lead to faster computers, and in a few years it may be possible to run jpegvid with real-time high quality output.

With respect to network transmission, a lot of research has been performed by other projects. Network transmission of continuous media requires Quality of Service parameters and guaranteed delivery, which is not supported by most common network technology today. Especially, the development of ATM networks is interesting concerning transmission of digital video. Also, other network protocols have been designed and implemented to support transmission of continuous media.

Software video is vulnerable to network transmission delays and delay jitter. The special requirements of software digital video has been investigated, but not implemented or tested in this project.

Working with the area of digital video has been very interesting, but simultaneously most time-consuming. A lot of work has been put into studying, report writing and implementation. Still, there remains much work that could be performed within this area.

9.2 Suggestions for Further Work

Working with a software digital video player has spawned many ideas for further work. This sections summarizes a few areas where others may continue working from this project

9.2.1 Network Transmission

Network transmission of video data from a remote host to the jpegvid player is not implemented in this project. Network protocols are investigated, and references to other work concerning transmission of continuous media is provided. A network solution for video transmission is desirable for user to access video data from a remote site.

9.2.2 Gaining Speed

The q-factor is a property of each video film, and is used to set different degrees of compression (see section 2.4). The q-factor specifies the degree of compression for JPEG compressed images. Images which are highly compressed (low q-factor), are decompressed faster than images compressed with a high q-factor. The videos experimented with in this work all have the same q-factor. It would be interesting to see if it is possible to achieve a higher frame rate using images compressed with a lower q-factor.

Also, all videos used in tests have the same original frame size. As seen from the previous chapter, jpegvid performs badly with such large image sizes. Scaling down images is a option of jpegvid, but the effect on decompression of a smaller original frame size is probably better than down-scaling to the same size.

9.2.3 No Compression

An interesting experiment would be to decompress a complete video and store it uncompressed on hard disk. Playback of this video would not require decompression at all, and the question would be whether reading the files from disk could be accomplished.

A video with no compression requires a lot of storage. However, storing a video sequence as frames with 8 bits Pseudo Color and size $200x148$ requires only 29600 bytes per frame, only about three times as much as the JPEG compressed frames stored in the video sequences used in this project.

A tool could be implemented to decompress complete movies as an 'on-demand' service. The architecture of jpegvid, where the decompression is performed as a separate process, would only need minor changes to handle such video sequences.

9.2.4 Integrating jpegvid with VideoSTAR Tools

As mentioned in the introduction, the Database Group at NTH already has some tools for describing video, searching in video information databases and and browsing of video, referred to as the VideoSTAR system. It is perfectly possible to integrate jpegvid with these tools.

Appendix A

VoDoo Copyright Statement

A.1 VoDoo Authors

VoDoo is not really an acronym, but if you insist it could probably have some relations to video on demand and object orientation, besides that it looks like voodoo.

VoDoo is written by Gisle Aas <aas@nr.no> and Arild Bekkadal <arild@nr.no> at Norsk Regnesentral based on code written by Brd Hfjeld at Televerkets Forskingsinstitutt.. The work was done during the spring and summer 1994.

The work was sponsored by Televerkets Forskningsinstitutt through contract "9F4 050 - MultiTorg - Brebnds videokommunikasjon".

A.2 Copyright statement

Appendix B

Class Descriptions

This appendix contains descriptions of some of the classes. For further reference, see the VoDoo system, which jpegvid was based on [VoDoo94].

B.1 Decompress

Class Decompress

Description:

Decompress is a JPEG decompression application. The process reads compressed frames from one shared memory segment, and puts decompressed data in another shared memory segment. Communication is managed by UNIX message queues, implemented by class MessageQueue. Shared memory is managed by objects of class SharedMemory.

The basis for decompression implementation is the Independent JPEG Group software library [IJG94].

Various parameters are set for decompression when initializing, including:
input colorspace
output colorspace (color/grayscale)
output components
IDCT method (integer, fast, floating point)
output colors (when quantizing)
scaling (1/n)

B.1.1 Public Methods

B.1.1.1 Decompress()

```
Decompress(msq_key, msq_size)

key_t msq_key;
size_t msq_size;
```

Arguments:

msqid:

Specifies the key of the message queue to connect to
msqsize:
Specifies the maximum size of a message

Description:

Sets up a JPEG decompression error handler to handle decompression errors. A decompress structure is created. Connects to the message queue and attaches to the shared memory segment where compressed JPEG frames are read from. The message queue and the shared memory segment are both created by the jpegcontroll process. Initializing of the communication is perfomed.

IJG functions:

```
jpeg_std_error()
jpeg_create_decompress()
```

B.1.1.2 Decompress()

```
~Decompress()
```

Description:

Releases the JPEG decompression object. Disconnects from the message queues and detaches from the shared memory segments.

IJG functions:

```
jpeg_destroy_decompress()
```

B.1.1.3 ReadHeader()

```
void ReadHeader( vis_class, w, h, q, sc_num, sc_denom)

int vis_class;
int w;
int h;
int q;
int sc_num;
int sc_denom;
```

Arguments:

vis_class:
Denotes the visual class that determines the format of the data output.
w:
Original width of the image in pixels.
h:
Original height of the image in pixels.
q:
Specifies the Q-factor that the image is compressed with, and that is used when decompressing.

sc_num:
Specifies the scaling numinator of the output image.
sc_denom:
Specifies the scaling denominator of the output image.

Description:

ReadHeader reads the JPEG header table jfif_header, which contains the component specification, q-tables, Huffman entropy tables etc. First the height and width of jfif_header is adjusted according to the input parameters width and height. The q-table is adjusted likewise, according to the input q-factor. The decompression parameters are the set according to the visual class, the appropriate decompression attributes and the scaling specified by scale_num and scale_denom.

IJG functions:

```
jpeg_memory_src()
jpeg_read_header()
```

B.1.1.4 DoDecompress()

```
void DoDecompress()
```

Description:

This method does the actual decompression of JPEG compressed images. Compressed data are read from one shared memory segment and written to an output shared memory segment. First an abbreviated JPEG header is read (not containing qtables or Huffman tables). Decompression parameters has to be set on every new frame.

Adress and size of the input and ouput data is received on the message queue. When the decompression is finished, messages are sent on the queue about the decompressed frame and releasing of the read data. Timing of the decomprssion of frames is also performed here.

IJG functions:

```
jpeg_memory_src()
jpeg_read_header()
jpeg_start_decompress()
jpeg_read_scanlines()
jpeg_finish_decompress()
```

B.1.2 Public Variables

None

B.1.3 Private Methods

None

B.1.4 Private Variables

`struct jpeg_decompress_struct cinfo`
The JPEG decompression object that holds the information about the decompresion attributes. Used in most calls to IJG functions.

`struct jpeg_error_mgr jerr`
The JPEG library error handler struct.

`int row_stride`
Number of bytes in a row of the output image.

`size_t read_size`
Specifies the size in bytes of a compressed frame.

`size_t write_size` Specifies the size in bytes of a decompressed frame.

`JOCTET *readbuf_start`
Specifies a pointer to the start adress of a compressed frame to be read.

`JOCTET *writebuf_start`
Specifies a pointer to the staart adress of the output frame to be written.

`MessageQueue *mqueue`
Specifies a pointer to the object of the MessageQueue class that manages the communication with the jpegcontroll and jpeg_display processes.

`SharedMemory* shm_read`
Specifies a pointer to an object of the SharedMemory class that manages the shared memory segment that holds compressed frames to be read.

`SharedMemory *shm_write`
Specifies a pointer to an object of the SharedMemory class that manages the shared memory segment where decompressed data are written.

`int visual_class`
Specifies the visual class the defines the format decompressed data are written.

`int height`
Specifies theoriginal height of the output image in pixels.

`int width`
Specifies the original width of the output image in pixles

`int qfactor`
Specifies the qfactor which the image was compressed with, and that scales the qtable matrixes.

`int scale_num`
Specifies the scaling numinator of the output image.

`int scale_denom`
Specifies the scaling denominator of the output image.

`int abbrev`
Specifies if the image should use the abbreviated JFIF header or the complete JFIF header. In ReadHeader, abbrev is False, and the static table jfif_header is read. Later, abbrev is set to True, and the static_table jfif_pic_header is read.

B.1.5 Global Static Variables

`static char jfif_header[623]`
Specifies the complete JFIF header including qtables and Huffman tables. This table is read only the first time to set the qtables and Huffman tables of the cinfo struct.

`static char jfif_pic_header[53]`
Defines the abbreviated JFIF header that is read after the initializing reading in Read-Header. jfif_pic_header does not carry any qtables or Huffman tables, they are preserved in the cinfo struct.

B.2 JpegDisplay

Class: JpegDisplay

Description:

Decides whether to use X Shared Memory Extensions.
Selects the an appropriate visual.
Allocates colors for the visual from the default colormap.
Attaches to shared memory from the decompress process, and reads decompressed data from it.
Displays an XImage on a DrawngArea widget.

B.2.1 Public Methods

B.2.1.1 JpegDisplay()

```
  JpegDisplay(msqid, msqsize, videoWindow, p_visual, int depth
  int width, int height);
```

```
key_t msqid
size_t msqsize
Widget videoWindow
Visual* p_visual
int depth
int width
int height
```

Arguments:

msqid:
Specifies the key of the message queue to connect to
msqsize:
Specifies the maximum size of a message
videoWindow:
Specifies the DrawingArea widget to draw the XImage
p_visual:
Specifies a pointer to the visual to use
depth:

Specifies the depth of the XImage
width:
Specifies the width in pixels of the DrawingArea
height:
Specifies the height of the DrawingArea

Description:

Connects to the message queue specified by msqid to communicate with the jpegcontroll process. Attaches to a shared memory segment created by jpegcontroll, where the decompressed image data are written. A graphics context is defined, and the window id is registered as an XProperty in the X root window, so the jpegcontroll process can communicate. Creates a shared memory XImage if MIT X Shared Memory is supported, if not succesfil, a common XImage is created. Sends a message on the message queue telling that the process is ready for displaying images.

B.2.1.2 JpegDisplay()

```
~JpegDisplay()
```

Description:

Detaches from the shared memory segment. Frees allocated varables.

B.2.1.3 Initialize()

```
void Initialize()
```

Description:

Adds a property eventhandler. PutImage trigs the displaying of an image by changing a property of the video window. Adds callbacks to handle resizing and exposure

B.2.1.4 ShowFrame()

```
void ShowFrame()
```

Description:

This method is called when the application recieves a PropertyNotify event. Fetches the decompressed image adress and size from the message queue, and calls PutImage to draw the XImage on the videoWindow. Sends a message on the message queue when the XImage is put on the widget.

B.2.1.5 PutFrame()

```
\void PutFrame()
```

Description:

Draws the decompressed data on the screen (the videoWindow widget), using either XPutImage or XShmPutImage. Waits until the XImage is completely drawn on the screen.

B.2.1.6 ExposeEvent()

```
void ExposeEvent()
```

Description:

Callback function, which is called whenever the application receives an ExposureNotify event.

B.2.1.7 ResizeEvent

```
void ResizeEvent()
```

Description:

Callback function that is called when the application receives a ResizeNotify event.

B.2.1.8 DrawingAreaSelect()

```
void DrawingAreaSelect(event)

Xevent event
```

Arguments:

event:
Specifies the PropertyNotify event

Description:

Used to catch user character commands

B.3 Private Methods

B.3.0.9 Send()

```
void Send(command);

char *command
```

Arguments

command:
The command input on the drawing area

Description:

Sends a PropertyNotify to jpeg_control by changing a propert in the control window.

B.3.0.10 CreateShmImage()

```
int CreateShmImage()
```

Return value:

1 on success, 0 on failure.

Description:

Queries the X server to find out whether shared memory extensions to X is supported. If shared memory is supported, a shared memory XImage is created the X server attaches to the shared memory segment, and True is returned. If shared memory is not supported, False is returned.

B.3.0.11 CreateImage()

```
int CreateImage()
```

Description:

Creates an ordinary XImage. Returns True on success, False on failure

B.3.0.12 InstallXErrorHandler()

```
void InstallXErrorHandler()
```

Description:

Installs an X error handler to handle Shared memory errors. The X server does not handle shared memory errors in a proper manner

B.3.0.13 DeInstallXErrorHandler()

```
void DeInstallXErrorHandler()
```

Description:

Deinstalls the X error handler installed by InstallXErrorHandler.

B.3.0.14 PropEventHandler

```
PropEventHandler( w, client_data, event);

Widget w
XtPointer client_data
XPropertyEvent *event
```

Arguments:

w:
Specifies the widget of callback
client_data:
Specifies the callback client data that is sent with the event. event:
Specifies a pointer to the event

Description:

This method is called when the application receives a PropertyNotify callback. PutVideo trigs this method when a decompressed image is to be displayed on the screen.

B.3.0.15 RegisterWindow()

```
void RegisterWindow(w)

Widget w;
```

Arguments:

w:
The widget used to obtain the window id and the display pointer.

Description:

Registers the window ID in the root window so it can be accessed by the PutImage object.

B.3.1 Public variables

None

B.3.2 Private variables

`Display *display`
A pointer to the applications Display structure.

`Widget win`
The DrawingArea video window widget that draws the images on the screen.

`Visual *visual`
A pointer to the Visual used when drawing images. The visual contains information about the screen properties.

`XImage *xim`
A pointer to the XImage structure. This structure contains properties of the image, like depth, visual, height, width, RGB-masks, a pointer to the imaage data etc.

`Dimension win_width`
The width of the image in pixels.

`Dimension win_height`
The height of the image in pixles.

`int win_depth`
The depth (bits per pixel) of the XImage.

`unsigned int c_image_size`
The size of the image in bytes.

`int shm_flag`
The shm_flag is set to True if shared memory extensions to x are supported, False if not supported. This flag is set in CreateShmImage().

`MessageQueue *mqueue`
A pointer to the message queue object that is used for communication with the jpegcontroll process.

`SharedMemory *shmem`
A pointer to the shared memory object that manages shared memory where decompressed data is read.

`GC theGC`
The Graphics Context that is used when drawing on the screen in PutImage.

`int CompletionType`
CompletionType is used to determine when an XImage has been drawn when using XShmPutImage;

`XShmSegmentInfo xshminfo`
Specifies information about the shared memoey that is used by the X server in XShmPutImage.

B.4 PutVideo

class PutVideo: public IsoReceiver

Description:

This is the class wich is in charge of trigging jpeg_display. Checks if a frame has been decompressed copies it to the second shared memory buffer and trigs jpeg_display. In charge of discarding frames

B.4.1 Public Methods

B.4.1.1 PutVideo()

```
PutVideo(width, height)

int width
int height
```

Arguments: width:
Width of frame
height:
Height of frame

Description:

Class constructor

B.4.1.2 PutVideo()

```
~PutVideo();
```

Description:

Clean up the mess

B.4.1.3 Initialize()

```
void Initialize();
```

Description:

Initialize after header arrives

B.4.1.4 Deliver()

```
void Deliver(ChunkData cdata);
```

Called by demux when a frame should be displayed. Calls `ShowFrame`.

B.4.1.5 ShowFrame()

```
void ShowFrame();
```

Description:

Check whether decompress and jpeg_display is finished. If not discard frame. Else copy the frame from shared buffer 1 to shared buffer 2 and trig jpeg_display by changing the XProperty

B.4.2 Public Variables

None

B.4.3 Private Methods

None

B.4.4 Private Variables

`Widget win`
Window to update

`int initialized`
Is system inited?

`unsigned int video_width`
Dimension on JPEG video frames

`unsigned int video_height`
Dimension on JPEG video frames

`Dimension win_width`
Dimension on video vidget

`Dimension win_height`
Dimension on video vidget

`unsigned int c_image_size`
The size allocated for the JPEG image

`int must_refresh`
A flag which is set on refresh `int decomp_pid`
pid of decompress process

`int frame_no`
Current frame number

`int skipped_frames`
Number of discarded frames

`int displayed_frames`
Number of displayed frames

`int use_time`
Use `gettimeofday()`?

`MessageQueue *read_mqueue`
Queue to decompress process

`MessageQueue *write_mqueue`
Queue to jpeg_display process

`SharedMemory *shm_decomp_write`
Shared memory buffer for decompress to write

`SharedMemory *shm_jdisplay_read`
Shared memory buffer for jpeg_diplay to read

Bibliography

[Arai88] Y. Arai, T. Agui, M. Nakajima, *A Fast DCT-SQ Scheme for Images*, Transactions of the IEICE, E 71(11):1095, November 1988.

[Banerjea92] A.Banerjea, B.A.Mah, *The Real-Time Channel Administration Protocol*, Network and Operating System Support for Digital Audio and Video, Heidelberg Germany, 1991.

[Banerjea94] Anindo Banerjea, Edward W. Knightly, Fred L. Templin, Hui Zhang, *Experiments with the Tenet Real-Time Protocol Suite on the Sequoia 2000 Wide Area Network*, ACM Multimedia'94, San Fransisco, ACM Press, 1994.

[Bekkadal94] *Editing Tool for Digital Video*, Diploma Thesis, for the Computer Science Institute at the Norwegian Institute of Technology (NTH), autumn 1994, vol. 1 (1994)

[COMP-FAQ94] Jean-loup Gailly, *Comp.Compression Frequently Asked Questions*, `http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/part[1-3]`

[Curry91] David A. Curry, *Using C on the UNIX system*, O'Reilly & Associates, Inc., 1991

[Delgrossi92] L.Delgrossi, F.O.Hoffman, *ST-II for the Heidelberg Transport System: Protocol, Design and Implementation*, Technical Report, IBM European Networking Center, 1992.

[Feig90] E. Feig, E.Linzer, *Discrete Cosinus Transform Algorithms for Image Data Compression*, Proceedings Electronic Imaging '90 East, pp.84-7, Boston, MA (Oct.29-Nov.1, 1990).

[Ferrari91] Domenico Ferrari, *Design and applications of a delay jitter control scheme for packet switching internetworks*, Network and Operating System Support for Digital Audio and Video, Springer Verlag, 1991.

[Ferrari92] D.Ferrari, H.Zhang, D.Verma, *Design and Implementation of the Real-Time internet protocol*, Proceedings of the IEEE Workshop on the Architecture and Implementation of High Performance Communication Systems, Tuscon, Arizona, USA, February 1992.

[Ferrari92b] D.Ferrari, A.Gupta, M.Moran, B.Wolfinger, *A Continuous Media Communication Service and Its Implementation*, Globecom'92, Orlando, Florida, USA, 1992.

[Floyd75] R.Floyd, L.Steinberg, *An Adaptive Algorithm for Spatial Gray Scale*, Society for Information Display 1975 Symposium Digest of Technical Papers, 1975.

[Foley91] Foley, van Dam, Feiner, Hughes, *Computer Graphics: Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, 1991.

[Gall91] Le Gall Didier, *MPEG: A Video Compression Standard for Multimedia Applications*, Communications of the ACM, Vol.4, april 1991.

[Gonzalez92] Gonzalez Rafel C, Woods Richard E, *"Digital Image Processing"*. Edition 3, Addison-Wesley. 1990.

[Halsall92] Fred Halsall, *Data Communications, Computer Networks and Open Systems*, Addison-Wesley, 1992.

[HAN91] Hanko James, Berry David, Jacobs Thomas, Steinberg Daniel, *Integrated Multidata at Sun Microsystems*, Network and Operating System Support for Digital Audio and Video, Heidelberg Germany, 1991.

[Haugen94] Geir Haugen, *The MPEG video standards - background, functionality and use*, Diploma Thesis, for the Computer Science Institute at the Norwegian Institute of Technology (NTH), December 1994, vol. 1 (1994)

[Hilton94] M. Hilton, B. Jawerth, and A. Sengupta, *Compressing Still and Moving Images with Wavelets*,
tt FTP: /pub/wavelet/papers/varia/tutorial/tutorial.ps.Z

[Holte94] *Digital Television Archives*, Diploma Thesis, for the Computer Science Institute at the Norwegian Institute of Technology (NTH), autumn 1994, vol. 1 (1994)

[IJG94] Independent JPEG Group, *IJG Software JPEG Library*, Documentation and source code, version 5
`FTP: ftp.uu.net/graphics/jpeg/jpegsrc.v5.tar.gz`

[JFIF94] *JPEG File Independent Format*,
`http://icib.igd.fhg.de/icib/it/defacto/research/jfif/jfif.1.02.ps.Z`

[Johnsen94] Roger Johnsen, *Innføring av ATM in UNITs Nett*, Diploma Thesis, for the Computer Science Institute at the Norwegian Institute of Technology (NTH), autumn 1994, vol. 1 (1994)

[JPEG94] International Standardization Organization, *ISO/IEC IS 10918-1 (JPEG) - Inforamtion technology - Digital compression and coding of continuous-tone still picture: requirements and guidelines*, ISO/IEC IS 10918-1:1994(E).

[Keller93] Ralf Keller, Wolfgang Effelsberg, Bernd Lamparter, *Performance Bottlenecks in Digital Movie Systems*,
`mail:keller.ip4.informatik.uni-mannheim.de`

[Lamparter91] Bernd Lamparter, Wolfgang Effelsberg, *X-MOVIE : Transmission and Presentation of Digital Movies under X*, Network and Operating System Support for Digital Audio and Video, Springer Verlag, 1991.

[Liou91] Ming Liou, *Overview of the px64 kbit/s Video Coding Standard*, Communications of the ACM, Vol.4, No.4, pp.60-63, April 1991.

[MIT-SHM] Jonathan Corbet, *MIT-SHM - The MIT Shared Memory Extension*, MIT X Consortium, anonymous ftp://ftp.x.org/pub/R6, 1991.

[Oreilly90] *X Toolkit Intrinsics Reference Manual*, Second Edition, O'Reilly & Associates Inc, 1990.

[OSFPROG91] *OSF/Motif Programmer's guide*, Release 1.1, Prentice Hall, 1991.

[OSFREF91] *OSF/Motif Reference manual*, Prentice Hall, 1991.

[Paradise94] *Uniflix 3.2: Installation and User's Guide*, Paradise Software Inc, May 1994,
`FTP: ftp.paradise.com /pub/uflix32xv23/uflix32.tar.Z`

[Parallax91] Parallax Graphics Inc., *XVideo Software Developers Guide*, Manual for the Parallax software package, 1991.

[Pennebaker93] William B. Pennebaker, Joan L. Mitchell, *JPEG Still Image Data Compression Standard*, ISBN 0-442-01272-1, Van Nostrand Reinhold, 1993.

[Rowe92] Lawrence A. Rowe, Brian Smith, Ketan Patel, Steve Yen, *A Continous Media Player*, Network and Operating System Support for Digital Audio and Video, Springer Verlag, 1992.

[Rowe93] L.A.Rowe, K.Patel, B.C.Smith, *Performance of a Software MPEG Video Decoder*, Proceedings of ACM Multimedia 93, Anaheim, California, August 1993.

[Rowe94a] L.A.Rowe, *Video Compression - What to do when everything is changing ?*, Invited talk Usenix 1994, Feb.1994,
`FTP: mm-ftp.cs.berkeley.edu:`
`/pub/multimedia/papers/Usenix94Talk.ps.Z`

[Rowe94b] L.A.Rowe, K.Patel, B.C.Smith, K.Liu, *MPEG video in software: Representation, Transmission and Playback*, Proceedings of SPIE 1994 International Symposium on Electronic Imaging: Science and Technology, San Jose, CA, February 1994.
`FTP: mm-ftp.cs.berkeley.edu:`
`/pub/multimedia/papers/CMMPEG-SPIE94.ps.Z`

[RTP94] Schulzrinne, Casner, Fredrick, Jacobson, *RTP: A Transport Protocol for Real-Time Applications*, INTERNET DRAFT, Work in progress, anonymous ftp://ds.internic.net/internet-draft/draft-ietf-avt-rtp-05.ps.

[Schumacher91] Dale Schumacher, *Graphics Gems II*, Academic Press, 1991.

[Shepherd91] Doug Shepherd, David Hutchison, Fransisco Garcia, Geoff Coulson, *Protocol Support for Distributed Multimedia Applications*, Network and Operating System Support for Digital Audio and Video, Heidelberg Germany, 1991.

[Skeide93] Skeide Sigurd *High Quality Video Server*, Master Thesis for the Computer Science Institute at the Norwegian Institute of Technology (NTH), vol. 1 (1993)

[ST-II94] L.Delgrossi, *Internet STream Protocol Version 2 (ST2)*, IETF ST Working Group, Internet Draft, October 1994.

[Szabo91] Szabo Bernard, Wallace Gregory, *Design considerations for JPEG Video and Synchronized Audio in a Unix Workstation Environment*, USENIX, Nashville TN, 1991.

[Vetterli88] M. Vetterli, *Trade-offs in the Computation of Mono- and Multi-dimensional DCT's*, Columbia University, Center for Telecom Res., Tech. Rep., jun. 1988.

[VoDoo94] Gisle Aas, Arlid E. Bekkadal, *Video on Demand - Object Orientated (VoDoo)*, Project at The Norwegian Computing Center (NR), WWW URL: http://www.nr.no/vodoo/, 1994.

[Wallace91] Gregory K. Wallace, *The JPEG Still Picture Compression Standard*, Communications of the ACM, (vol. 34 no. 4), pp. 30-44, April 1991.

[Walmsley94] N.P.Walmsley, A.N.Skodras, K.M.Curtis, *A Fast Picture Compression Technique*, IEEE Transactions on Consumer Electronics, Vol.40, No.1, February 1994.

[Washburn93] K. Washburn, J.T. Evans, *TCP/IP - Running a Successful Network*, Addison-Wesley, 1993.

[Wolfinger91] Bernd Wolfinger, Mark Moran, *A Continuous Media Data Transport Service and Protocol for Real-Time Communication in High Speed Networks*, Network and Operating System Support for Digital Audio and Video, Heidelberg Germany, 1991.

[Young90] Young Douglas, *X Windows System, Programming and applications with Xt*, 1990.

[Zhang92] Hui Zhang, Tom Fisher, *Preliminary Measurement of the RMTP/RTIP*, Network and Operating System Support for Digital Audio and Video, La Jolla, California, USA, 1992.