

Integration of Digital Video into Distributed Hypermedia Systems

Diplomarbeit in Telematik

Bernhard Marschall

März 1995

Begutachter: o.Univ.-Prof. Dr. phil. Dr. h.c. Hermann Maurer

Betreuer: Dipl. Ing. Dr. techn. Frank Kappe

Technische Universität Graz

Institut für Informationsverarbeitung und Computergestützte neue
Medien (IICM)

Abstract

This thesis discusses digital video and its integration into hypermedia systems. It shows some algorithms and formats to compress and store digital video, focusing on the MPEG standard.

General topics on hypertext and hypermedia systems are discussed as well as their problems and drawbacks, and ways to overcome them. As a special example the architecture and features of Hyper-G and its Unix/X11 client Harmony are described.

Finally, the features and implementation of the Harmony Film Player are described. It fully integrates MPEG movies into the hyperlink structure of Hyper-G/Harmony by allowing links from and to movie documents. Any part, both temporal and spatial, can be used either as source or destination anchor of a hyperlink.

Contents

1. Introduction.....	6
2. Colour Spaces & Compression.....	9
2.1. Colour Spaces	9
2.1.1. The RGB Colour Space.....	9
2.1.2. The YUV (YCbCr) Colour Space	10
2.2. Compression.....	11
3. Digital Video	13
3.1. Motion-JPEG	13
3.2. MPEG.....	15
3.2.1. MPEG System.....	16
3.2.2. MPEG Video	16
3.2.3. MPEG Audio	20
3.2.4. Extensions of MPEG (MPEG-2, MPEG-3, MPEG-4).....	20
3.3. H.261	21
3.4. Quicktime.....	22
3.5. Video for Windows (RIFF / AVI).....	23
4. Hypertext and Hypermedia.....	25
4.1. What is Hypertext?	25
4.2. History & Applications of Hypertext/Hypermedia	26
4.3. Architecture of Hypertext/Hypermedia Systems.....	27
4.3.1. Structure of Hypertext Systems	27
4.3.2. Documents	27
4.3.3. Links.....	28
4.4. Navigation and Information Retrieval.....	28
4.5. Digital Video in Hypermedia Systems	29
5. Hyper-G	31
5.1. Navigation and Information Retrieval in Hyper-G	31
5.2. Architecture.....	32
5.3. Interoperability	34

6. Harmony	35
6.1. Architecture.....	35
6.2. A Tour through Harmony	35
7. The Harmony Film Player	45
7.1. Using the Film Player.....	45
7.1.1. The Control Elements.....	45
7.1.2. The Menu and Button Bars.....	47
7.1.3. The Progress Indicator	49
7.1.4. Dialogs.....	49
7.1.5. Selecting and Following Anchors.....	50
7.1.6. Defining and Deleting Anchors	51
7.1.7. The Film Player's X-Resources	54
7.2. Implementation Details	55
7.2.1. The MPEG Decoder.....	55
7.2.2. Encapsulating the MPEG decoder	58
7.2.3. The Harmony Communication Protocol.....	59
7.2.4. Implementing the Communication - The Dispatcher	62
7.2.5. The Frame List.....	63
7.2.6. Anchors.....	64
7.2.7. The User Interface - InterViews	69
7.2.8. Embedding the MPEG decoder into InterViews.....	70
7.2.9. The Viewer Class	71
8. Summary.....	75
Appendix A: The Harmony Colormap	77
References.....	79

1. Introduction

Information and its exchange has become an important part of our society. Communication networks arise everywhere around the world, most known and most widespread the Internet. The number of participants of the Internet has reached 20 - 30 million and keeps rising, and now covers private persons as well as universities and companies.

But with the enormous success and growth of the Internet, also its disadvantages have become evident: on the one hand, the individual is overwhelmed by the amount of data in the net, on the other hand, one can hardly keep track of the information provided; in short, it is nearly impossible to find exactly the relevant information.

To overcome these problems, distributed information systems have been developed, namely the *Wide Area Information System (WAIS)* [Stein91], which offers a set of search facilities over the WAIS servers and *Gopher* [Alberti92], which structures the information in a hierarchical way, very similar to the way files are structured in a file system of a modern operating system.

A third system has become very popular recently, the *World Wide Web (WWW, W3)* [Berners92], which was developed at CERN in Geneva, where it was used among physicists to store and distribute their data. In the meantime, almost all institutions, research institutes, universities, as well as trading companies, restaurants, and others have their own WWW-servers to present their work or to offer their services. WWW is essentially a hypertext system, which means, text documents are connected by links (which are similar to cross-references in traditional books). External viewers are used to display other kind of documents, such as images, digital videos or audio.

Nevertheless, the existing systems still have their drawbacks: WAIS allows to search the database, but does not try to structure the data, Gopher on the other hand provides no real searching mechanism, and WWW suffers from the typical "lost-in-hyperspace" syndrome of hypertext: users do not know, how much information exists to a certain topic, how much they have already seen and where in information space they currently are.

So a very ambitious project was started at the Institute for Information Processing and Computer-supported New Media (IICM) of the Graz University of Technology (TUG): *Hyper-G*, the first "second generation" hypermedia system [Andrews94b]. It takes the hypertext structure of WWW, but expands it in three essential points: first, it structures the data in a gopher-like way in so called collections, second, it provides a universal search mechanism on the whole database, and third, it is not restricted to texts, but fully supports other types of documents, such as images, films, audio, and others, as well as links between any type of documents.

Although Gopher and WWW clients can be used to retrieve data from a Hyper-G server, only native Hyper-G clients make use of the full functionality of the server. The most advanced Hyper-G client up to now is *Harmony*, which runs on Unix machines under the X-Window

system. For each type of document (text, image, film, audio, postscript, ...) there is a separate viewer, which visualises these kind of documents.

One of these multimedia data types is digital video. Digital video becomes more and more important, as new ideas and concepts for television arise. These concepts are also strongly supported by the entertainment industry and aim towards a new, "individual" kind of television. Most known and discussed are:

- **Video on Demand (VoD):** a service provider sets up a video server; that is essentially a large database storing motion pictures in digital form. The television viewer can select any of these movies, which is then transmitted over some communication network and shown on the customer's television set.
- **Interactive Television:** Interactive television goes one step further: the television viewer doesn't have to watch a movie passively, but can effect the flow of action. To implement interactive television, existing prototypes store various parts of a movie, which correspond to different flows of action. The viewer can choose among them by some interaction mechanism.
- **Three-Dimensional Movies:** In a 3D-Movie the viewer can move through a scene, look into arbitrary directions, or zoom to certain points, thus pushing television towards virtual reality. Apple is going to release Quicktime VR (for Virtual Reality) this year, which will implement 3D-Movies as an extension to Quicktime (Apple's format for digital video).

When one wants to store or transmit digital video, **compression** becomes inevitable. A full colour picture requires 3 bytes per pixel; the PAL standard in Europe requires 25 frames per second, the NTSC standard in the USA requires 30 frames per second, yielding to a total data volume of 75-90 bytes per second and pixel. A television frame has about 352x288 pixels, which gives a total data rate of about 7.25 MByte per second. Different image and video compression algorithms and standards have been developed, among them MPEG (for Moving Pictures Expert Group), an international ISO-standard for compressing and storing digital video with associated audio. Although image (and more video) compression is of a high computational complexity, it becomes more and more used even with PC's, as hardware implementations of both the compression and the decompression algorithm become available.

Digital video is not only important as a replacement or further development for television, but also an interesting feature for new information systems. A known proverb says "A picture's worth more than thousand words". Of course, this holds even stronger for moving pictures. For example, short film clips can effectively be used to illustrate technical processes or present "virtual walks" through scenes. What makes them even more useful is their full integration into a hypermedia system's link structure, making parts of a film a source or destination of "cross-references".

This thesis discusses how digital video has been integrated into Hyper-G and its client Harmony: First, some general topics about compression and image and video compression in particular are discussed. Then the most common formats to store digital video, Motion JPEG, MPEG, H.261, Quicktime and AVI, are described, with the emphasis on MPEG.

After that hypertext and hypermedia, their history, areas of application as well as their problems are presented. Hyper-G as one existing hypermedia system, its architecture and features are described. Harmony, the Unix/X11 client for Hyper-G is discussed as well as the

main theme of this thesis: the Harmony Film Player, which integrates MPEG movies into the Hyper-G/Harmony environment.

When integrating a new type of document into a hypermedia system the most important question is how to integrate it into the hyperlink structure. In theory a link could be attached to any spatial and temporal part of the movie (called its source anchor). Real implementations have to be more restrictive: The Harmony Film Player allows users to define shapes (i.e. rectangles, circles, ...) in so called key frames. The shape in frames between two key frames is interpolated (either linear or as a quadratic B-spline). The first and the last key frame also define the temporal region of the anchor.

2. Colour Spaces & Compression

2.1. Colour Spaces

Colour is a subjective impression of individuals. A colour space is an abstract model, how colour can be "measured". Such an objective description of colour is the base to store and exchange colour data. There exist a lot of different colour spaces [Hill90, Color94]; each provides a somehow different view of colour. In the context of image compression and digital video, the most important colour spaces are the RGB- and YUV-colour spaces.

2.1.1. The RGB Colour Space

The RGB colour space is an additive colour space, i.e. each colour is described as a linear combination of the three primary colours red, green and blue (hence its name). Each colour is therefore given by a three-dimensional vector (R,G,B) (where $R, G, B \in [0;1]$). The set of all describable colours (the colour space) forms the unit-cube; this RGB-cube is shown in figure 2.1.

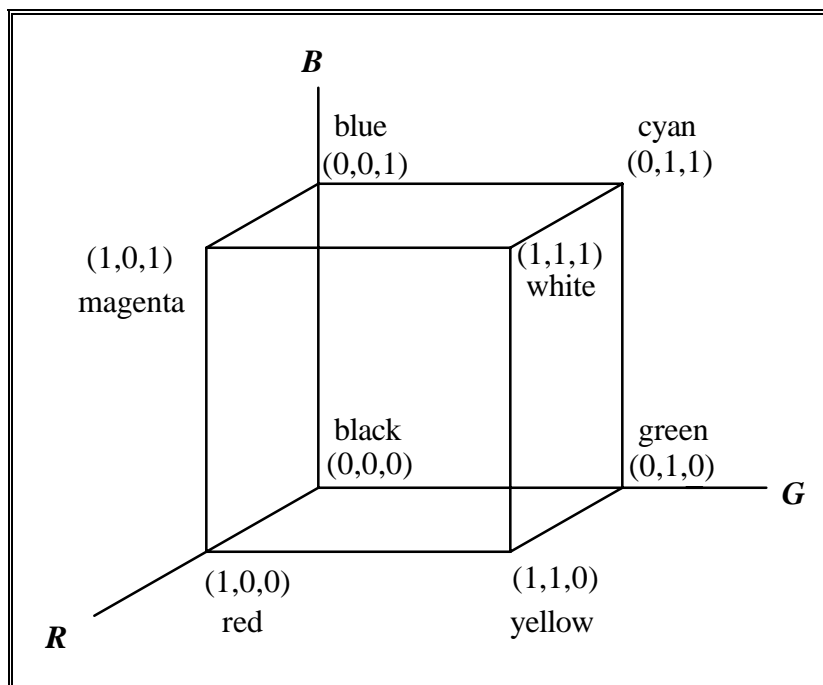


Figure 2.1: The RGB Colour Space

Almost all colour screens today are raster displays, which use a red, a green and a blue glowing phosphor, i.e. the RGB colour space, to display colours.

Normally, computers use one byte to store each colour component; the possible values for R, G, B are therefore not [0;1] but {0, 1, 2, ..., 255}. That gives a total number of $256^3 = 16.777.216$ colours, which can be displayed on a colour screen.

If all these 16 million colours can be displayed on the screen simultaneously, the screen is called a **True-Colour-Device**. True colour devices must store 3 bytes per pixel in their frame buffer.

Other screens cannot display all these 16 millions colours, but only a smaller number - for instance 256 - of them simultaneously. They are called **Pseudo-Colour-Devices**. Pseudo colour devices use one or more colour maps, where the actual colour (the three bytes describing the red, green and blue component) are stored; the frame buffer contains only an index to this colour map.

2.1.2. The YUV (YCbCr) Colour Space

This colour space describes a colour by its luminance Y (the luminance signal) and two colour difference signals, called U and V (the chrominance signals). It is used by PAL, the standard for analogue transmission of colour TV in Western Europe.

The conversion from the RGB colour space to the YUV colour space is as follows:

$$\begin{aligned}Y &= 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\U &= B - Y \\V &= R - Y\end{aligned}$$

Note that U and V can be negative. Therefore a variant of the YUV colour space, the **YCbCr colour space**, is used in digital systems: C_b and C_r correspond to U and V respectively, but they are scaled and translated so that their values are in [0;1], as are the values of R, G, B and Y:

$$\begin{aligned}C_b &= U/1.402 + 0.5 \\C_r &= V/1.772 + 0.5\end{aligned}$$

So we get the conversion from RGB to YCbCr colour space:

$$\begin{aligned}Y &= 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\C_b &= (B - Y) / 1.402 + 0.5 \\C_r &= (R - Y) / 1.772 + 0.5\end{aligned}$$

and back from YCbCr to RGB:

$$R = Y + 1.402 \cdot (C_r - 0.5)$$

$$G = Y - 0.34414 \cdot (C_b - 0.5) - 0.71414 \cdot (C_r - 0.5)$$

$$B = Y + 1.772 \cdot (C_b - 0.5)$$

The advantage of this colour space lies in the fact that the luminance and the chrominance signal, which are perceived differently by the human eye, can be treated differently by an encoding algorithm.

2.2. Compression

Compression algorithms in general, and image compression algorithms in particular, can be divided into lossless and lossy algorithms.

Lossless algorithms loose no information during the encoding process; that means, if an image is compressed and decompressed, the reconstructed image is exactly the same as the original image. Examples for lossless algorithms are huffman coding, runlength coding or the LZW-algorithm [Welch84]. These algorithms reach compression ratios of about 2:1 - in general their compression ratio depends on the input data.

Lossy algorithms, on the other hand, loose information during encoding; that means that the reconstructed image is not exactly the same as the original image. The art of lossy compression is to discard that information, which the human eye can not perceive. Lossy algorithms reach much higher compression ratios than lossless ones, up to 100:1. Examples of lossy compression algorithms are the Color Cell Compression (CCC) [Pins91], JPEG [ISO10918, Wallace91] or MPEG [ISO11172, LeGall91].

Compression for digital video can further be divided into intra- and interframe compression. In **intraframe** compression each picture is encoded independently of any other, just as it were a still image. **Interframe** compression uses the dependencies between pictures to gain higher compression ratios.

The aim of image compression, in general, is to reduce redundancy in the input data; images (as well as digital video) essentially contain three types of redundancy [Gonzales92]:

- **Coding Redundancy** is redundancy due to non-uniform probability distribution of the code symbols. It can be reduced by runlength or huffman coding.
- **Interpixel Redundancy** is due to dependencies, both spatial and temporal, of neighbouring pixels. It can be reduced using difference coding techniques or transformations to frequency domain (discrete Fourier transform or discrete cosine transform).
- **Psychovisual Redundancy** is due to information, which is not perceived by the human seeing. Algorithms that reduce psychovisual redundancy are always lossy, and try to discard non-significant information.

Fidelity Criteria

When using lossy compression algorithms, fidelity criteria get important. After all one doesn't want to discard too much or too important information. There are two kinds of fidelity criteria:

Objective fidelity criteria are the mean square error (MSE) and the signal-to-noise ratio (SNR). Both are measures for the difference of the original and the reconstructed image:

$$\text{MSE} = \frac{1}{M \cdot N} \cdot \sum_{i=1}^N \sum_{j=1}^M (O_{i,j} - R_{i,j})^2$$
$$\text{SNR} = \frac{\sum_{i=1}^N \sum_{j=1}^M O_{i,j}^2}{\sum_{i=1}^N \sum_{j=1}^M (O_{i,j} - R_{i,j})^2}$$

where $O_{i,j}$ are the brightness values of the original image and $R_{i,j}$ are the brightness values of the reconstructed image.

Often objective fidelity criteria are not satisfactory, because they just give the numerical difference of the two images, but not how strong this difference is perceived by a human spectator. This is where **subjective fidelity criteria** cut in: here the quality of the reconstructed image is judged by a group of humans.

3. Digital Video

3.1. Motion-JPEG

JPEG (for Joint Photographic Experts Group) is an ISO standard for compression and coding of still images [ISO10918, Wallace91]. Motion-JPEG, or short MJPEG (which must not be confused with MPEG), is just JPEG used for digital video: that means each frame is intracoded, independently of other frames, using JPEG. JPEG defines a number of compression methods; the most important of them (and what is normally meant when talking about JPEG) is the baseline sequential codec.

A single component (i.e. grayscale) image is encoded in three steps:

First the image is split into blocks of 8x8 pixels. The brightness values are shifted from range $[0; 2^P-1]$ to $[-2^{P-1}; 2^{P-1}-1]$. Each block is then transformed using the forward discrete cosine transform (FDCT):

$$F(u, v) = \frac{1}{4} \cdot C(u) \cdot C(v) \cdot \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \cos \frac{2(x+1)\pi \cdot u}{16} \cos \frac{2(y+1)\pi \cdot v}{16}$$
$$\text{where: } C(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u = 0 \\ 1 & \text{if } u \neq 0 \end{cases}$$

where $f(x,y)$ is the brightness value of the pixel (x, y) of the block; the $F(u,v)$'s are called DCT-coefficients. For decoding the inverse DCT (IDCT) is used:

$$f(x, y) = \frac{1}{4} \left[\sum_{u=0}^7 \sum_{v=0}^7 C(u) \cdot C(v) \cdot F(u, v) \cdot \cos \frac{(2x+1)\pi \cdot u}{16} \cdot \cos \frac{(2y+1)\pi \cdot v}{16} \right]$$

The DCT is similar to the discrete fourier transform, which implies that the DCT-coefficients represent a two-dimensional frequency spectrum. Since $F(0,0)$ is proportional to

the average brightness of the block $\left(F(0,0) = \frac{1}{8} \cdot \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \right)$, it is (in analogy to signal theory) called *DC-coefficient*; the other 63 coefficients are called *AC-coefficients*.

In the second step the DCT-coefficients are quantized using a quantization matrix $Q(u,v)$ with 8x8 elements. What you get are the quantized DCT-coefficients $F^Q(u,v)$:

$$F^Q(u,v) = \text{IntegerRound}\left(\frac{F(u,v)}{Q(u,v)}\right)$$

The dequantization, needed for decoding, is the inverse function:

$$F^*(u,v) = F^Q(u,v) \cdot Q(u,v)$$

Note that this is the only step in the whole process where information is lost, since $F^*(u,v) \neq F(u,v)$ due to the round function. The aim of quantization is to discard information, which is not visible to the human eye. (For instance the human eye is much more sensible to the average brightness of a block (i.e. the DC-coefficient) than to very high frequencies.)

In the third step these quantized DCT-coefficients are entropy coded: At first the coefficients are arranged in zig-zag order (see figure 3.1).

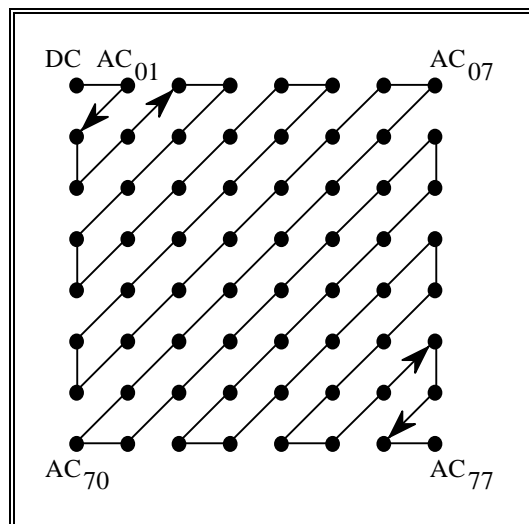


Figure 3.1: Zig-zag sequence

The DC-coefficients are differentially encoded; that means, not the current value of the DC-coefficient is stored but only the difference of the current DC-coefficient and the one of the previous block. Because normally the average brightness of consecutive blocks does not vary too much, the value of the difference is rather small and can well be compressed using huffman coding.

The sequence of AC-coefficients is runlength encoded as a pair of symbols $s1 = (runlength, size)$ and $s2 = (amplitude)$: *runlength* is the number of consecutive zero-valued AC-coefficients, and *size* is the number of bits used to encode *amplitude*, which is simply the value of a nonzero AC-coefficient. These symbols are finally huffman encoded using a table according to a probability distribution of the symbols.

JPEG handles colour images as so called multiple component images, where each primary of the colour space represents one component. The components need not have the same spatial resolution. Normally colour images are encoded using the YC_bC_r colour space,

were C_b and C_r are subsampled. Different quantization tables may be used for different components.

Figure 3.2 summarises the JPEG encoding process:

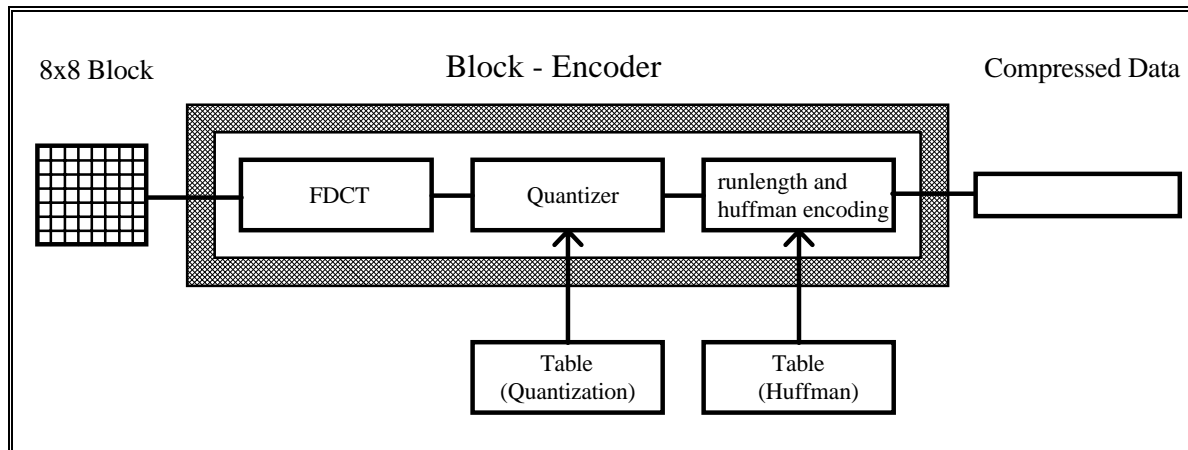


Figure 3.2: JPEG encoding process

MJPEG reaches compression ratios of 10:1 to 20:1 (i.e. 1-2 bits/pixel) without visible effects, 30:1 to 50:1 (i.e. 0.5-1 bits/pixel) with only moderate defects, and up to 100:1 (i.e. 0.25 bits/pixel) for low-quality purposes. This compression ratio is certainly not optimal, since the redundancy between two consecutive frames is not exploited (as is done in MPEG). Nevertheless MJPEG provides some advantages over MPEG:

- The JPEG committee started earlier than the MPEG committee, so the standardisation is further developed and hardware components supporting JPEG compression are well established.
- An MJPEG video can be more easily processed than a MPEG video since there are no interframe dependencies.

3.2. MPEG

MPEG (for Motion Pictures Expert Group) is also an international ISO-standard [ISO11172, LeGall91] which describes how to encode digital video with associated audio. The standard defines the syntax and semantics of the MPEG bitstream and hence the decoder. The encoder is not specified directly; so it is possible to adapt the encoder according to the needs of the application (real time compression vs. high compression ratio vs. high quality).

The bitstream is compressed to about 1.5 Mbit/s, which is also the bit rate of the (uncompressed) audio CD. The standard consists of 3 parts: part 1 describes the system coding layer, part 2 the encoding of digital video and part 3 the encoding of digital audio.

3.2.1. MPEG System

This first part of the standard describes, how video, audio, and private data streams are combined, synchronised and multiplexed.

An MPEG system data stream may contain up to 32 audio, 16 video, and 2 private data streams. An MPEG system stream is divided into packs. A *pack* may either contain system data (a system header) or one or more packets. Each *packet* contains an identifier, which gives its type and number (video 0-15, audio 0-31, or private), as well as a presentation time stamp, so that it can be displayed (or played) at the correct time.

3.2.2. MPEG Video

The encoding scheme for MPEG video is based on JPEG. The input data are the frames of the digital video in the YC_bC_r colour space. As shown in figure 3.3, the chrominance signals are subsampled: whereas the luminance value (Y) of every pixel is stored, the chrominance values (C_b, C_r) of 4 pixels are combined. This can be done without visible loss of quality because the human eye is much more sensible to luminance than to chrominance and it already halves the amount of data to encode.

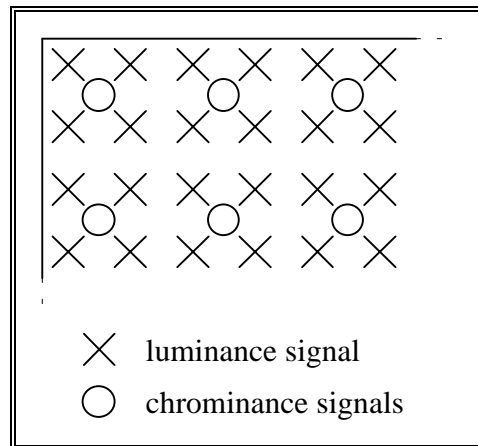


Figure 3.3: Subsampling of the chrominance signal

There are three types of frames: I-frames, P-frames, and B-frames. (There are also D-frames, which encode only the luminance signal in low quality, but they must not be mixed with other frame types. Video streams with D-frames are only used within system streams, and their only intention is to allow a fast forward search mode). Each frame is split into macroblocks of 16x16 pixel size; each macroblock contains 6 blocks of 8x8 brightness values (4 luminance blocks and 2 chrominance blocks).

I-Frames

I-frames (for *intracoded frames*) are encoded just like still images using JPEG encoding, with some minor changes and simplifications:

The 6 blocks of each macroblock are DCT-transformed, the DCT-coefficients are quantized using a quantization table (in contrast to JPEG there is only one quantization table for both, the luminance and the chrominance signals) and finally the quantized DCT-coefficients are runlength and huffman encoded.

I-frames provide a starting point for decoding, since they do not depend on any other frame.

P-Frames

P-frames (for *predictive coded frames*) depend on the previous I- or P-frame (called reference frame). Motion compensation is used to "predict" frames: for each macroblock, you search for the most similar region of 16x16 pixel in the reference frame (this so-called reference block need not be a macroblock of the reference frame, but can be any region of 16x16 pixels). The search can be done in one- or half-pixel resolution; the search algorithm itself is left to the encoder (the standard just suggests some reasonable algorithms).

What is really encoded is the prediction error: that is the difference between the current macroblock and the reference block. The same procedure as in I-frames is used to encode this error block: the prediction errors are DCT-transformed, quantized (with an other quantization matrix than the DCT coefficients of I-frames) and finally runlength and huffman encoded. Further you have to encode the motion vector (i.e. the number of pixels or half pixels in both directions by which the position of the current macroblock has to be translated to get the position of the reference block). These motion vectors are also differential encoded: you encode the difference between two consecutive motion vectors, not the motion vector itself. This is reasonable, because normally two consecutive blocks will be translated by approximately the same amount, so the differential motion vectors tend to be rather small and can be encoded efficiently using huffman coding.

If both the differential motion vector and the prediction errors are zero, the macroblock is encoded as *skipped macroblock*. For skipped macroblocks no actual data are stored (only the macroblock header, identifying the macroblock as skipped macroblock), so they compress extremely well.

If the encoder cannot find a good reference block, the current macroblock is encoded as in I-frames. So P-frames can contain both I-blocks and P-blocks, whereas I-frames contain only I-blocks. It's entirely left to the encoder, which type of block to use.

B-Frames

B-frames (for *bidirectional predictive coded frames*) are an extension of P-frames. Motion compensation is not only done for the previous (past) I- or P-frame but also for the following (future) one and the average of any past and any future block (it's up to the encoder which combinations it's taking into consideration).

So you have to encode a forward and a backward motion vector as well as the prediction errors. As in P-frames, if both motion vectors and the prediction errors are zero the block can be encoded as skipped macroblock.

Since there are now a lot more choices for the reference block, it is more likely that a really good one is found. Future prediction furthermore makes it possible that a good reference block can be found for previously hidden objects.

As in P-frames, if no good reference frame is found, the macroblock is intracoded.

Summary of Compressing Techniques used in MPEG

MPEG combines a number of compression techniques, which are here summarised:

- *Subsampling of the chrominance signal:* For 4 luminance values there is only one value of each chrominance signal; this can be done without visible effects, because the human eye is much more sensible to luminance than to chrominance.
- *Quantization:* A whole range of values is represented by a single value in that range. Since the range of possible values decreases, the number of bits needed to encode a value decreases. Quantization causes loss of information, which is called quantization noise.
- *Predictive Coding:* One tries to estimate the value of a pixel by values of previously encoded pixels; so only the difference between the estimation and the current value (the prediction error) has to be encoded. These prediction errors are normally small and can be encoded more efficiently. MPEG uses predictive coding to encode the DC-coefficients and the motion vectors.
- *Motion Compensation and Interframe Coding:* One tries to predict a block by a block of a known (reference-) frame. This technique relies on the fact, that within a short sequence frames don't change very much.
- *Frequency Transformation (DCT):* Brightness values are transformed to frequency coefficients. In general the energy is concentrated in the lower frequencies, and most of the higher frequencies are zero.
- *Runlength coding:* If there is a series of equal values (i.e. zeros), one doesn't encode them all, but only their number and once the value.
- *Huffman Encoding (variable length coding):* Highly probable values are encoded using only few bits, whereas improbable values get longer codes.
- *Picture Interpolation:* This is used in B-frames. One does motion compensation also with the average of a past and a future block, i.e. blocks that do not really exist in any frame of the sequence, but which are likely to be similar to blocks of the B-frame between.

Sequence of frames

The choice of the type of the frames used and their sequence, is entirely left to the encoder. An often used sequence is IBBPBBPBBPBB.... This sequence and the dependencies of the frames are shown in figure 3.4.

This sequence guarantees that there is an I-frame every 0.4 seconds (for the American TV standard NTSC with 30 frames per second) or every 0.48 seconds (for European TV standards PAL and SECAM with 25 frames per second) where decoding can start.

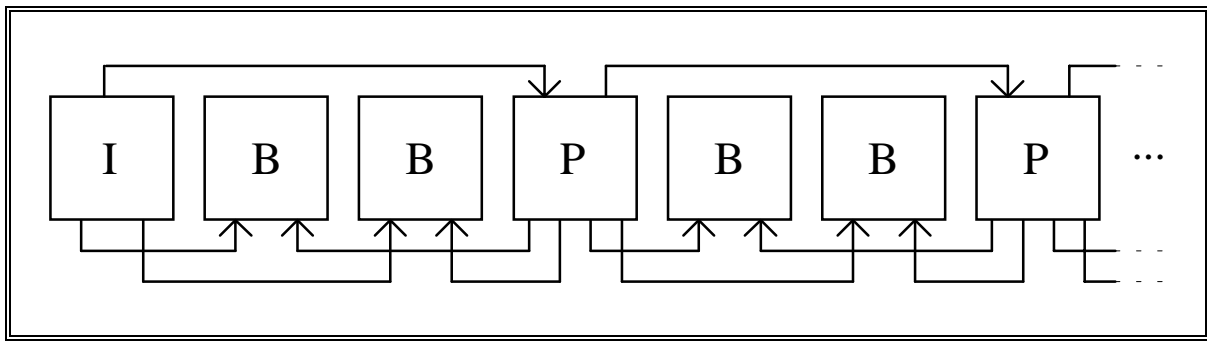


Figure 3.4: Dependencies of Frames

The motion prediction to future frames in B-frames rises one problem: to be able to decode a B-frame, you have to decode the future I- or P-frame in advance; therefore it has to be stored in front of the B-frame. So one has to distinguish between the display order and the file order of the frames.

The *display order* gives the order in which frames have to be displayed, for example: $I_1 B_1 B_2 P_1 B_3 B_4 P_2 B_5 B_6 P_3 B_7 B_8 I_2 \dots$

The *file order* gives the order in which the frames are encoded in the MPEG stream, in our example: $I_1 P_1 B_1 B_2 P_2 B_3 B_4 P_3 B_5 B_6 I_2 B_7 B_8 \dots$

Bitstream Hierarchy

The MPEG stream is composed of six layers:

Sequence Layer
Group of Pictures Layer
Picture Layer
Slice Layer
Macroblock Layer
Block Layer

Figure 3.5: MPEG bitstream hierarchy

The *Sequence Layer* is the top layer: it starts with a sequence-start-code, where various parameters are set, followed by one or more groups of pictures and finally a sequence-end-code.

A *Group of Pictures (GoP)* consists of one or more pictures: it starts with a group-start-code and ends with the next group-start-code (or the sequence-end-code). A GoP starts with an I-frame in file order and ends with an I- or P-frame in display order. GoPs are independent of each other (except for B-frames previous to the first I-frame in display order); their aim is to allow random access to the stream.

A *Picture* corresponds to a single frame in the video. Each picture starts with a picture-start-code, where among other parameters its frame type is stored (I-, P- or B-frame).

Each picture consists of *Slices*. Slices provide some immunity to corrupt data (for example due to transmission errors). The slice-start-code provides a new starting point to continue decoding; so not the whole corrupted picture has to be discarded. The number of slices per picture can be chosen by the encoder according to the quality of the line of transmission. If the line is reliable (or the video is just stored as a file on disk), the slice-start-codes are an additional overhead, which decreases both, the compression ratio and the speed of decoding.

A *Macroblock* consists of 16x16 pixels of a frame and is the coding unit for motion compensation. A macroblock consists of six blocks.

Finally, *Blocks* consist of 8x8 brightness values (either for the luminance or a chrominance signal) and are the coding unit for the DCT.

3.2.3. MPEG Audio

The encoding of video and audio differ, because the eye and the ear work very differently. MPEG audio defines a family of three audio coding schemes, called Layer-1,-2,-3. The decoders are hierarchical, i.e. a Layer-n decoder can also decode bitstreams encoded in all layers below n.

The basic encoding scheme is the same for each layer: the input signal is a digitised audio signal in Audio-CD quality (i.e. 44.1 kHz sampling frequency and 16 bit resolution to encode the amplitude). The input signal is divided into frequency subbands; for each subband a just noticeable noise-level is estimated using a psychoacoustic model. This noise level determines the number of bits necessary to encode a frequency.

The *psychoacoustic model* uses a masking effect of the human ear: If there is a strong tone with a certain frequency, for example at 1000 Hz, and a lower tone nearby, for example at 1100 Hz, this second tone is masked and cannot be heard if it stays below a certain level. This masking effect means that you can raise the noise level around a strong frequency (because you can't hear this noise), and raising the noise level means using less bits to encode the amplitude of the signal near this strong tone. Furthermore this masking effect occurs also 2-5 ms before and up to 100 ms after a strong tone; this is called pre- and postmasking respectively.

For stereo signals the dependency of both channels is further exploited by using a joint-stereo-mode. Layer 3 further reduces redundancy by applying huffman encoding.

3.2.4. Extensions of MPEG (MPEG-2, MPEG-3, MPEG-4)

MPEG-2 [ISO13818] was finished in November 1994. Whereas MPEG-1 was dedicated to applications with data rates of CD-ROMs (1.5 Mbit/s), MPEG-2 tries to catch the requirements for digital television: here quality is more important, whereas the data rate may be higher (about 4 MBit/s). Analogue television is transmitted interlaced (i.e. 2 fields are transmitted: the first field contains only the odd rows, the second contains only the even rows

of a frame). MPEG-2 supports the encoding of interleaved video and exploits the redundancy between the fields and frames.

MPEG-3 was intended for high definition television (HDTV) using data rates of 20-30 MBit/s, but this application is now already covered by MPEG-2.

MPEG-4 is dedicated to applications with very low data rates (around 4.8 - 64 kBit/s) like the video phone. The CCITT H.261 standard (see chapter 3.3. below) is also intended for these applications.

3.3. H.261

H.261 is a recommendation of the ITU/CCITT [ITU90], the international telecommunication unit. The intention of H.261 is to provide a standard for videophone applications over an ISDN network with a bit rate of p times 64 kbit/s.

The source format are non-interlaced pictures in either the CIF format (i.e. 352x288 pixels) or the QCIF format (176x144 pixels) and 30000/1001 (=29.97) pictures per second in the YC_bC_r colour space, where - as in MPEG - C_b and C_r are spatially subsampled.

Error handling is done by a BCH (511,493) forward error correction code. The implementation of this BCH code is optional.

If the encoder is too slow to meet the required picture rate, temporal subsampling is done by discarding complete pictures.

Bitstream Hierarchy

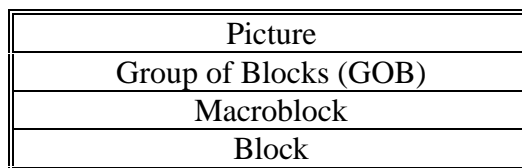


Figure 3.6: H.261 Layers

The top layer is the *picture layer*. A picture starts with a header and is followed by a number of group of blocks.

A *group of blocks (GOB)* covers 176x48 luminance values and 88x24 chrominance values. A GOB also begins with a header and is followed by 33 macroblocks

A *macroblock* covers the area of 16x16 pixels. The header of the macroblock states if the macroblock is intra- or intercoded. Interframe coding is done by motion compensation (and optionally applying an additional filter).

A macroblock - as in MPEG - consists of six *blocks*: four luminance blocks and two chrominance blocks. For intraframe coding the luminance or chrominance values are used respectively, for interframe coding the prediction error values are used. As in MPEG these values are DCT-transformed and the DCT-coefficients are runlength and huffman encoded.

Comparison with MPEG

H.261 is a much simpler than MPEG. It essentially provides equivalents of I- and P-frames, but there is no such thing as B-frames. Therefore its compression ratio is surely lower than the one of MPEG. On the other hand, H.261 was mainly developed for the use in videophone applications where consecutive frames do hardly differ at all (normally speakers just moves their heads and hands a bit) and it is not very disturbing if some frames are discarded from time to time, if the encoder is too slow or has already used too many bits. Moreover H.261 has to work in real time (encoding as well as decoding), so its lesser computational complexity is a big advantage.

3.4. Quicktime

Quicktime [Apple93, Günter92] is the multimedia extension for the Apple Macintosh computer. It is a part of its operating system, and therefore comes free with every Mac. The aim of Quicktime is to provide the integration of general time-based data into the Macintosh hard- and software environment. Therefore it can be best compared with MPEG system.

Quicktime has five main tasks:

- It provides the Movie Toolbox, the Image Compression Manager and the Component Manager, which can be used by applications.
- It standardises the management of input media, like frame-grabbers, by offering a concept similar to printer drivers.
- It provides different compression algorithms.
- It implements a new data type, the movie, which can be cut and pasted like text or images, or integrated into other applications.
- It provides a standardised user interface to record, play, edit, and compress movies.

A *movie* is a container, which contains one or more *tracks* with data of different type and origin, for instance a film and the corresponding audio, both in English and German.

Quicktime is a scheme to store time-based data like digital video and not a compression algorithm. In contrary, different compressors are provided by Quicktime. All compressors support *frame differencing*: that means, not the whole frame is compressed and stored, but only the difference between two succeeding frames. This increases the compression ratio, on the other hand, it is no longer possible to jump directly to a frame, decode and display it. So the algorithm inserts so-called key frames into the data streams. Key frames are just intracoded images (they correspond to MPEG's I-frames). The number of frames per key frame can be set by the user in the standard compression dialog; as a rule of thumb the number of frames between to key frames should be approximately half the frame rate (i.e. when displaying 30 frames per second, after each 15th frame a key frame should be inserted). Furthermore Quicktime automatically inserts a key frame, if a frame differs from its predecessor by more than 90 per cent.

Quicktime supports the following compressors:

- The **Raw Compressor** is not a real compressor; it just stores the frames uncompressed, but it is possible to change (decrease as well as increase) the number of bits per pixel.
- The **Photo- or JPEG Compressor** uses JPEG to compress the frames. Frame differencing is not possible, when using this compressor.
- The **Video Compressor** is a lossy algorithm, which uses spatial and temporal dependencies to compress frames. It was developed by Apple to compress video with 24 bits colour information in real time and play it with at least 10 frames per second. Its compression ratio is between 5:1 to 8:1.
- The **Compact Video Compressor** is a variant of the Video Compressor, which has greater compression ratio, faster playback speed, but slower compression time.
- The **Animation Compressor** is based upon the compressor for still images in Apple's PICT-format. It uses run-length encoding, optionally lossy or lossless. It is best suited for computer generated animations.
- The **Graphics Compressor** is an alternative to the animation compressor, which works for images with 8 bits/pixel and has approximately the double compression ratio and needs the double time to compress a video.
- An **MPEG Compressor** is provided by the new version Quicktime 2.0.

As a special feature, Quicktime allows to mark a frame as **poster frame**. This poster frame should be a frame representative for the whole film. If a Quicktime movie is integrated into another application, this poster frame (along with a movie icon) is shown; if the user clicks onto this frame, the movie starts playing.

3.5. Video for Windows (RIFF / AVI)

Video for Windows [Muray94, Pöpsel94] is the most used system to play digital video on PC's under Microsoft Windows. In fact, Video for Windows is just a subset of Microsoft's RIFF (Resource Interchange File Format) format. RIFF is (as Quicktime) a multimedia file format, which defines a structured framework, which may contain various data types stored in already existing data formats.

The actual data type contained in a RIFF file is indicated by its file extension; there are

- AVI (audio/visual interleaved) for digital video with associated audio
- WAV (waveform data) for audio data
- RDI for bitmaps
- RMI for MIDI data
- BND for a bundle of other RIFF's

Till now, only the AVI and WAV formats are fully specified and implemented.

A RIFF file consists of multiple nested data structures, called *chunks*. For an AVI file, there are three types of chunks:

- The list-chunk contains a header, indicating the format of the data streams.
- The AVI data chunk contains the actual video and audio data, either stored interleaved in subchunks or as a whole block in one chunk.
- The index chunk contains a list of all other chunks and their locations in the file; it is used to allow random access to the video.

4. Hypertext and Hypermedia

4.1. What is Hypertext?

Traditional texts are sequential: normally readers start at page one, then they read page two and so on. Hypertext ([Nielson90, Berk91]) is nonsequential; there is no given sequence in which the text is to be read.

Hypertext consists of pieces of text (called *nodes* or *documents*), which are connected by *links*. Figure 4.1 shows an example of a rather small hypertext consisting of 6 documents and several links.

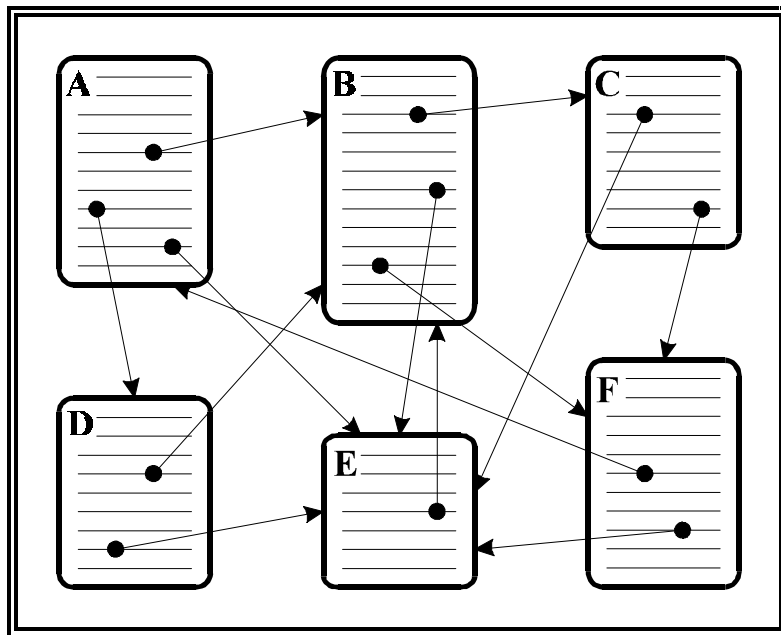


Figure 4.1: An Example of a small Hypertext

Links can be compared with cross-references or footnotes in traditional texts. They make different options to browse through the documents available to the reader. The actual sequence in which the text is read, is determined by the reader at the time of reading. Whereas in traditional texts authors have full control of the sequence of reading, in hypertexts they can only make suggestions (i.e. links) for possible sequences, but it is up to the reader which way to choose.

As can be seen in figure 4.1, hypertext forms a network of documents and links. Reading the hypertext corresponds to moving around this network. To emphasise the reader's

responsibility of choosing which document to read next, reading a hypertext is often called *navigation*.

The start point of a link is called its *source anchor*, its endpoint is called *destination anchor*. The source anchor is that part of the link that is depicted on the screen to call the reader's attention to the link. The destination anchor describes the "target area" of the link; so it is possible that a link refers only to a part of a document, instead of the whole one.

Hypermedia [Maurer92] is the extension of hypertext to multimedia. Documents may not only contain text, but also images, movies, audio, animations and so on. Since the idea of hypermedia systems and hypertext systems is the same, people often use the two words as synonyms. Nevertheless hypermedia introduces some additional problems, for instance the amount of data needed to store raster images, digital video or audio.

4.2. History & Applications of Hypertext/Hypermedia

Memex described by Vannevar Bush in 1945 is the first hypertext system proposed. Bush claimed that the progress of research is slowed down by the inability of researchers to find the relevant information. So he proposed Memex as a device in which people would store their books, records and communications and link them. Since computers filled up whole rooms and were not considered very practical those days, Bush's proposal included a sophisticated mechanism of microfilms and microfilm projection devices; it has never been implemented.

Augument/NLS by Doug Engelbart started in 1962; during the Augument project, the researchers stored all their papers and reports in a shared journal, which included cross-references to other works.

Xanadu was proposed by Ted Nelson in 1965: it was Nelson, who coined the word "hypertext". The idea of Xanadu was to store everything that anybody has ever written (and will ever write). Nelson believes that "everything is deeply interwined" and therefore has to be stored and linked together.

The *Aspen Movie Map* of 1978 was probably the first true hypermedia system. All the streets of the city of Aspen (Colorado) were filmed from a truck driving through the streets. Each photograph was linked to other photographs in that sequence a person would see when walking straight ahead, backing up, or moving to the left or to the right. The user of this map can navigate through this information space using a joystick. Furthermore it is possible to walk into buildings, since many of them were filmed, too.

Hypertext or hypermedia systems may be of use for a variety of applications, especially when the information is organised in numerous fragments, these fragments relate to each other and the user needs only a small fraction of the information at any time.

Examples of already existing hypertext/hypermedia applications are: online documentation, help, and tutorials within software systems, dictionaries and encyclopaedias, especially electronic publishing [Maurer94], and systems used for education, in libraries or museums ("interactive library/museum"). There also exist some works of interactive fiction,

where a story is told not sequentially as in a traditional novel, but as a hypertext, where the reader determines the actual flow of action.

4.3. Architecture of Hypertext/Hypermedia Systems

4.3.1. Structure of Hypertext Systems

One can distinguish three architectural levels of a hypertext or hypermedia system: The *database level* is the bottom layer of the system. It deals with traditional database issues, such as information storage, multi-user access to the information, and security considerations. For the database level documents and links of the hypertext are just ordinary data objects with no particular meaning.

The middle layer is the *hypertext abstract machine (HAM)*. Here the basic nature of the documents and links of the hypertext system is defined: the HAM knows of the form of documents and links and their attributes. The HAM is also a subject for standardisation, which is important if one wants to interchange hypertexts between different systems.

Finally, the top layer is the *presentation level*: here the actual user interface is defined, how documents and links are actually displayed, how commands are issued and what commands exist.

4.3.2. Documents

Documents are the smallest unit of information in a hypertext system. Depending on the actual system they are also called *frames*, *cards* or *nodes*. There are two types of systems:

In *frame based* systems, each document (or frame) has exactly the size of the computer's screen, independently of the information it contains. A page may consist of more than one frame. This implies, that the author has full control of the look of the node. On the other hand nodes are hard to modify in frame based systems. (Consider you want to add one line to a node, but this line doesn't fit on the screen anymore; so you have to create a new frame, which can affect some destination anchors, etc.)

In contrast, *window based* systems require the user to scroll the text, because the document may be too large to fit into the window. In such systems the author has no control over the presentation of the document (it may be in a rather small window as well as in a full-screen window). On the other hand, users can adjust the sizes and positions of the windows according to their needs. Window based systems can also change the metaphor of presentation according to the circumstances, and changes of nodes can be achieved rather simple, since other windows are not affected.

4.3.3. Links

Links are the second fundamental data type in hypermedia systems. Normally, links are anchored: the departure and destination points of the link are called anchors. Anchors are the structures which are displayed on the screen and indicate the existence of the link to the user.

Links can be either *unidirectional* or *bidirectional*. Most systems only support unidirectional links, because they are easier to implement (one just needs to store the destination of the link in the document), whereas bidirectional links require a separate link database. Nevertheless they provide two advantages: First, they make it easier to support navigational facilities such as overview maps (see chapter 4.4.), and second, they allow to check the consistency of the database, if documents are to be modified or deleted.

A seldom used type of links are *multi-ended links*: such a link has not one but many destinations; normally, when the link is activated, a menu is displayed to allow users to choose to which destination they really want to go.

"Normal" links replace documents, that means, the old document, containing the link, is replaced by the new document (this corresponds to the goto-statements of traditional programming languages). A special kind of link is the *annotation*: an annotation typically opens a pop-up window, when it is activated. When users have read the annotation and close the window, they return to the old document (this corresponds to the gosub-statement or function call).

4.4. Navigation and Information Retrieval

Users moving around a large hypertext, visiting document after document, will become disoriented rather quickly. This is known as the famous "*lost in hyperspace*" syndrome. Also, it is difficult for readers to find all the relevant nodes, they are interested in, or to estimate how much of the relevant information they have seen already. Various solutions to overcome these problems exist. Most implemented systems provide a combination of them, rather than just one.

The perhaps most simple solution are *guided tours*: a guided tour connects documents in a predefined way, thus relieving users of navigation. They just have to issue some sort of "next" or "previous" command. Guided tours are well suited to introduce users to a certain subject and to provide a first overview over documents in the database. Of course users may abort the tour at any time and come back to the "guide" later. Nevertheless guided tours bring users back to reading a text sequentially.

Surely, the most important navigational facility is the *backtrack*, which takes users back to the previously visited document. Backtrack allows users to return to "known" territory, no matter how far they have gone. A more general mechanism are *history lists*: Whenever a document is visited, it is inserted into the history list, thus allowing users to jump back to this document later.

Some systems allow users to define a list of *bookmarks*; the difference between this list (also called *hot list*) and the history list is, that users must explicitly add a document to the hot

list, whereas they are inserted automatically into their history list. Frequently used documents can be put to the hot list, so users are able to find them again later quickly. A special kind of bookmark is the possibility to interrupt the current session and resume it later without changing the state of the hypertext.

Overview diagrams are another way to support navigation. Unfortunately, the number of documents and links in a (distributed) hypertext system is too large to show them all on a single map. Therefore various levels of detail are provided: for instance a rough view of the information space is provided and the user can zoom in to get more details. Another possibility is the use of a **local map**: it shows only the surroundings of the current document, i.e. these documents which are connected to the current one by a path of links of a certain maximal length (for example all documents which can be reached from the current one by following at most two links); if links are bidirectional, the local map cannot only be displayed for outgoing links, but also for incoming ones.

Information retrieval is an important topic for all information systems, and therefore also for hypermedia systems. The subject is how to find all relevant information. Traditional information systems allow users to query the database using some sort of query language. In hypermedia systems all information can, of course, be found purely by navigation. But, as mentioned above, it is difficult for users to guess, where this information is located in the information space. Therefore it makes sense, if hypermedia systems also allow search queries as do traditional database systems. The answer to such a query could be a starting point for hyper-navigation. The simplest query, at least in hypertext, is a **full text search**; but it can be impractical in really big, distributed systems, and, when it comes to hypermedia, how do you search in pictures, movies, or audio? A fairly good search strategy is to search for **attributes** of the nodes, such as title, keywords, or author. Since links describe relationships of nodes, they could be used by more sophisticated methods to perform some sort of "semantic search".

4.5. Digital Video in Hypermedia Systems

The first problem arising, when trying to integrate digital video into a hypermedia system, is the enormous amount of data needed to store digital video. So compression becomes inevitable. Compression and formats to store digital video have already been discussed in the chapters 2. and 3.

The next question is, what a link in a film document is, i.e. how its anchors look like. With films, in contrast to texts or still images, the temporal dimension must be taken into account. In general, an anchor can be any region of a picture of the film, which can move and change its size and shape in time from one picture to the next one. In practice, one has to constrain from arbitrary shapes to fixed ones, such as rectangles or circles.

Next, the question, how such an anchor can be defined, must be considered: surely it would not be practical, if the position and size of the anchor region has to be defined for each single picture. One way to overcome this problem is to let the user define the region in some pictures (called **key frames**); in the pictures between key frames the region is calculated by interpolation.

Two kinds of links can be identified, when dealing with time-based data:

- User activated links: These links must be activated by the user explicitly, for example by clicking on its source anchor. They can be compared with links in texts or still images.
- Automatically activated links: These links are bound to a certain point in time, and are activated automatically, when the corresponding picture is displayed. Films containing this type of links are also called *annotated films* [Lennon94], since the links can provide some kind of annotation (or explanation) of the film.

Digital video in hypermedia systems can be used for many applications, for example:

- **Visualisation:** users can look at or walk through real or virtual scenes. Links could provide additional explanations or detailed images of the objects seen in the movie.
- **Multimedia presentation:** An annotated film can also be used for a multimedia presentation. The film provides the timeline, which causes other documents (texts, images, ...) to be displayed.
- **Programmable film:** the programmable film (sometimes also called PILM) is the pendant of a literary hypertext: the film itself consists of a lot of small clips; the user can control the flow of action by following different links, which will select the appropriate clip.

5. Hyper-G

Hyper-G is a large-scale, general-purpose, distributed, multi-user, hypermedia information system, which is currently developed at Graz University of Technology. The aim of the Hyper-G project [Kappe91] is to study and (if possible) eliminate the typical problems of hypermedia mentioned in the previous chapter.

5.1. Navigation and Information Retrieval in Hyper-G

Hyper-G combines a number of navigational and search facilities, in order to overcome the "lost in hyperspace" problem [Andrews93].

Hyper-Navigation

Of course, since Hyper-G is a hypermedia system, the user can navigate using hyperlinks: a link may lead from one part of a document (the source anchor) to a part of another (or the same) document (the destination anchor). Hyper-navigation is somehow the natural way of finding information in a hypermedia system, especially as multimedia documents (i.e. other than text documents), in general, cannot be searched for.

Collection Hierarchy

Every Hyper-G document is a member of one or more collections, which are in turn members of one or more collections, thus creating a hierarchy of collections. Note that this hierarchy is not a tree, but an acyclic directed graph, since objects may be members of more than one collections. This collection hierarchy provides an additional structure over the document space and serves three purposes:

- **Navigation:** The collection hierarchy offers some kind of global map; whenever users visit a document (no matter which navigational paradigm is used), the location of this document in the collection hierarchy is shown. Furthermore, since collections are used to combine "similar" documents (i.e. documents belonging to the same subject), users get an overview of documents belonging to the same or a similar topic rather quickly.
- **Search scope:** Users may mark certain collections as active, before issuing a search query; thus, the number of found documents can be restricted.
- **Access rights:** In a large hypermedia system with thousands of users and thousands of authors it is necessary to provide some kind of access rights. Hyper-G allows to grant or deny read or write access to collections, similar to the read and write access rights in the UNIX file system.

Currently three types of collections are defined:

- Ordinary *collections* as described above: when visited, a list of all items the collection contains is displayed. The order of the items may be defined statically or dynamically sorted by certain attributes of the collection members.
- A *cluster* is a special collection: when it is visited, all its members are visited (i.e. visualised) too. Clusters allow it to combine different types of documents, for instance a film with its textual description. Further, they are used to implement multilingual documents and version control.
- A *tour* is a collection, which visits its members in a certain order. It is used to linearize hypertexts and to implement guided tours by automatically generating a "next" and a "previous" link.

Search Facilities

There are two modes of searching. Every Hyper-G object has a set of associated *attributes*, such as title, keywords, type, author, creation time, expiration time, etc. These attributes can be searched for, including boolean combinations. Typical queries could be: "Search for all text documents having digital video and hypermedia in the title" or "Search for all documents written by Smith this year".

Additionally, texts can be searched for by *full-text* queries, since they are automatically indexed on insertion into the database. The Hyper-G full text server supports both, fuzzy boolean queries and nearest-neighbour searches based on the vector space model.

In both modes the collection hierarchy can be used to define the scope of the search ranging from a single collection on a single Hyper-G server to all collections on all Hyper-G servers world-wide.

Of course, users are not expected to stay with one of the above navigation paradigms. Contrary, they will change it constantly according to their needs. For example, users might use the collection hierarchy to go to a certain encyclopaedia, then activate this collection and issue a search query. When reading the found documents they may jump to other documents using hyper-links ("cross-references"). Therefore it is important that the user interface is kept consistent, even when the navigation paradigm is changed, so that users are not confused.

5.2. Architecture

Hyper-G uses a *client/server model* [Kappe93], with clients and servers connected via the Internet using TCP/IP. Figure 5.1 shows the architecture of Hyper-G.

Note that unlike Gopher or WWW the client has to connect to only one Hyper-G server. If the client needs information from a remote server, the local server fetches it and passes it on to the client. This has some advantages:

- Clients are kept simple, since they need not connect to many servers.

- It enables caching of remote documents in the local server.
- The maintenance of user accounts and access rights is kept simple, since the user has to identify to one server only.
- The link server can gather statistics and user profiles on a per-session basis.

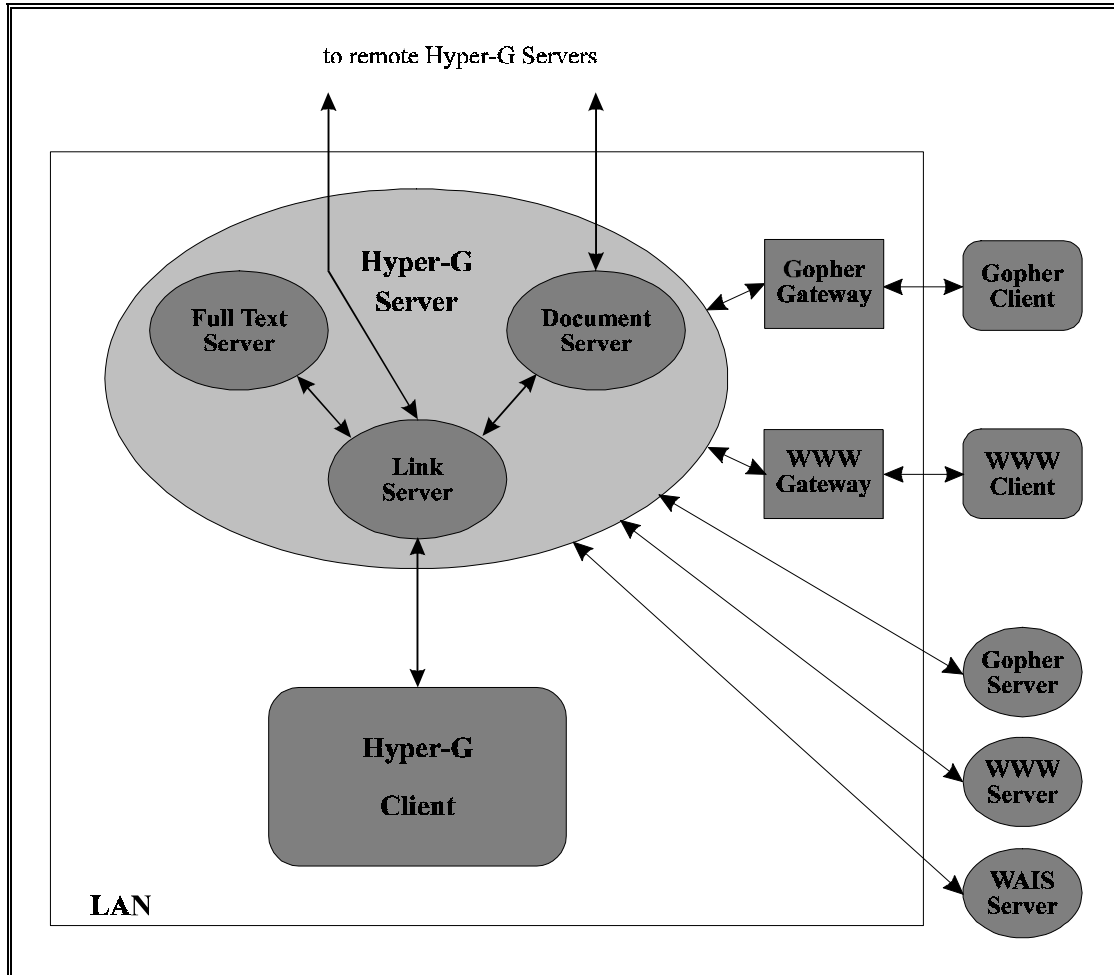


Figure 5.1: The Architecture of Hyper-G

For the client only the local server is visible; it connects to other Hyper-G servers and performs searches across server boundaries. As indicated in figure 5.1, the server consists of three distinct server processes:

The **full text server** offers full text retrieval facilities and automatic link generation.

The **link server** is an object-oriented database of objects and relations between objects. An object is a description of either a document, link, anchor, collection, tour, remote database, etc. The relations state which document belongs to which collection, which anchors are attached to which documents, etc. Its main functions are:

- It assigns object IDs to objects, ensuring that no two objects share the same ID and maps object IDs to objects (only the link server stores more information about an object, making it simple to modify objects, since they have to be modified only at the link server).
- It separates links from documents. So users can attach a link (for example an annotation) to documents, which they must not modify for various reasons (for instance because they

have no access rights, or the document is stored on a read-only medium such as a CD-ROM).

- It enables bidirectional links. They allow the client to draw overview maps and can be used to keep the database consistent when documents are modified or deleted (since links to non-existing or outdated documents can be found and handled automatically).
- It is aware of the collection hierarchy and uses it to maintain database consistency.
- It stores attributes of each object, such as title, keywords, author, creation time, etc. and thus allows to perform boolean search queries on the database.
- It contains a scheme of access rights to allow or disable the access to certain documents or collections to certain users or groups of users.
- It can gather detailed statistics about system usage.

Finally, the *document server* stores local documents and caches remote ones. The client connects to the link server and uses it to search and browse for documents. When a document is needed, it is retrieved from the document server. If the document is a remote one, the document server retrieves it from the remote document server, transmits it to the client and caches it. If the document is already in the document cache, it is transmitted to the client immediately.

5.3. Interoperability

Hyper-G is able to interact with Gopher [Alberti92], WAIS [Stein91], and WWW [Berners-Lee93] servers and Gopher and WWW clients.

Hyper-G can be accessed by Gopher and WWW clients. When using a Gopher client, the collection hierarchy is mapped into a Gopher menu tree; since hyperlinks cannot be represented in Gopher, one loses the possibility of hyper-navigation. A synthetic search item is generated at the foot of each Gopher menu to allow the user to search the corresponding collection.

When using a WWW client, each level of the collection hierarchy is converted to a HTML document, which contains hyperlinks to the members of the collection. Hyper-G text documents, including their links, are converted to HTML documents, other document types such as images or films are sent as pure data file and normally visualised by an external viewer. Special HTML documents are generated by the server to access some of the more sophisticated functions of the Hyper-G server such as search, identification, or the change of the language.

On the other hand, the Hyper-G server can connect to Gopher, WWW, and WAIS servers, in order to retrieve information from them. The Hyper-G server is able to store pointers to such remote objects. Gopher menus are transformed into Hyper-G collections. WWW text documents into Hyper-G text documents, other WWW documents (images, films, audio, ...) are sent directly to the appropriate Hyper-G viewer. WAIS queries and responses are transformed into Hyper-G queries and responses.

6. Harmony

Harmony [Andrews94a] is the native client for Hyper-G for X Windows [Nye88, Nye89] on Unix platforms. It takes advantage of Hyper-G's structural and retrieval features to provide intuitive navigational facilities and informative feedback about the location of information in the collection hierarchy. Harmony is written in C++ [Stroustrup91] and uses the InterViews toolkit [Linton89] to build the graphical user interface.

6.1. Architecture

Harmony is a multi-process Unix application (see figure 6.1). Its main process is the Session Manager. For each type of document there is a separate viewer process, which displays this kind of document. Currently there are six native Harmony Viewers: the Text Viewer, the Image Viewer, the Postscript Viewer, the Scene Viewer, the Film Player and the Audio Player.

The big advantage of this modular multi-process structure is, that new types of documents can be implemented easily, just by implementing a new viewer for this type of document. If no native viewer exists for a certain type of document, the Session Manager can also start an external viewer. Of course, external viewers do not support the whole functionality provided by Hyper-G and Harmony, in particular, they do not support hyperlinks.

This approach, with implementing native viewers, has some advantages, over a solution using only external viewers:

- All viewers, as well as the Session Manager, share a common user interface. If users know how to use one viewer, it is easy for them to use the other viewers as well.
- All viewers support links in their documents, thus creating a real hypermedia system, with links between any types of documents.

6.2. A Tour through Harmony

The Session Manager is the main process of Harmony. It starts the necessary viewers and initiates the communication between the viewers and the Hyper-G server. Its main functions and features are:

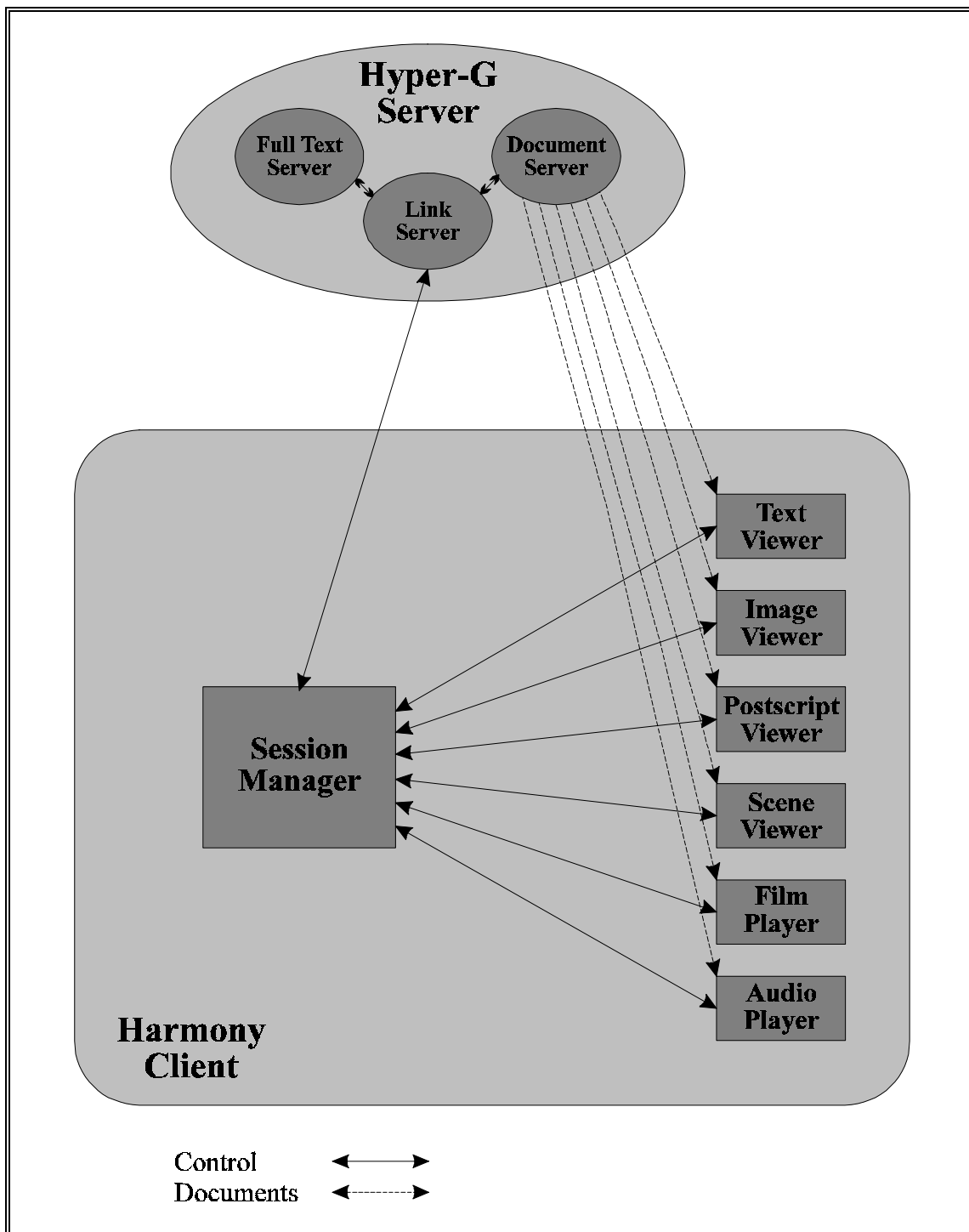


Figure 6.1: The Architecture of Harmony

The Session Manager's *Collection Browser* (the top left window in figure 6.2) shows the collection hierarchy and allows navigating through it. Furthermore, it shows the location of the current document within the collection hierarchy, independently by which means of navigation or search this document has been retrieved. This is important, because it shows the user the context of the current document.

The Session Manager provides an interface to Hyper-G's *Search Engine* (the bottom right window in figure 6.2): searches can be performed on attributes (title, author, keywords, ...) as well as full text in either a single (the current) collection, a set of activated collections or the whole local server.

The Session Manager provides a *History List* (the bottom left window in figure 6.2), which stores the accessed collections and documents of the current session as well as search operations and their results. In the example session shown in figure 6.2, the Hyper-G server of Graz University of Technology has been accessed. It first presented a welcome page (as can be seen in the History Browser). Then a collection containing system documentation has been opened and a search query for "grep" issued. Note that the Session Manager also inserts search queries into the History List, thus storing the search parameters and its result. The search found 9 objects containing "grep" in their title or keywords; two of them are text documents, the others are anchor objects.

As the first found text document was activated, its location in the collection hierarchy was displayed in the Collection Browser, showing that the text belongs to the collection "Hacker's Jargon", which indicates to users, that perhaps this document is not quite what they expected, and should not be taken too seriously. The second text document belongs to the collection "Manual Pages" and is probably, what the user was looking for.

The Session Manager creates on demand a *Local Map* (figure 6.3), showing the link structure around the current document, thus showing the "surroundings" of a document. The figure shows two levels of incoming and outgoing links of the text document "grep" previously accessed.

Figure 6.4 shows Harmony's support for *multiple languages*. The user can specify a list of language preferences. Harmony not only changes its user interface to the selected language, but also displays documents in the order given by the preference list.

An unique feature of Harmony is the *Information Landscape* (figure 6.5): it visualises the collection hierarchy as three dimensional cubes on a plane and allows the user to "fly" over this information space searching for data. The Landscape and the Session Manager's Collection Browser work synchronously: opening a collection or path in the Landscape also opens it in the Collection Browser and vice versa. Additionally, the Landscape provides an overview window to give users an overview where over the plane they currently are.

The Session Manager allows users to *update* the server database (figure 6.6): new documents can be inserted, the attributes of existing documents can be modified, or documents can be deleted. In the example shown in the figure 6.6, a new film has been inserted into the database and a German title has been added afterwards.

Hyper-G and Harmony is a true *hypermedia system*: figure 6.7 shows an example session with a text, an image, a film and an audio document presenting the Institute of Information Processing and Computer supported new Media (IICM).

But what really makes the hypermedia system is the support for links between any type of documents. Figure 6.8 shows information about the Austrian National Library. Starting at a text about the library, a film showing a flight through the great hall of the library was found; in this film there is a source anchor attached to the statue of Emperor Karl; by activating this source anchor, a link to a 3D scene document was followed. This 3D scene document visualises the model of the statue and allows the user to look at the statue from various point of views.

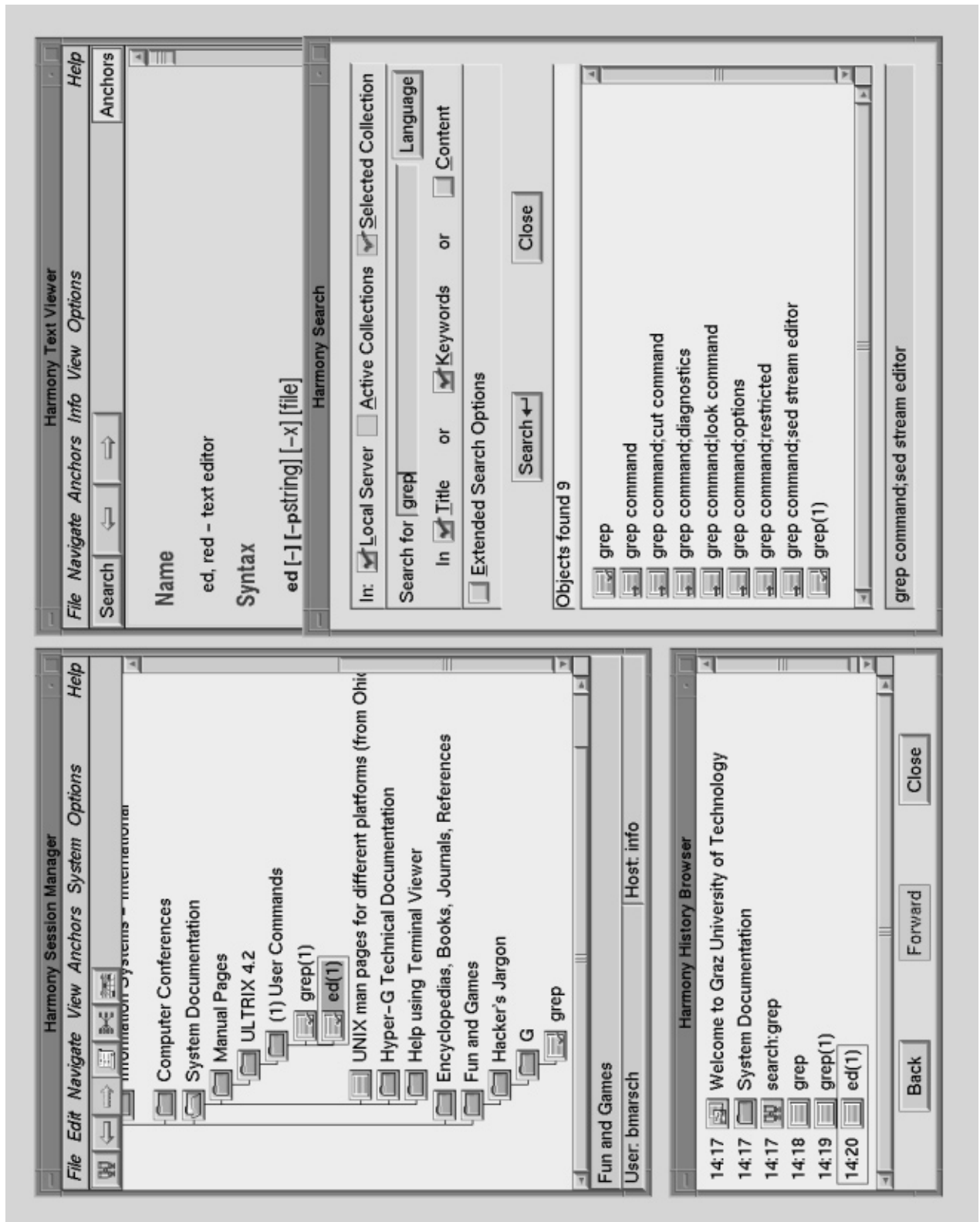


Figure 6.2: Session Manager: Collection Browser, History and Search Dialogue

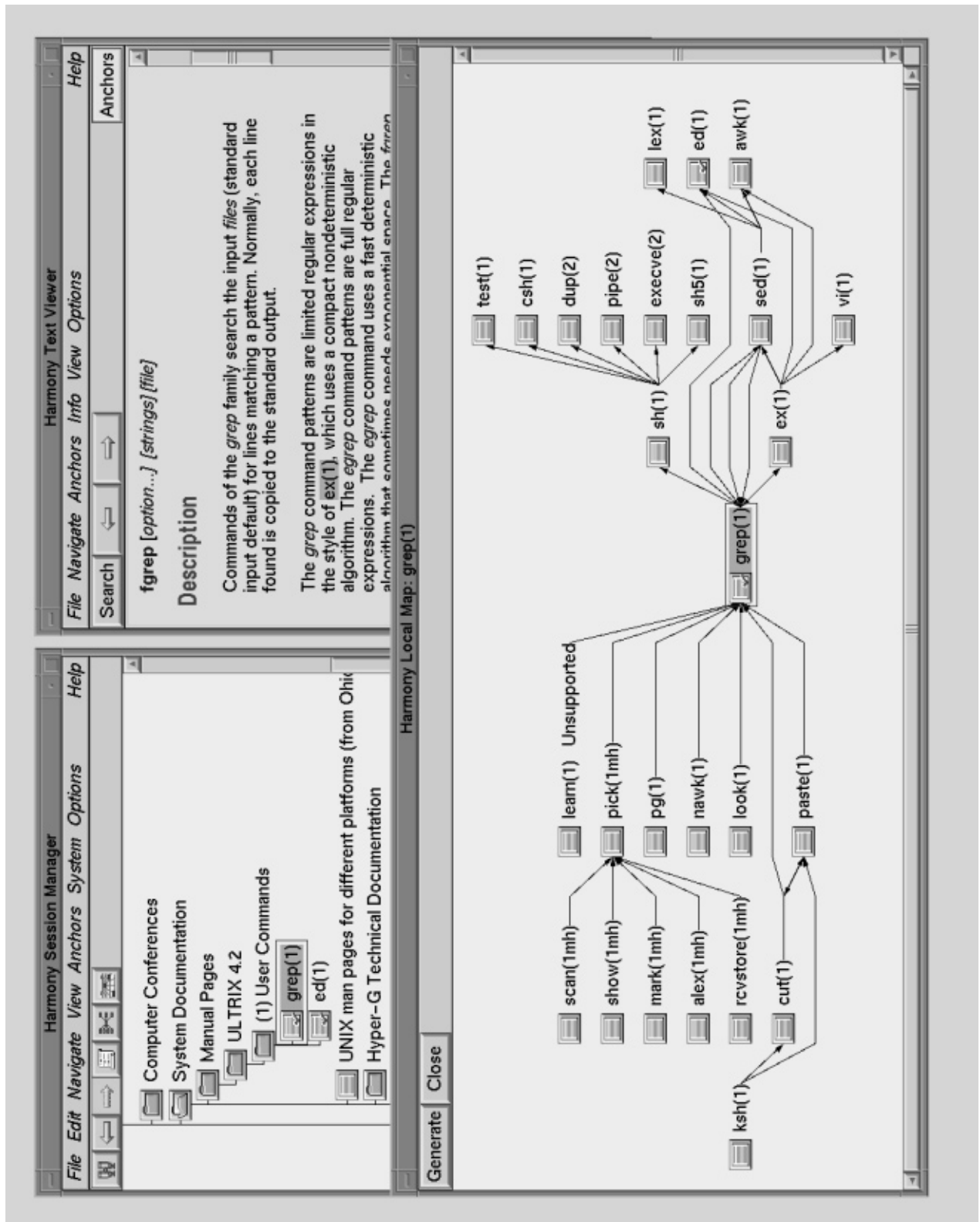


Figure 6.3: Session Manager: Local Map

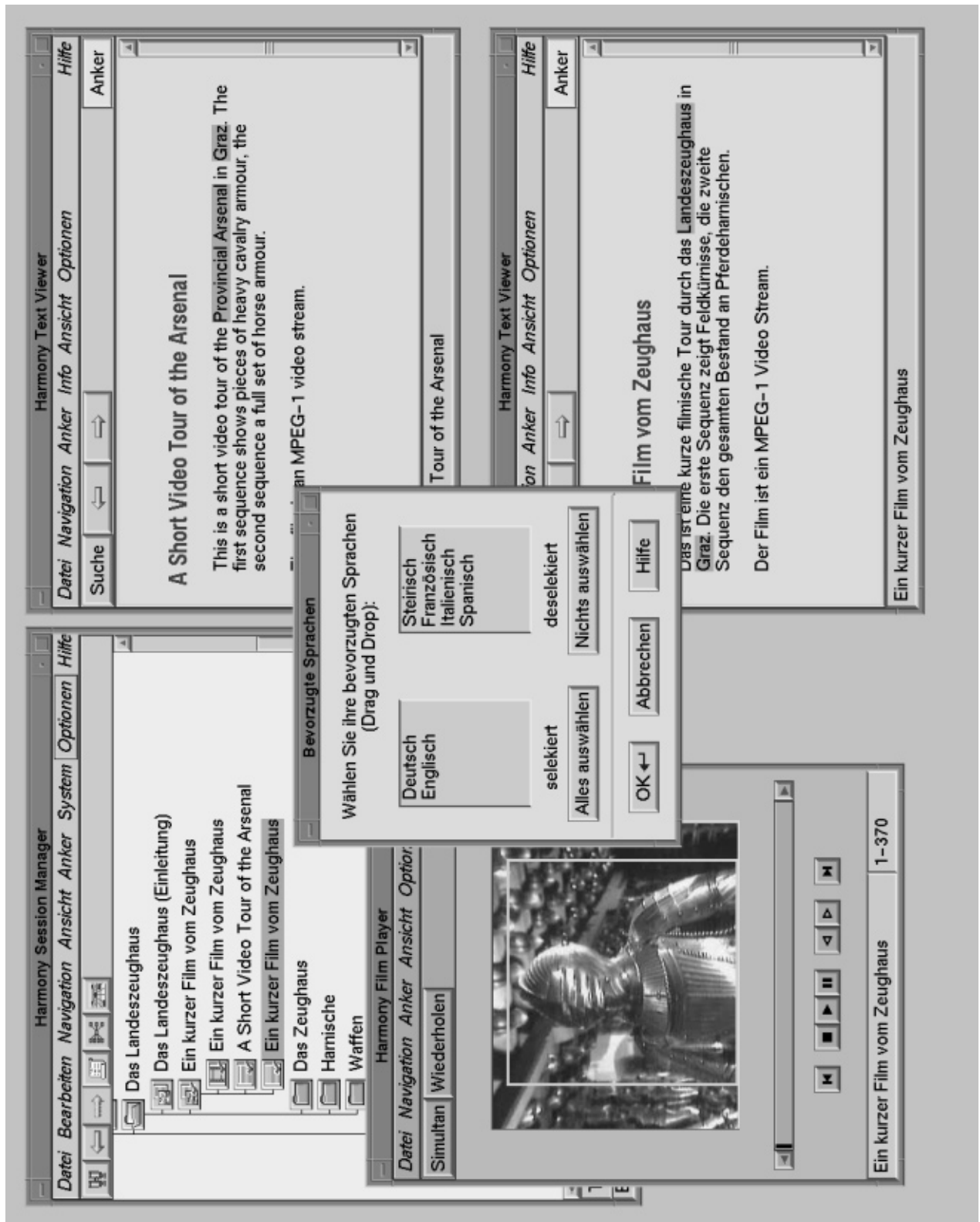


Figure 6.4: Multilingual Features of Harmony

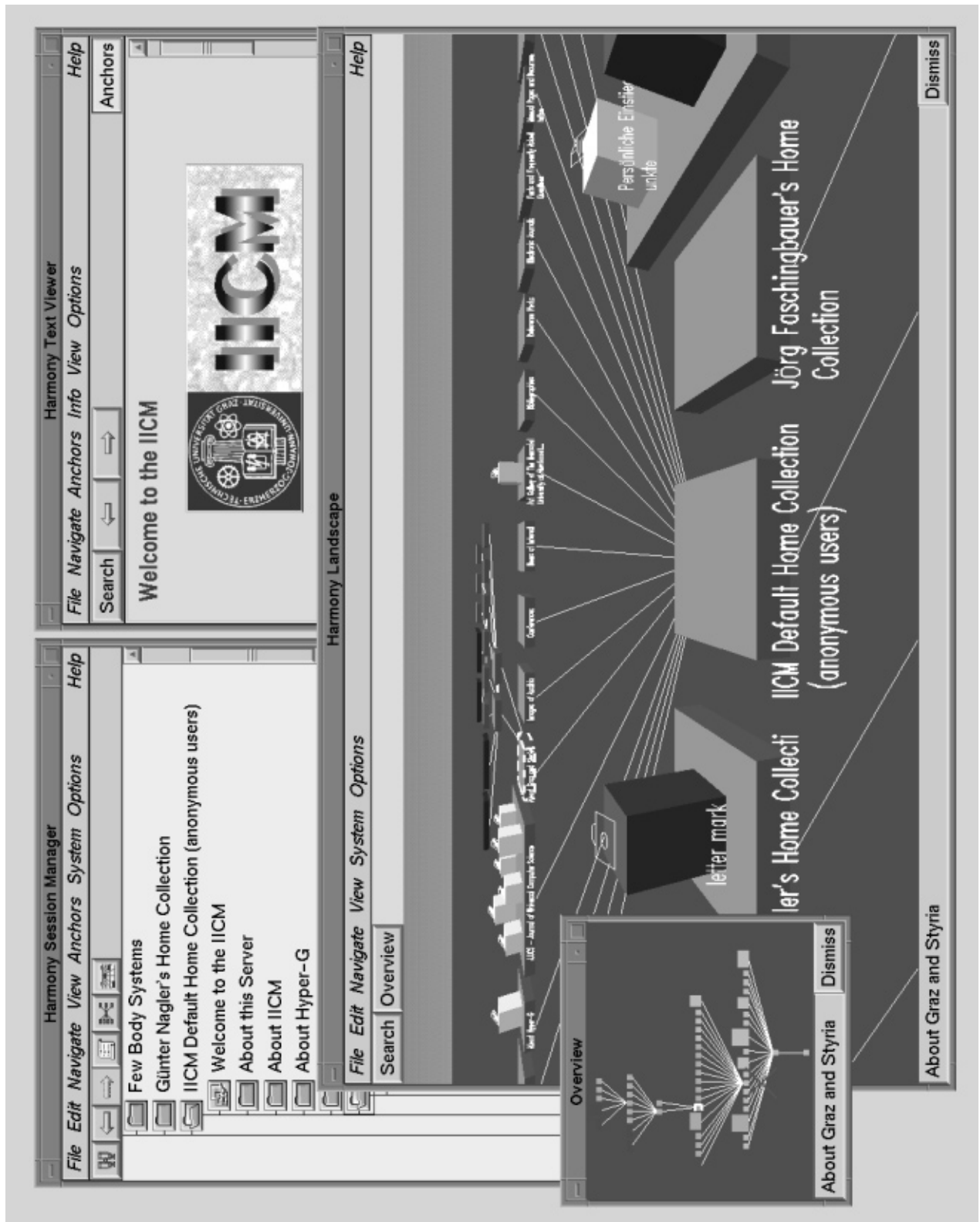


Figure 6.5: The Information Landscape

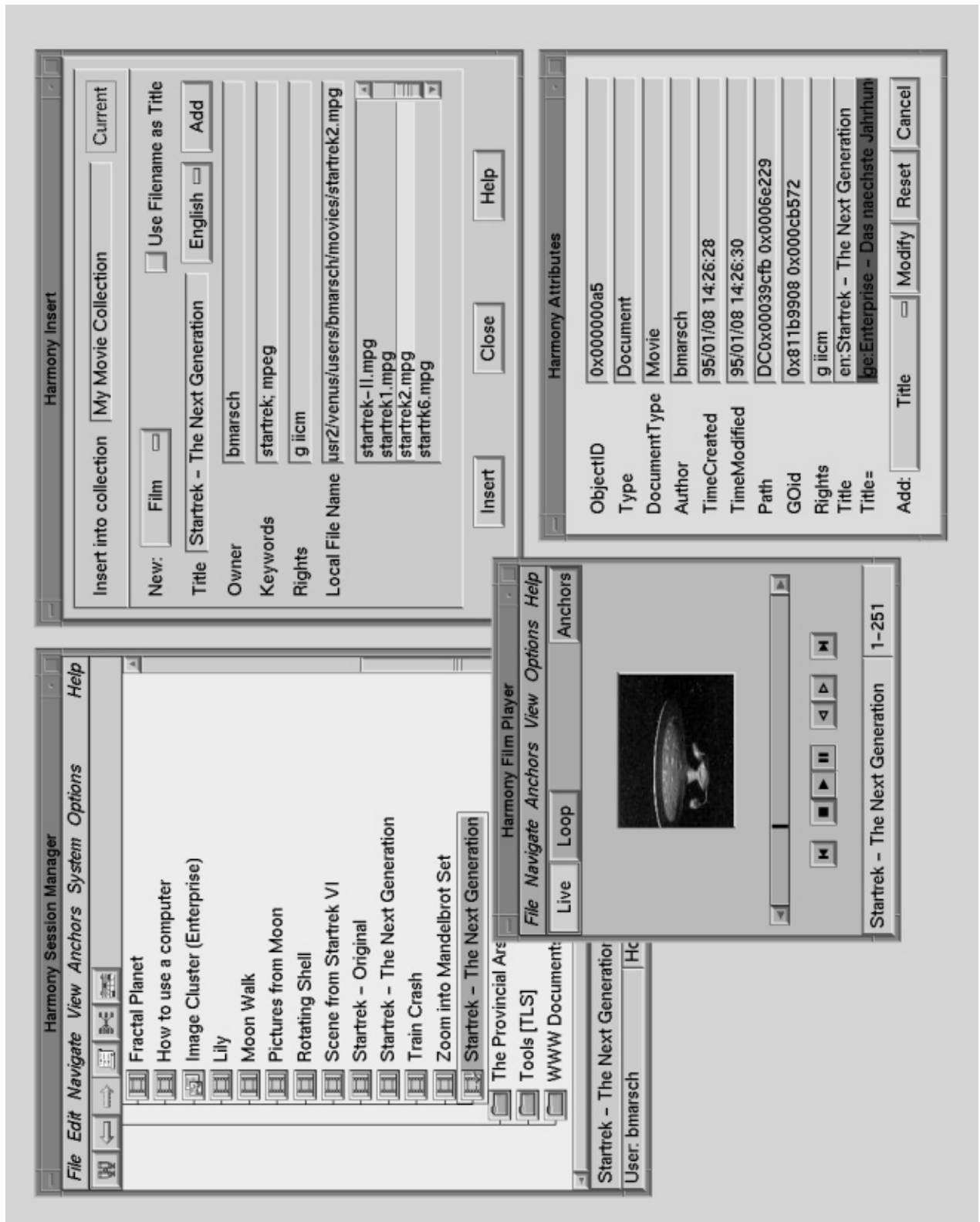


Figure 6.6: Session Manager: Insertion of new documents

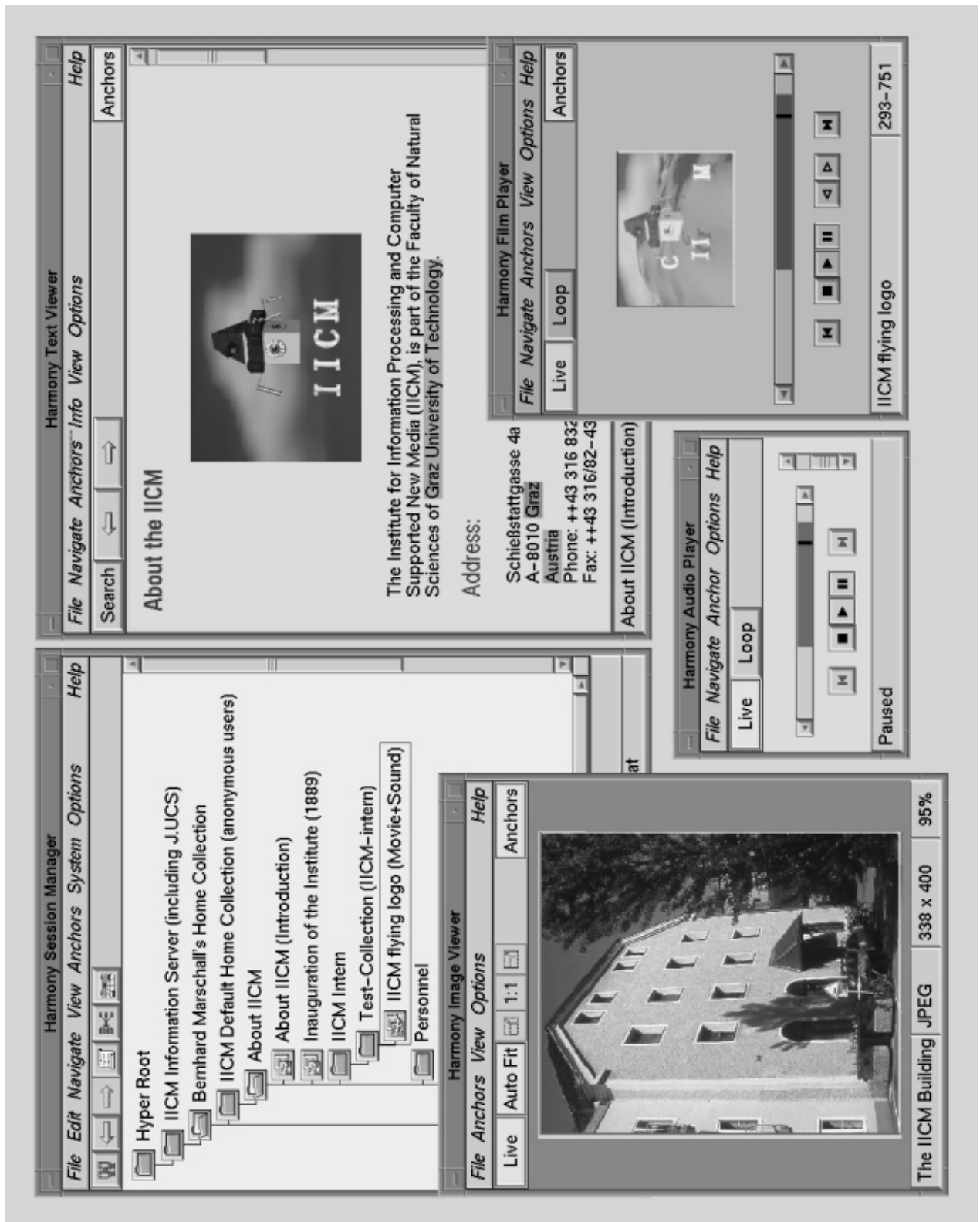


Figure 6.7: Multimedia Features of Harmony

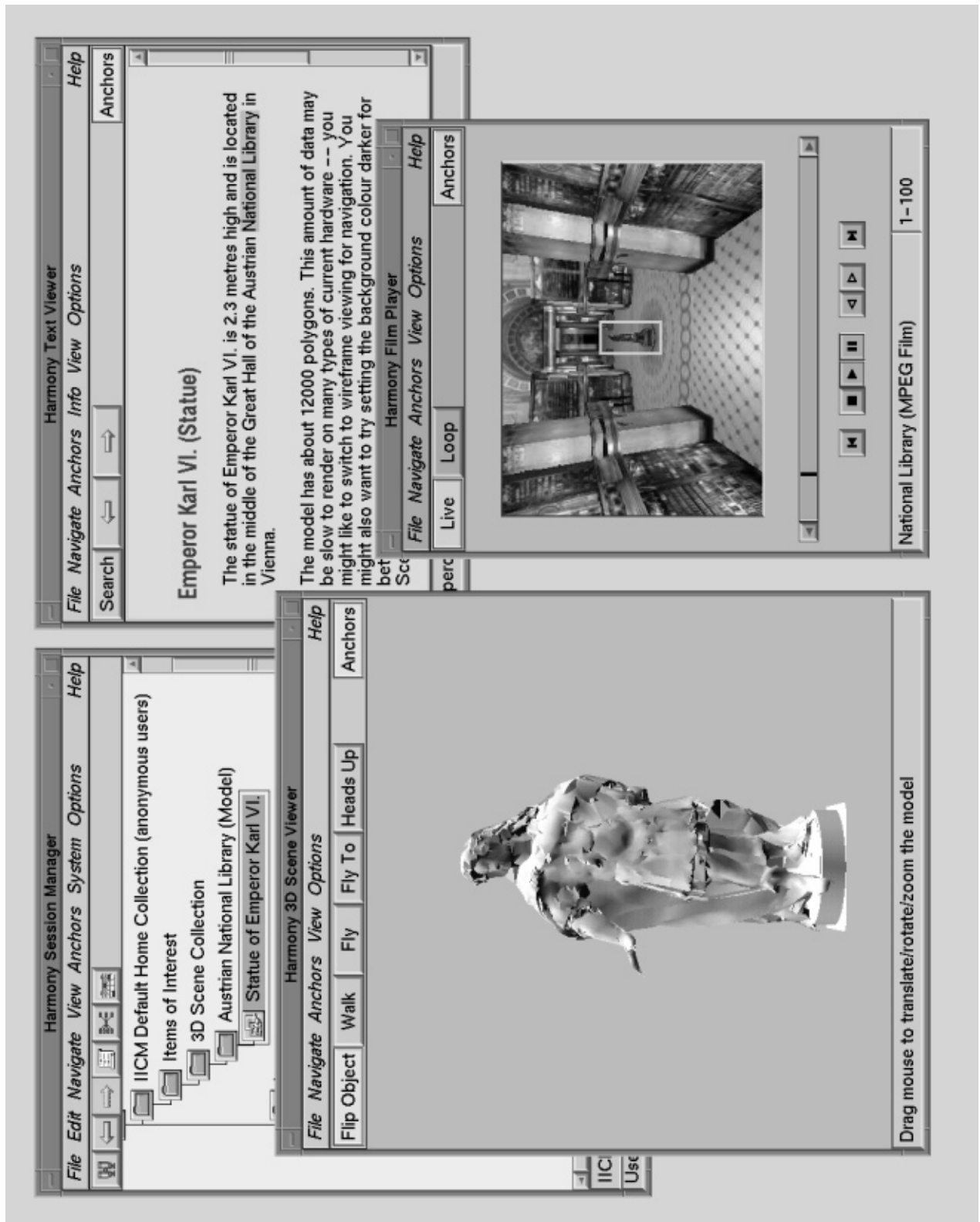


Figure 6.8: Hypermedia Features of Harmony

7. The Harmony Film Player

7.1. Using the Film Player

The Harmony Film Player is started by the Session Manager the first time the user wants to see a film. During loading (figure 7.1) the progress bar shows the percentage of data already read; simultaneously, if the live option is enabled, the film is already played.



Figure 7.1: Film Player during loading

As soon as loading is complete the control elements (scrollbar and push buttons) are shown and the first frame of the destination anchor is displayed (figure 7.2).

7.1.1. The Control Elements

Below the actual film there is a scrollbar and some push buttons: the scrollbar has two functions: First, it can be used to scroll along the time axis of the film (scrolling is done only on the I-frames of the MPEG stream for performance reasons). Clicking on the scrollbar with the *left mouse button* activates a "large" scroll, clicking with the *middle mouse button* causes the player to jump to the clicked position. Smooth scrolling can be done by dragging the mouse

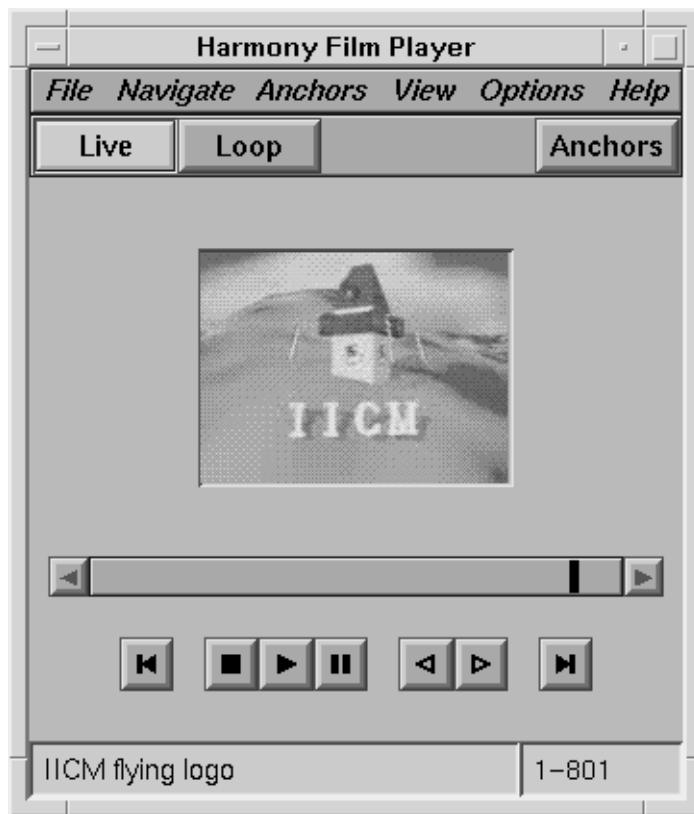








Figure 7.2: Film Player with control elements

when holding the left or middle mouse button. The two buttons at the left and right of the scrollbar can be used to do a "small" scroll.

The second function of the scrollbar is to mark a frame range. This can be done by dragging the mouse with the *right button* pressed. An already existing range can be modified by additionally holding down the *shift-key*: then, the boundary nearest to the cursor is updated, the other boundary is left unchanged. If a certain range of the film is marked, only this part of the film is played.

Below the scrollbar there are some push buttons. They have the following functions:

-  Play: start playing.
-  Pause: pause the film.
-  Stop: stop playing.
-  Step back: go to the previous picture.
-  Step forward: go to the next picture.
-  Rewind: go to the first picture of the film (or to the begin of the marked frame range, if there is one)



Forward: go to last picture of the film (or to the end of the marked frame range, if there is one)

Note that the pause and play buttons stay pressed to indicate the current state of the player. The functions of the back and forward step buttons are not the same as the buttons of the scrollbar: the first do a really frame-wise step, whereas the second only do a "small scroll" on the I-frames of the film.

7.1.2. The Menu and Button Bars

All functions of the Film Player (except play, pause, stop, step, rewind, and forward) are accessible by the menu. For some frequently used functions there are also shortcuts, either in form of a button in the button bar or as a keystroke. Most of the actions described here are also available during loading; if not it is stated explicitly below.

The File Menu

- *Open*: allows you to open and play a film from your local file system. When playing a local film, some functions of the player are not available and therefore disabled (for example all functions dealing with anchors and links). This function is not available during loading.
- *Save as*: allows you to save the current film to your local file system. This function is not available during loading.
- *Exit Viewer*: terminates the film player. (Note that the film player is automatically closed, when you terminate the Harmony session.)

The Navigate Menu

- *Goto Frame* (shortcut: "g"): open the Goto-Frame-Dialog window (see below).
- *Back* (shortcut: "b"): go to the previous object in the history list.
- *Forward* (shortcut: "f"): go to the next object in the history list.
- *History* (shortcut: "h"): open the History Browser.
- *Hold*: holds the current film in the Film Player; a new Film Player is started by the Session Manager, when the next film is to be played.

The Anchors Menu

- *Follow* (shortcut: "Return"): follow the selected source anchor.
- *Next* (shortcut: "Tab"): select the next source anchor.
- *Define as Source*: insert the newly defined path as source anchor into the database.

- *Define as Destination*: insert the newly defined path as destination anchor into the database.
- *Use Default Destination*: define the default destination (that is the whole film) as destination anchor.
- *Delete*: delete the selected source anchor.
- *Shape*: allows you to choose the shape for defining anchors. Currently only rectangles are available.
- *Interpolation*: allows you to choose the method for interpolating the shape in non-key frames, when defining anchors. Currently spline (a quadratic B-spline) and linear are implemented.

The View Menu

- *Settings* (shortcut: "^s"): open the settings dialog (see below).
- *Synchronise*: when activated, the film is played with a constant frame rate. Synchronised playing is not available during loading.
- *Zoom*: allows you to enlarge or reduce the size of the picture.
- *Anchors*:
 - Display: switch on or off the displaying of anchors.
 - Colours: allows you to change the colours of anchors.

The Options Menu

- *Loop*: when on, the player restarts playing the film, when the end has been reached.
- *Live*: when on, the player plays the film already during loading.
- *No B-Frames*: when activated, no B-frames of the MPEG stream are decoded and displayed. Switching off B-frames can speed up playback, and may therefore be useful on low performance machines.
- *No P-Frames*: when activated, no P-frames of the MPEG stream are decoded and displayed. If no P-frames are to be decoded, B-frames also cannot be decoded, so they are skipped too. This option may be useful on machines with extremely low performance.

The Help Menu

- *Overview*: display a tutorial about the film player; this function is not yet implemented.
- *Index*: display a help index; this function is not yet implemented.
- *About*: display the Harmony logo.

Buttons

Some of the most frequently used menu functions can also be activated using the push buttons of the button bar. These functions are: Live (same as Options/Live), Loop (same as Options/Loop) and Anchors (same as View/Anchors/Display).

7.1.3. The Progress Indicator

The bottom line of the film player is the so called progress indicator. It shows the current state of the player and has two major modes:

- During loading (figure 7.1) it displays a progress bar showing the percentage of data already read. Pressing the stop button (at the right of the progress indicator) causes the player to abort loading. The already loaded part of the film can be played normally. Note that aborting loading may take a while, since nevertheless the player has to wait for all data of the picture it is currently decoding.
- After loading (figure 7.2) the progress indicator shows the title of the current document, and the current frame range.

7.1.4. Dialogs

The Goto-Frame Dialog

This dialog (figure 7.3) allows you to jump to any frame of the film by just entering its number and pressing return or clicking on the *Goto-Button*. The dialog window can be closed by clicking on the *Close-Button*.

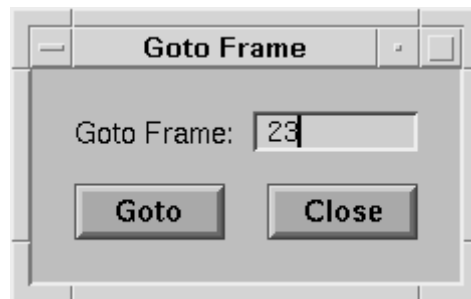


Figure 7.3: The Goto-Frame Dialog

The Settings Dialog

The Settings dialog (see figure 7.4) provides three functions; it can be closed by pressing the *Close-Button*.

The first part of the dialog allows you to change the brightness of the frames. The Film Player does a gamma correction; gamma may range from 0.3 to 2.5 and can be changed either

by the scrollbar or by directly entering the desired value into the field editor. Pressing the *Reset-Button* resets the value of gamma to 1.0.

In the middle part of the dialog you can set the frame rate (i.e. the number of frames displayed per second). As with gamma, the frame rate can be changed using the scrollbar or the field editor. Possible frame rates are between 1.0 and 60.0. Pressing the *Default-Button* resets the frame rate to the default value stored in the MPEG header. Synchronised playback can be turned on or off with the *Synchronise-Button*; this button has the same function as (and is synchronous with) the View/Synchronise menu item.

The third part of the dialog can be used to mark a frame range. This is an alternative to marking a frame range with the Film Player's scrollbar. The numbers of the start and the stop frame can be entered into the two field editors. Pressing the *Reset-Button* clears the currently marked frame range.

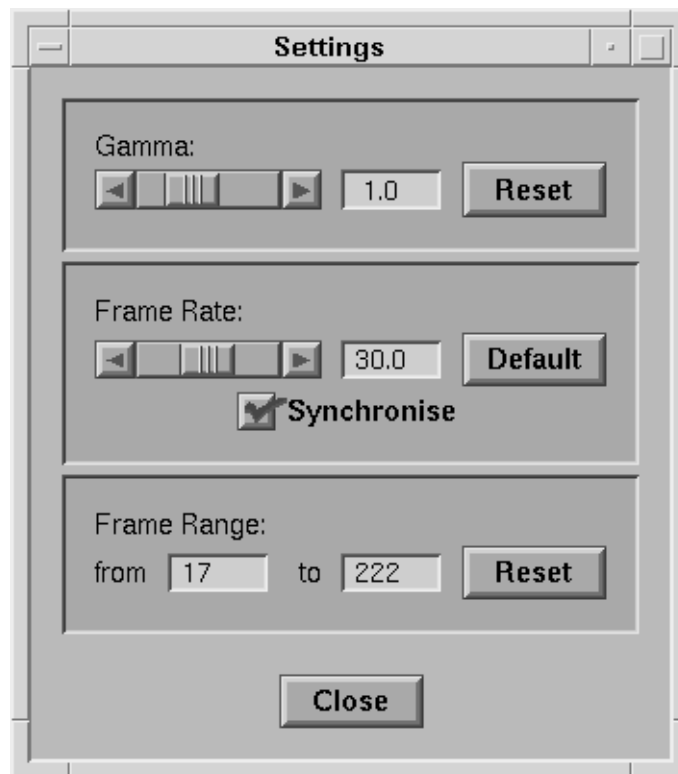


Figure 7.4: The Settings Dialog

7.1.5. Selecting and Following Anchors

Using the Anchor button or the menu item View/Anchors/Display the anchors can be switched on or off.

There are two ways to select a source anchor: Just click (once) into the region defined by the anchor with the *left mouse button* and it will be selected. If more than one anchors are hit, some (arbitrary) anchor is selected and you can step through all hit anchors by holding down the *shift key* and clicking with the left mouse button. If you click outside any anchor, all anchors will be unselected. The second way is to use the menu: select *Anchors/Next* to step

through all anchors visible in the current frame. This menu function has the *TAB-key* as shortcut. The selected anchor is displayed in a different colour.

To follow a link you have to activate its source anchor. This also can be done in two ways: *double-clicking* with the left mouse button will activate the selected source anchor, if it is hit, or (if no anchor is selected yet or the selected anchor is not hit) an arbitrary hit anchor. The other method is to activate the selected anchor using the *Anchors/Follow* menu item; the same action is activated by pressing the *RETURN-key*.

Figure 7.5 shows a film document with two source anchors; the lower right of them is selected and could be followed by pressing RETURN or choosing Anchors/Follow:



Figure 7.5: Selecting and Following Source Anchors

7.1.6. Defining and Deleting Anchors

The Harmony Film Player (as do all native Harmony viewers) allows users to define source and destination anchors.

The most simple way to define an anchor, is to define the whole film as destination anchor; this can be done by choosing the *Anchors/Use Default Destination* menu item.

To define any general path in a film as anchor take the following steps:

- Choose the shape of the anchor in the *Anchors/Shape* menu; currently, you can choose between rectangles and circles. The shape of the anchor cannot be changed afterwards!
- Define key frames: just scroll to any frame of the film (either by using the scrollbar, the step buttons, the Goto-Frame Dialog, or just playing to the frame) and mark the region of interest (i.e. the rectangle or circle) by dragging the mouse with the *right mouse button* pressed. If a region has been marked already in this frame, it is replaced by the new one. Just clicking with the right mouse button deletes the current region. Alternatively you can change the marked region by additionally holding down the *shift-key*; this allows you to drag the definition point closest to the mouse cursor. In this manner, define as many key frames as you need or like.
- Choose a method of interpolation in the *Anchors/Interpolation* menu; this can be done anytime during the definition of key frames. The marked regions in the key frames are used to calculate the regions in all frames between two key frames. Currently you can choose between linear interpolation and spline interpolation. When using linear interpolation, the region between two key frames is just calculated as the linear interpolation of the regions of the two neighbouring key frames. Linear interpolation is very simple, but may result in jerky motion. When using spline interpolation, the regions of interest are calculated using a

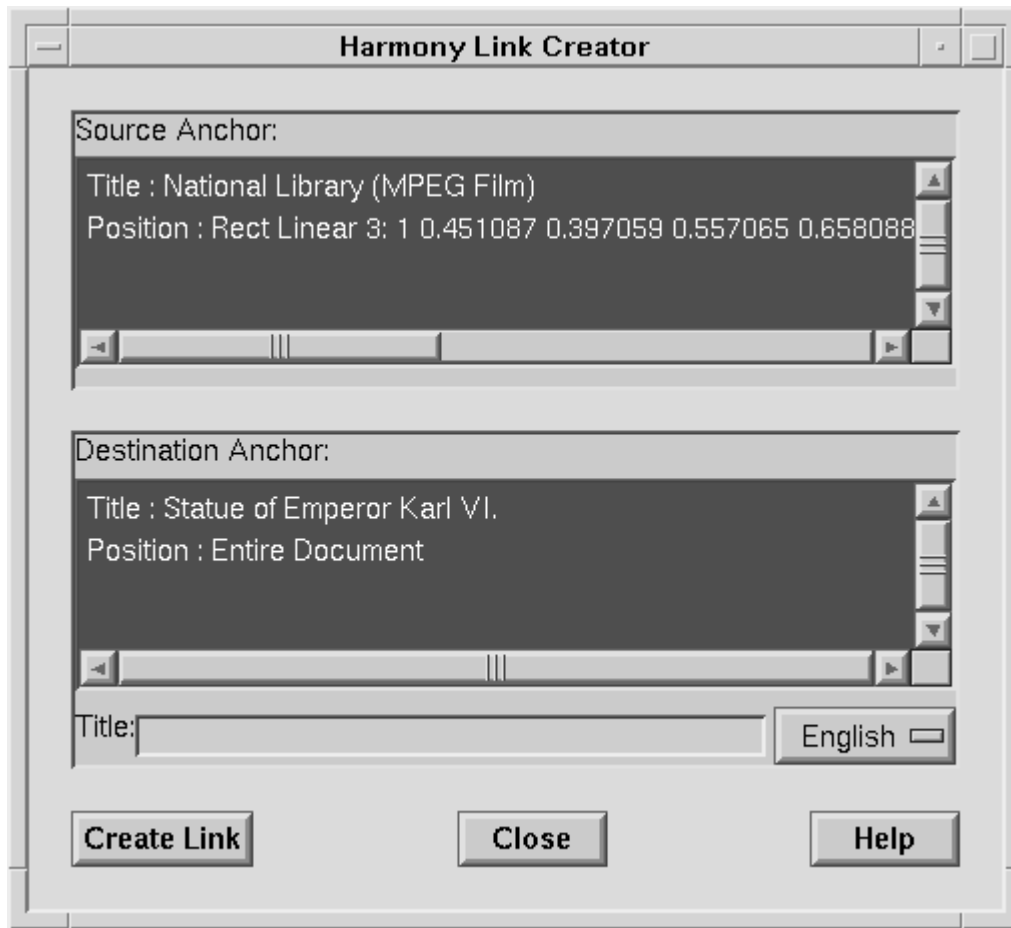


Figure 7.6: The Harmony Link Creator

quadratic B-spline. This spline results in smooth motion, but in general the defined regions in the key frames do not belong to the spline.

- Finally, when the defined path fits (which can be checked by just playing the film, or scrolling to any frame, since the path is recalculated each time you change a key frame or the method of interpolation), you can define this path either to be a source anchor by choosing the *Anchors/Define as Source* menu item or to be a destination anchor by choosing the *Anchors/Define as Destination* menu item. This causes the viewer to send the definition of the anchor to the Session Manager, which now opens the Link Creator (see figure 7.6) to display the definition of the "new" anchor. Now, you have to define the other end of the link in any other (or the same) document. The link is actually created and inserted into the database, when you press the *Create-Link* Button in the Link Creator.

Figure 7.7 shows a newly defined path and the Anchors Menu. By selecting *Define as Source* or *Define as Destination* this path can be defined as source or destination anchor respectively.



7.7: Defining a new Anchor

To delete a source anchor, just select it and choose the menu item *Anchor/Delete*. The Session Manager will display a confirmation dialog and actually delete the anchor (as well as the corresponding link and destination anchor).

7.1.7. The Film Player's X-Resources

The Harmony Film Player defines some X resources (additionally to those already defined by InterViews). The value of a lot of the resources can be changed interactively as well, using some menu function. For each resource its domain and default value (if any) is given in brackets.

The main resource class is *Harmony.Film* with the resources

- *shmem* [boolean][off]: use shared memory extension of X.
- *anchordisplay* [boolean][on]: show anchors.
- *live* [boolean][on]: play film during loading.
- *loadtoend* [boolean][on]: when on, the film is played to the end during loading; when off, the first frame of the destination anchor is shown immediately after all data arrived.
- *loop* [boolean][off]: rewind the film and start playing automatically when the end of the film has been reached.
- *dither* [string][color]: default dithering method. Possible values are *color* (for true colour displays), *ordered* (ordered dither), *fs* (Floyd-Steinberg dithering), *hybrid* (hybrid of ordered dithering and Floyd-Steinberg), and *gray* (grayscale dithering).
- *anchordisplay* [boolean][on]: display anchors.
- *activateFeedbackTime* [float][0.5]: how long (in seconds) the hourglass cursor is shown, when a link is followed.
- *brushWidth* [int][1]: line width for rubberbanding.
- *linewidth* [int][2]: line width to draw the anchors with.
- *anchorColour* [colour]: colour of source anchors.
- *anchorSeenColour* [colour]: colour for already seen source anchors.
- *selectedAnchorColour* [colour]: colour for the selected source anchor.
- *destinationColour* [colour]: colour for the destination anchor.
- *markerColour* [colour]: colour for marking new anchors.

There are resource subclasses to define the geometry and look of the dialog windows. These are: *Harmony.Film.Open*, *Harmony.Film.Save*, *Harmony.Film.GotoFrame*, and *Harmony.Film.Settings*. All of them also use the alias *Harmony.Film.Dialog*.

7.2. Implementation Details

7.2.1. The MPEG Decoder

The base of the Harmony Film Player is the public domain Berkeley MPEG decoder of the University of California [Patel93]; it is available by anonymous ftp from mm-ftp.berkeley.cs.edu:/pub/multimedia/mpeg/mpeg_play-2.0.tar.Z. It is written in C and has a simple interface to the X-Windows system to display frames. First, its most important data structures are described, followed by the functional interface, the dithering algorithms and finally the changes made to the Berkeley code.

Data Structures

The decoder defines the structure *VidStream*, which contains all information about the MPEG stream and its current state of decoding:

```
/* Video stream structure. */
typedef struct vid_stream {
    unsigned int h_size;          /* Horiz. size in pixels.    */
    unsigned int v_size;          /* Vert. size in pixels.    */
    ...
    Pict picture;                 /* Current picture.        */
    ...
    unsigned int *buf_start;       /* Input buffer             */
    unsigned int *buffer;         /* Pointer to next byte in
                                   input buffer.            */
    ...
    PictImage *past;              /* Past predictive frame.  */
    PictImage *future;           /* Future predictive frame.*/
    PictImage *current;          /* Current frame.          */
    PictImage *ring[5];          /* Ring buffer of frames.  */
} VidStream;
```

The most important sub structures of *VidStream* are *Pict* and *PictImage*; a *Pict* contains the information about the current picture (most important its type I, P or B):

```
/* Picture structure. Contains compressed image */
typedef struct pict {
    ...
    unsigned int code_type;       /* Frame type: P, B, I */
    ...
} Pict;
```

A *PictImage* contains a whole uncompressed picture in the $YCbCr$ colour space. *VidStream* contains a buffer of *PictImages* and three pointers to this buffer pointing to the current frame, the future and the past reference frame (possibly) needed to decode the current frame.

```

typedef struct pict_image {
    unsigned char *luminance;           /* Luminance plane.    */
    unsigned char *Cr;                 /* Cr plane.           */
    unsigned char *Cb;                 /* Cb plane.           */
    unsigned char *display;            /* Display plane.      */
    ...
} PictImage;

```

The Functional Interface

The main interface to the MPEG decoder is the function *void mpegVidRsrc(TimeStamp, VidStream*)*, which decodes a fixed amount of macroblocks of the given stream; whenever a frame is completely decoded, it is dithered and displayed (it should be displayed at the time indicated by the TimeStamp, but this feature has not been implemented yet; so this argument is ignored).

The function *void DoDitherImage(...)* is called by the decoder to dither a picture before it is displayed. It is essentially a big switch statement, which calls the appropriate dithering function.

Function *int get_more_data(...)* is called by the decoder, when its internal buffer runs out of data. It reads data from the input file and writes it into a buffer; if the end of the file has been reached, a sequence end code is written into the buffer.

Dithering and the Harmony Colormap

MPEG stores the picture data in YC_bC_r colour space; on the other hand, all display hardware displays colour in RGB colour space. This conversion of the colour spaces is a time critical point, since it has to be done for each pixel. On pseudocolour devices, which cannot display all 2^{24} colours at once, the picture also has to be dithered.

The Berkeley decoder provides several different dithering algorithms, which are called by *DoDitherImage()*. One dithering algorithm is for truecolor displays; it just does the YC_bC_r to RGB conversion. The other algorithms really do dithering: the clever thing about them is, that they do the dithering in YC_bC_r colour space, thus reducing the number of colours to 128 (in YC_bC_r); during initialisation, these 128 colours are converted to RGB and allocated in the system's colormap. That means, that on pseudocolour devices, the colour space conversion is done by the colormap (i.e. by the display hardware), which essentially reduces the execution time.

These dithering algorithms use 128 colours. They are equally spaced in YC_bC_r colour space with 8 values for Y, 4 values for C_b , and 4 values for C_r . All Harmony viewers (and of course also the Session Manager) should use the same colormap, otherwise some windows would be displayed in false colours. Since the above 128 colours are constructed in YC_bC_r colour space, they do not contain gray tones, which on the other hand are likely to be used for the user interface (for instance push buttons or menus). So the Harmony colormap contains the above 128 colours (two of them are double in RGB colour space, so there are only 126 unique colours), 16 gray tones (from black to white, equally spaced) and the 6 primary colours red, green, blue, cyan, magenta, and yellow, making 148 colour totally. Normally pseudocolour devices have a colour depth of 8, which means 256 colours can be displayed at once; so Harmony leaves 108 colormap entries to other applications, which is surely enough for other

non-graphical applications, running parallel to Harmony. Appendix A shows the entire Harmony Colormap.

Changes to the Decoder

A global variable *curFrame* has been added, which counts the frame currently being decoded. It is needed by the viewer to keep track of the playing.

The return value of function *mpegVidRsrc()* has been changed; now it returns an integer value, which reports information about the state of decoding (which start codes has been read, which type of frame is currently being decoded, if a frame has been fully decoded, or if a frame has been displayed).

DoDitherImage() is already implemented in *mpeg_play*, which comes with the MPEG decoder. Only some new dithering functions to display with double and half size for truecolor as well as pseudocolour and grayscale, have been added. Grayscale dithering had to be reduced from 128 grayscales to the 16 ones contained in the Harmony colormap. Unfortunately this leads to a clearly visible loss of quality, but on the other hand it is not possible for the Harmony Film Player to allocate 128 grayscales (as *mpeg_play* does), thus leaving no colours for the other Harmony processes (in particular for the Image Viewer).

To be able to **change the brightness** of the frames (when they are displayed; not in the MPEG file itself), a new table has been added. The dithering algorithms do not use the luminance values directly, but use them as an index into this table; the value stored in this table is used as actual luminance value. By setting this table appropriately any function on the luminance values could be implemented. Currently the table is set up to perform a gamma correction (i.e. $Y' = Y^{1/\gamma}$).

get_more_data() also was taken from *mpeg_play* and extended to support decoding while loading data from a remote database: a new global variable *loading* has been added (as its name suggests, it is set when the player loads new data from the server). When *loading* is set and *get_more_data()* could read no more data from the input file, it calls a function (*HgMpeg::waitForData()*) to allow the player to read new data and append it to the input file.

The decoder has been extended to support **synchronised playback**. For that purpose, whenever a frame is displayed and synchronisation is on (flag *doSynch*) a new function *void SynchStream(VidStream*)* is called. It checks, if decoding was in time. When decoding was too fast (for instance because the user wants to play the film in slow motion), the decoder delays for the appropriate time. When decoding was too slow, the corresponding number of frames is skipped. Since frames in MPEG frames can depend on other ones, it is also necessary to skip all frames, which depend on already skipped ones. That means whenever an I- or P-frame has to be skipped, all following frames up to the next I-frame also have to be skipped. This very simple scheme of synchronisation causes the film to be played rather jerkily, because generally too many frames are skipped, which may cause the player to delay after displaying the next frame, just to find that it is late, when displaying the next but one frame. A more sophisticated synchronisation scheme (see [Rowe94]) would try to skip at first only B-frames; if that is still too slow, it would also skip P-frames, and finally I-frames.

7.2.2. Encapsulating the MPEG decoder

Beside the above functions, there are some global variables, which are important to the Harmony Viewer, such as the total number of frames, the number of the current frame, the dithering type, and some more. All these have been encapsulated into the C++ *class HgMpeg* to "hide" the details of the decoder. (Since these variables and functions are global C variables and functions, this cannot really prevent the application from accessing them directly; the class just provides a clean interface without forcing its user to deal with the decoder's details.)

```
class HgMpeg {          // encapsulation of Berkeley MPEG decoder
public:
    HgMpeg();
    virtual ~HgMpeg();
    static HgMpeg* instance();

    virtual void initXDisplay(Window*); // initialize display

    unsigned int width() const;        // get width of frame
    unsigned int height() const;       // get height of frame
    int totNumFrames() const;          // get total number of frames
    int curFrame() const;              // get current frame
    boolean getSynch() const;          // get synchronisation flag
    void setSynch(boolean synchronise); // set synchronisation flag
    boolean getLoop() const;           // get loop flag
    void toggleLoop();                 // toggle loop flag
    double defaultFrameRate() const;   // get default frame rate
    double getFrameRate() const;       // get actual frame rate
    void setFrameRate(double framerate); // set actual frame rate
    int getDitherType() const;         // get dither type
    void setDitherType(int);           // set dither type
    void changeDither(int);            // change dither type
    boolean getBFlag() const;          // get no B-frames flag
    void toggleBFlag();                // toggle no B-frames flag
    boolean getPFlag() const;          // get no P-frames flag
    void togglePFlag();                // toggle no P-frames flag
    boolean trueColor() const;         // TrueColor visual exists?

    virtual void waitForData() {};
    virtual void drawSpecial() const {};
    virtual void error(const char* msg);

protected:
    int decodeMpeg();                  // decode a bit of the stream
    void nextStartCode();              // parse to next start code
    void initVidStream();              // initialize struct VidStream
    void setDitherType(const char* ditherType);
                                        // set dither type by name

    void inputFile(FILE* input);       // set input file
    void noDisplay(boolean noDisplayFlag);
                                        // set noDisplayFlag

    void resetSynch();                 // start synchronisation
    void setIO();                      // set global I/O-variables
    void resetIO();                    // reset global I/O-variables
    void totNumFrames(int numFrames);  // set total number of frames
    void curFrame(int curFrame);       // set current frame
};
```

Only one object of *class HgMpeg* per application has to be created; this object is returned by the static member function *instance()*. This was necessary to be able to call member functions of *HgMpeg* in functions of the decoder (for instance in *get_more_data*) without having to pass the actual object all the way down, which would have made necessary a change in the whole interface of the decoder.

By calling *initXDisplay()* a window is passed to the decoder; into this window the decoder displays the frames. The window itself is created and mapped by InterViews (see chapter 7.2.8. on page 69).

There are a lot of member functions to get information about the MPEG stream (for example the size of the frames, the number of frames and so on) or to change the behaviour of the decoder (like switching on or off synchronisation, changing the frame rate, changing the dithering algorithm and so on), which are rather straightforward.

The following virtual members can be implemented in a subclass:

- *waitForData()* is called by the decoder, when it runs out of data and the loading flag is set. When *HgMpeg* is used to build a standalone player this function can be left empty, and the loading flag must not be set.
- *drawSpecial()* is called immediately after a frame has been displayed; the Film Player uses this function to draw the anchors above the picture.
- *error()* is called when an error condition occurs during decoding; a description of the error is passed as parameter. The default behaviour of error is to print this description on stderr. If the application wants to deal with errors differently, it has to overload this function.

7.2.3. The Harmony Communication Protocol

The viewer has two channels of communication (see figure 7.8): the first one, which

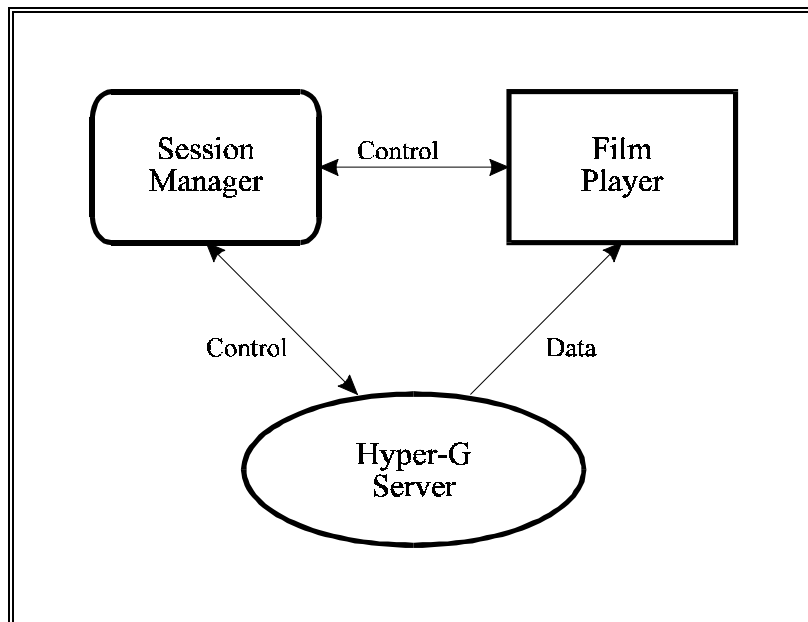


Figure 7.8: The Harmony Communication Protocol

handles the flow of control, is to the Session Manager, the second one, over which the actual data are sent, is to the Document Server within the Hyper-G server.

The first time the user wants to see a document of a certain type (in case of the Film Player, a film), the Session Manager forks a viewer daemon process. When the Session Manager wants to attach a new viewer to a document (the first time a document of this type is requested, or the user wants to hold the current document), it makes an IPC connection to this daemon, which then forks the actual viewer.

Each Harmony viewer is derived from *class HgViewer*:

```
class HgViewer {
public:
    HgViewer(HgViewerManager* manager);
    virtual ~HgViewer();

    virtual void load(const char* document,
                     const char* source_anchors,
                     const char* info = 0);
    virtual void browse(const char* destination_anchor);

    virtual void map();
    virtual void unmap();
    virtual void iconify();
    virtual void deiconify();
    virtual void raise();
    virtual void lower();

    virtual void terminate();

    virtual void setLanguage(int language);
    virtual void getPos(RString& position);
    virtual int port();
};
```

The viewer opens a port, which will be used to receive the documents; the Session Manager asks for this port using the function *port()*.

The following steps are taken to load a document:

- The Session Manager tells the Hyper-G Link Server to send a document to the above viewer port.
- The Link Server checks if the document is available and if the user has appropriate access rights; if not, an error is returned to the Session Manager.
- The Link Server tells the Document Server to send the document to the viewer.
- The Document Server tries to open an IPC connection to the viewer. If the connection is not accepted by the viewer for some reason, the Document Server returns an error to the Link Server, which in turn passes it to the Session Manager.
- The Session Manager tells the viewer to load the document by calling the function *load()*; with the load call the viewer also gets the object description of the document (object id, title, author, ...) and the list of all source anchors attached to the document.

- Afterwards the Session Manager calls *browse()* to tell the viewer to display the document; as parameter the viewer gets the destination anchor attached to the document.

All **object descriptions** (i.e. the description of documents, anchors, ...) come to the viewer as a string (either as `const char*` or as `RString`, which is the IICM implementation of a string class); this string is build up of fields, which are separated by newlines, the string ends with two newlines. Each field contains a field name and a value; these are separated by an equal-sign. The following example shows the description of some film document:

```
const char* document =
  "ObjectID=0x00000047\n"
  "Type=Document\n"
  "DocumentType=Movie\n"
  "Author=hgsystem"\n"
  "TimeCreated=94/05/10 09:09:54\n"
  "TimeModified=94/10/10 13:50:55\n"
  "Title=en:A Short Video Tour of the Arsenal\n"
  "Title=ge:Ein kurzer Film vom Zeughaus\n"
  "Path=DC0x00031df7 0x000a9e4e\n"
  "GOid=0x811b9908 0x00002c26\n"
  "\n"
```

Two fields of the document's description coming with *load()* are of particular interest: the field *Path* has the form "*Path=document_path document_size*". Since MPEG nowhere stores the total number of frames or the size of the stream, the second value of the *Path*-field is used to determine the size of the MPEG stream (in bytes); this size is needed to display a progress during loading.

If the viewer has already loaded the document and only the anchors changed (for instance because the user has inserted or deleted an anchor), or a link whose destination is in the same document has been followed, the document's description contains the additional field *Function* with the value *reload*. This tells the viewer to use the already loaded document and not to wait for data from the document server.

The *terminate()* call is used to tell the viewer to terminate (when the Harmony Session is terminated by the user), *setLanguage()* is used to tell the viewer the current language. The viewer then changes its user interface to the new language; the version of the document in the new language is loaded by the normal *load()/browse()* mechanism.

The function *getPos()* is called by the Session Manager to get the current position within the document (in the Film Player, that is the current frame); the Session Manager stores this position with the history list; when the user jumps back to this document using the Session Manager's history functions, this position is sent to the viewer as destination anchor with the *browse()* call.

The remaining functions are rather straightforward: *map()* tells the viewer to map its window, *unmap()* to unmap its window, *raise()* to raise its window (i.e. make the window the topmost window), *lower()* to lower its window, *iconify()* to iconify its window and *deiconify()* to deiconify its window.

The communication in the other direction, from the viewer to the Session Manager, is implemented by the *class HgViewerManager*:

```

class HgViewerManager {
public:
    virtual void viewerTerminated(const char* document,
                                  HgViewer* viewer);
    virtual void viewerError(HgViewer* viewer);

    virtual void followLink(const char* source_anchor,
                            HgViewer* viewer);
    virtual void defineSourceAnchor(const char* anchor);
    virtual void defineDestAnchor(const char* anchor);
    virtual void deleteLink(const char* source_anchor,
                             HgViewer* viewer);

    virtual void back(HgViewer* viewer);
    virtual void forward(HgViewer* viewer);
    virtual void history(HgViewer* viewer);
    virtual void hold(RString document,
                     HgViewer* viewer);
};

```

When the viewer calls one of these functions, a remote procedure of the Session Manager is called; all of the functions need the calling viewer as one of the parameters.

By calling *viewerTerminated()* the viewer tells the Session Manager, that it will terminate now (for example, because the user wants to quit the viewer). *viewerError()* tells the Session Manager, that an error condition occurred in the viewer; the (protected) member variable *error_* of *HgViewer* specifies the exact error condition.

The function *followLink()* is called by the viewer to tell the Session Manager to follow the link attached to the source anchor passed as argument. *defineSourceAnchor()* and *defineDestAnchor()* are used to define a new source or destination anchor respectively. The position of the anchor in the document is encoded in the position field of the anchors object description (see 7.2.6. on page 63). Finally, *deleteLink()* is used to delete a link.

The function *history()* causes the Session Manager to open the history browser, *back()* reloads the previous document in the history list (if there is one), *forward()* reloads the next document in the history list, and *hold()* tells the Session Manager, that it should start a new viewer, when the next document of the same type should be loaded, so that the current document remains on the screen.

7.2.4. Implementing the Communication - The Dispatcher

With the InterViews toolkit (see chapter 7.2.7. on page 68) comes a set of classes to handle asynchronous communication. The central class of these is the **class Dispatcher**: IOcallback functions can be linked to file descriptors using the *Dispatcher::link()* function. This file descriptor hides the communication via a socket. Whenever data can be read from or written to this file descriptor, the linked callback function is called, until it is taken off the Dispatcher by calling *Dispatcher::unlink()*. When *Dispatcher::dispatch()* is called, the Dispatcher checks if any I/O-event occurred and calls the linked callback function. When using InterViews *Dispatcher::dispatch()* need not be called explicitly, since it is done implicitly by the Session run-loop.

The Film Player uses the Dispatcher to get its document from the server; the communication itself is managed by the *class Reader*:

```
class Reader {
public:
    Reader(HgIvMovieViewer*);
    ~Reader();

    int port() const;
    void create();
    void abort();
    void getData();

private:
    int readInput(int fd);
};
```

When an object of *class Reader* is created, it creates a port and listens. The number of this port can be asked by calling *Reader::port()* and is returned to the Session Manager.

When the Session Manager tells the viewer to load a new document, the viewer calls *Reader::create()*, which causes the Reader to accept a request from the document server on its port and link its private method *Reader::readInput()* to the file descriptor belonging to this connection. Whenever data can be read from this file descriptor, the Dispatcher calls *Reader::readInput()*. This function just reads the data into an internal buffer and passes them on to the viewer, which in turn appends it to a temporary file; this file is also passed to the MPEG decoder as input file.

Reader::getData() is called by the viewer, when the MPEG-decoder ran out of data, to read new data; it just calls *Reader::readInput()*, which performs a blocking read if no data are available.

By calling *Reader::abort()* the communication to the document server is aborted by closing the corresponding file descriptor and unlink the callback function.

7.2.5. The Frame List

One of the major features of the Film Player is, that it is possible to jump to any frame of the film directly. MPEG does not really support this, since frames may depend on each other and there's no other way to find the start of a frame in the MPEG stream than searching for its picture start code. So the viewer has to take care of the frames by itself. An object of *class Frame* describes one frame:

```
class Frame {
public:
    long pos() const;           // get file position of frame
    int fileOrder() const;     // number of frame in file order
    int dispOrder() const;     // number of frame in display order
    int past() const;          // get past reference frame
    int future() const;        // get future reference frame
};
```

A frame object stores the position of a frame in the MPEG file, its number in both file and display order and its past and future reference frames. Each of this information can be accessed by the appropriate member function. The functions *Frame::past()* and *Frame::future()* return the number of the past or future reference frame in display order respectively, or -1 if there is no such reference frame (so I-frames are frames with past = -1 and future = -1, P-frames are frames with past \neq -1 and future = -1 and B-frames are frames with past \neq 1 and future \neq -1).

All objects of *class Frame* are created by *class FrameList* (which is a friend of *class Anchor*, since *Anchor* does not possess a public constructor). When loading is complete, the viewer creates a frame list. Its constructor parses the MPEG stream for picture start codes, reads the picture header to find the frame type, calculates the reference frames, and builds up a list of all frames in the MPEG stream sorted by their number in display order.

```
class FrameList {
public:
    FrameList(FILE* mpegStream); // constructor
    ~FrameList();                // destructor

    long count() const;          // get number of frames
    Frame* item(long i) const;   // get i-th frame in list
    const Frame* findClosestIFrame(long i) const; // find closest I-frame to i-th frame
};
```

By calling *FrameList::count()*, the total number of frames in the MPEG stream is returned, *FrameList::item(long i)* returns the i-th item of the list (i.e. the i-th frame in display order, since the list is sorted by display order) and *FrameList::findClosestIFrame(long i)* can be used to find the I-frame closest to the i-th frame.

7.2.6. Anchors

Format of Anchors:

Anchors are Hyper-G objects and passed to the viewer as parameter of either the *load()* (source anchors) or the *browse()* (destination anchor) call. The part of the anchor object describing the shape of the anchor is the field *Position*; for anchors in film documents, the format of this field is shown in figure 7.9.

<pre><position> ::= <rect-position> <circle-position> <frame-position> <rect-position> ::= Position=Rect <interpolation> n_k: k_i x₀ y₀ x₁ y₁, ... <circle-position> ::= Position=Circle <interpolation> n_k: k_i x₀ y₀ r, ... <frame-position> ::= Position=n <interpolation> ::= Linear Spline</pre>

Figure 7.9: Format of Position Field

For rectangular anchors *<interpolation>* defines the interpolation algorithm used to calculate the shape of the anchor for non-key frames; n_k is the number of key frames; k_i is the number of the key frame (in display order), (x_0, y_0) is the lower left corner and (x_1, y_1) the upper right corner of the rectangle in the key frame, where x_0, y_0, x_1, y_1 are normalised coordinates in $[0;1]$; the origin is the lower left corner of the frame.

Anchors with circular shape are analogous to rectangular ones: here (x_0, y_0) defines the center of the circle, and r the radius, normalised in x-direction.

<frame-position> are "pseudo-destination anchors" used with the *getPos()* call. Whenever the Session Manager tells a viewer to load a new document, the Session Manager asks the viewer for its current position (i.e. frame) in the (old) document by calling *getPos()*. When this document is reloaded later by some history function, the Session Manger sends the viewer the argument got by *getPos()* as destination anchor, so that the viewer can jump to the previous position. These special destination anchors are not stored in the database; they can be identified by the value *NoMark* in the field *Function*. The *Position* field contains only the number of the frame (in display order) to jump to.

Implementation of anchors

There is an abstract base *class Anchor* which defines the functionality of anchors, without specifying its shape:

```
class Anchor {
public:
    enum InterpolationMethod {
        LINEAR,
        SPLINE
    };

    Anchor(Object&);
    virtual ~Anchor();

    boolean ok() const;
    virtual int count() const = 0;

    // drawing
    virtual void draw(XDisplay*, XDrawable, GC,
                     int frame,                // current frame
                     float sx, float sy) = 0; // scaling factors
    virtual void drawKey(XDisplay*, XDrawable, GC,
                        int frame,
                        float sx, float sy) = 0;
    virtual void drawMark(XDisplay*, XDrawable, GC,
                          float sx, float sy) = 0;

    // "utilities"
    virtual boolean hit(int frame, float x, float y) = 0;
    virtual int keyFrame(int frame) = 0;
    virtual RString position() const = 0;

    // marking
    virtual void startMarking(int frame, float x, float y) = 0;
    virtual void updateMarking(int frame, float x, float y) = 0;
    virtual void marking(int frame, float x, float y) = 0;
```



```

    virtual void stopMarking(int frame, float x, float y) = 0;

    InterpolationMethod interpolation() const;
    int startFrame() const;
    int stopFrame() const;
    boolean seen() const;
    boolean visible(int frame) const;

protected:
    friend class AnchorList;

    virtual void interpolate() = 0;
    virtual void interpolateLinear() = 0;
    virtual void interpolateSpline() = 0;

```

Normally an anchor is constructed by its Hyper-G object description. *Anchor::ok()* returns if the construction was successful (i.e. if the parameter passed to the constructor was a valid Hyper-G anchor description). *Anchor::count()* returns the number of key frames.

Anchor::draw() tells the anchor to draw its region for the specified frame on the specified Drawable. *Anchor::drawKey()* tells the anchor to draw the region of its key frame (if the specified frame is a key frame). *Anchor::drawMark()* is used for rubberbanding during defining a new anchor.

Anchor::hit() returns if the specified point is inside the anchor's region. *Anchor::keyFrame(int frame)* returns the number of the frame-th frame in the list of key frames, or -1 if it is no key frame. *Anchor::position()* returns the position field of the anchors object description.

Anchor::startMarking(), *Anchor::updateMarking()*, *Anchor::marking()* and *Anchor::stopMarking()* are called to define a new anchor. They are called when the user presses the mouse button, drags the mouse, and releases the mouse button respectively.

Anchor::interpolation() returns the interpolation algorithm used by the anchor (either linear or spline), *Anchor::startFrame()* and *Anchor::stopFrame()* return the number of the first and last key frame (in display order), *Anchor::seen()* returns if the anchor's destination has already be seen and *Anchor::visible()* returns if the anchor is visible in the specified frame.

The private members *Anchor::interpolate()*, *Anchor::interpolateLinear()*, and *Anchor::interpolateSpline()* are called to interpolate the shape of the anchor in non-key frames.

class *RectAnchor* is derived from *Anchor*. It implements the pure virtual methods of *Anchor* for rectangular shaped anchors:

```

class RectAnchor : public Anchor {
public:
    RectAnchor(Object&);
    virtual ~RectAnchor();

    virtual void draw(XDisplay*, XDrawable, GC,
                    int frame,
                    float sx, float sy);

```

```

virtual void drawKey(XDisplay*, XDrawable, GC,
                    int frame,
                    float sx, float sy);
virtual void drawMark(XDisplay*, XDrawable, GC,
                    float sx, float sy);

virtual int count() const;
virtual boolean hit(int frame, float x, float y);
virtual RString position() const;

virtual void startMarking(int frame, float x, float y);
virtual void updateMarking(int frame, float x, float y);
virtual void marking(int frame, float x, float y);
virtual void stopMarking(int frame, float x, float y);

private:
    virtual void interpolate();
    virtual void interpolateLinear();
    virtual void interpolateSpline();
};

```

An analogous *class CircleAnchor* exists for circular shaped anchors.

The *class AnchorList* is the central class for the management of anchors. It handles source anchors as well as the destination anchor and the definition of new anchors.

```

class AnchorList {
public:
    AnchorList(HgIvMoviePlayer*);
    ~AnchorList();

    void setSource(const char*);
    void setDest(const char*);
    void clearDest();

    void setDrawable(Window*);

    void updateDraw();
    void draw() const;
    boolean selected() const;
    boolean mark() const;

    void interpolation(Anchor::InterpolationMethod);

    void follow();
    void follow(Coord x, Coord y) const;
    void next();
    void selectNew(Coord x, Coord y);
    void selectNext(Coord x, Coord y);

    void defineSource();
    void defineDest();
    void useDefault();
    void deleteLink();

    void startMarking(float x, float y);
    void updateMarking(float x, float y);

```

```

void marking(float x, float y);
void stopMarking(float x, float y);

// colors
void getAnchorColor(int&, int&, int&) const;
void getSeenColor(int&, int&, int&) const;
void getSelectedColor(int&, int&, int&) const;
void getDestColor(int&, int&, int&) const;
void getMarkColor(int&, int&, int&) const;

void setAnchorColor(int, int, int);
void setSeenColor(int, int, int);
void setSelectedColor(int, int, int);
void setDestColor(int, int, int);
void setMarkColor(int, int, int);
};

```

AnchorList::setSource() is used to set the source anchors got from *load()*, *AnchorList::setDest()* is used to set the destination anchor got from *browse()*. *AnchorList::clearDest()* clears the current destination anchor.

AnchorList::draw() draws all visible anchors (source anchors as well as the destination anchor and a newly marked anchor), *AnchorList::updateDraw()* updates the list of visible source anchors (according to the current frame of the film player) and then draws all visible anchors.

AnchorList::interpolation() sets the interpolation algorithm for the newly defined anchor.

AnchorList::follow() activates the selected source anchor (if there is one), *AnchorList::follow(Coord x, Coord y)* selects the first source anchor found, which encloses the point (x, y), and activates it.

AnchorList::selectNew(Coord x, Coord y) and *AnchorList::selectNext(Coord x, Coord y)* select the first or the next source anchor found, which encloses the point (x, y). *AnchorList::next()* just selects the next visible source anchor.

AnchorList::startMarking(), *AnchorList::updateMarking()*, *AnchorList::marking()*, and *AnchorList::stopMarking()* are called when the user defines a new anchor; they are passed to the newly defined anchor object.

AnchorList::defineSource() and *AnchorList::defineDest()* are called to send the position of a newly marked anchor to the Session Manger as source or destination anchor respectively. It is the Session Manager's task to insert the anchor into the database.

AnchorList::useDefault() tells the Session Manager to use the whole film as destination anchor.

AnchorList::deleteLink() tells the Session Manager to delete the selected source anchor, as well as the attached link and the attached destination anchor.

7.2.7. The User Interface - InterViews

The InterViews toolkit [Linton89] is based upon the X11 library [Nye88, Nye89] and provides a set of C++ classes to build user interfaces. It is public domain and available by anonymous ftp from interviews.stanford.edu/pub/3.1.tar.Z.

Session

Each InterViews application must create exactly one object of class *Session*; the one and only object of this class is accessible by the *Session::instance()* member function. The *Session::run()* member function provides the main run loop: X events (for example generated by user interactions when moving the mouse, pressing mouse buttons or keys) are read and appropriate functions are called until the Session is ended by calling *Session::quit()*. The Session run loop also calls the Dispatcher (see chapter 7.2.4. on page 61) to handle I/O-events.

Window

Another important InterViews class is the *Window* class; there are some derived classes for various purposes, the most important of them is the *ApplicationWindow*. An *ApplicationWindow* gets a *Glyph* (see below) and is mapped to the screen when *Window::map()* is called. A shortcut for mapping a window and then starting the session run loop is provided by the member *Session::run_window()*.

Glyph

Glyphs are these InterViews' objects which actually do drawing. They can be combined to build a directed acyclic graph. The main member functions of class *Glyph* are:

Glyph::request(Requisition&) asks the *Glyph* about its size (which in InterViews is called *Requisition*, which in turn consists of a *Requirement* for each dimension. A *Requirement* defines a natural ("default") size of the *Glyph*, as well as its maximal and its minimal size for the given dimension).

Glyph::allocate(Canvas, const Allocation&, Extension&)* tells the *Glyph*, at which *Allocation* (i.e. which region of the *Canvas*) it may draw itself; the *Canvas* defines a two-dimensional surface to which a group of *Glyphs* are attached and onto which they may draw. By setting an *Extension* the *Glyph* tells the application, onto which part of its *Allocation* the *Glyph* really draws; the *Extension* is used by InterViews to decide if the *Glyph* has to be redrawn, when the window (or part of it) has been damaged.

Glyph::draw(Canvas, Allocation&)* tells the *Glyph* to draw itself at the given *Allocation*. A draw must not occur before the *Glyph* is told its *Allocation* by an *allocate* call.

A special kind of *Glyph* are *MonoGlyphs*. A *MonoGlyph* can be wrapped around another *Glyph* (called its body). They add some special functionality or behaviour to their body.

A simple example of a *MonoGlyph* is the *Background*. It just draws some background colour before it draws its body on this background. More complicated *MonoGlyphs* are the *Patch* and the *InputHandler*: A *Patch* stores the *Allocation* of its body; it defines the member

functions *Patch::reallocate()* which calls the body's *allocate()* with the stored *Allocation* and *Patch::redraw()* which causes the body to be redrawn.

An ***InputHandler*** adds input handling to its body. It defines the member functions *InputHandler::press()* and *InputHandler::release()*, which are called when a mouse button is pressed or released respectively, *InputHandler::drag()* and *InputHandler::move()*, which are called, when the mouse cursor is moved in the body's *Allocation*, *InputHandler::double_click()*, which is called when a mouse button is double clicked, and finally *InputHandler::keystroke()*, which is called when a key has been pressed.

WidgetKit

A set of elements, such as menus, push buttons, and scroll bars, needed to build user interfaces are provided by InterViews in the class *WidgetKit*. For example *Button* WidgetKit::push_button(char* label, Action* action)* creates a push button with a given label; when the button is pressed by the user, InterViews calls the provided callback function *action*, so the application can react to the users action. Menus and scroll bars are dealt with similarly.

The *WidgetKit* also provides some more simple *Glyphs* such as *WidgetKit::inset_frame(Glyph*)* or *WidgetKit::outset_frame(Glyph*)* to draw a "3-dimensional" border around a *Glyph*, or *WidgetKit::background()* and *WidgetKit::foreground()* which return the current background and foreground colour respectively.

LayoutKit

The *LayoutKit* provides *Glyphs* to structure other *Glyphs* or to change their layout, such as *LayoutKit::margin()* to draw margin around a *Glyph*, *LayoutKit::hspace()* and *LayoutKit::vspace()* to draw a horizontal or vertical space respectively, and *LayoutKit::hbox()* and *LayoutKit::vbox()* to group *Glyphs* in a horizontal or vertical box respectively.

Style

As usual in X applications, user-customizable attributes, such as foreground and background colour, or the size and position of the window, are provided by X resources. InterViews supports these in the class *Style*. When creating the *Session*, one must specify the base name of the resource class. *Styles* can be nested hierarchically, by creating a new *Style* with an existing one as parent.

Style::attribute() can be used to set ("hard-code") attributes, *Style::find_attribute()* or *Style::value_is_on()* can be used to read the X resources matching the *Style*'s resource class name.

7.2.8. Embedding the MPEG decoder into InterViews

InterViews was used to build a user interface around the MPEG decoder. InterViews' run loop (*Session::run()*) could not be used for this purpose, since it provides no mechanism to

call a function, when the event queue is empty. This is necessary, since when there are no events in the queue, the player should decode and display frames. Therefore *class HgIvMoviePlayer* (see chapter 7.2.9. on page 70) defines its own run loop:

```
void HgIvMoviePlayer::run() {
    Session* session = Session::instance();
    Dispatcher& dispatcher = Dispatcher::instance();
    long sec = 0;
    long usec = 0;
    Event event;
    do {
        if (state == MovieLoading || (state == MoviePlaying && live_)) {
            dispatcher.dispatch(sec, usec);
            if (session->pending()) {
                session->read(event);
                event.handle();
            }
            else {
                callParser();
            }
        }
        else {
            session->read(event);
            event.handle();
        }
    } while (!session->quit());
}
```

During loading, if the live option is turned on, and during playing, the *Session::pending()* call is used to check, if there are any events in the applications event queue. If there are events, the first one is read (*Session::read()*) and handled (*Event::handle()*), if there are none, the MPEG parser is invoked (*HgIvMoviePlayer::callParser()*) to decode some part of the MPEG stream and display a frame. Additionally the *Dispatcher* is activated by calling *Dispatcher::dispatch()* to handle the communications to both the document server and the Session Manager.

When not loading or playing (i.e. when the film is paused, or no MPEG stream has been opened yet), *Session::read()* is called directly; this blocks the application until an event occurs; the *Dispatcher* is called periodically by *Session::read()*, so there is no need to call it explicitly.

InterViews has also to create, map, and place the window into which the decoder draws the frames. This is handled by *class X_Context*, derived from *Glyph*. *X_Context* maps an *X_Window* (which is just a very simple subclass of *Window*, without any special functionality) at its *Allocation*. The window-id of this window is then passed to the decoder by calling *HgMpeg::initXDisplay()*.

7.2.9. The Viewer Class

The central class of the Film Player is *class HgIvMoviePlayer* (the "Hg" stands for Hyper-G, the "Iv" for InterViews). It combines all of the above classes and really implements the player's functionality.

```

class HgIvMoviePlayer : public HgMpeg, public HgViewer {
public:

    // internal state of the player
    enum MovieState {
        MovieInvalid,          // no movie loaded
        MovieLoading,          // just loading movie
        MovieLoaded,           // loading complete occurred
        MoviePlaying,          // just playing movie
        MoviePaused,           // movie paused (by user)
        MovieDone               // playing is done; last frame shown
    };

    HgIvMoviePlayer (HgViewerManager*, boolean);
    virtual ~HgIvMoviePlayer();

    // my runloop
    void run();

    // from HgViewer
    virtual void load(const char* doc,
                     const char* anchors,
                     const char* info = nil);
    virtual void browse(const char* dest);
    virtual void terminate();
    virtual void setLanguage(int);
    virtual int port() const;
    virtual void getPos(RString&);
    virtual void iconify();
    virtual void deiconify();
    virtual void map();
    virtual void unmap();
    virtual void moveto(float, float);
    virtual void resize(float, float);
    virtual void raise();
    virtual void lower();

    // from HgMpeg
    virtual void initXDisplay(Window* win);
    virtual void drawSpecial() const;
    virtual void changeDither(int);
    virtual void waitForData();
    virtual void error(const char*);

    // functionality
    void play();
    void pause();
    void step();
    void backstep();
    void rewind();
    void wind();
    void gotoIFrame(int frame);
    void gotoFrame(int frame);
    void quit();
    void abort();
    void openFile(const char* filename);
    void saveAs(const char* filename);

    // miscellaneous

```

```

void redrawFrame() const;
int aborted() const;
void toggleLive();
void toggleShowAnchors();
void changeSize(HgIvMovieViewer::MovieSize);

private:
    void callParser(); // call MPEG parser
};

```

The class maintains an internal state of type *HgIvFilmPlayer::MovieState*, which can be *MovieInvalid* (when some error occurred), *MovieLoading* (during loading), *MovieLoaded* (when loading was completed), *MoviePlaying* (when the movie is playing), *MoviePausing* (when the movie is paused) or *MovieDone* (when the end of the movie has been reached). This state also affects the run loop of this class (see chapter 7.2.8. on page 69), which calls the MPEG decoder by calling *HgIvMoviePlayer::callParser()* when the state is either *MoviePlaying* or *MovieLoading*.

Additionally to overloading and implementing methods inherited from *HgViewer* and *HgMpeg*, this class also implements the functionality. A video recorder provides the obvious user metaphor for the Film Player; this is also reflected by the methods implementing the functionality:

HgIvMoviePlayer::play() starts the playing of the movie, *HgIvMoviePlayer::pause()* pauses the playing of the movie, *HgIvMoviePlayer::stop()* stops the playing of the movie. These functions essentially just set the state of the player to the appropriate value.

HgIvMoviePlayer::step() can be used to step through the movie picture-wise. It calls *HgMpeg::decode()* until a picture has been displayed.

HgIvMoviePlayer::gotoFrame(int frame) jumps to the frame-th picture (in display order) of the film. The implementation of this function is more complicated, since in MPEG frames may depend on other ones: when loading is complete, the viewer creates an object of *class FrameList* (see 7.2.5. on page 62), which is then used to determine the reference frames of the target frame and their position in the MPEG stream. Then the decoder is called to decode the reference frames (and recursively their reference frames, if there are any) and finally decodes the target frame and displays it. This function can be rather time consuming, as maybe a larger number of reference frames has to be decoded too. Therefore the function *HgIvMoviePlayer::gotoIFrame(int frame)* has been implemented. This function jumps to the I-frame closest to the frame-th picture. Since I-frames do not depend on other frames, this frame can be decoded and displayed immediately. The scrollbar of the viewer uses *HgIvMoviePlayer::gotoIFrame()* for performance reasons, but it could be replaced by *HgIvMoviePlayer::gotoFrame()* any time.

The other functions *HgIvMoviePlayer::backstep()*, *HgIvMoviePlayer::rewind()* and *HgIvMoviePlayer::wind()* just use *HgIvMoviePlayer::gotoFrame()* to jump to the previous frame, or to the first frame and the last frame of the destination anchor (or the whole film, if there is no destination anchor) respectively.

HgIvMoviePlayer::quit() quits the viewer (after telling the Session Manager), and *HgIvMoviePlayer::abort()* is used to abort loading: it stops the reader process, searches for the start of the last frame already loaded and truncates the MPEG stream at that point. Additionally the viewer sets its variable *error_* (a protected member of the base *class*

HgViewer) to *INCOMPLETELOAD* and calls *HgViewerManager::viewerError()*. So the Session Manager knows that loading was incomplete and does not send a reload, but a real load, when the user wants to see the document again.

Calling *HgIvMoviePlayer::saveAs(const char* filename)* saves the MPEG stream. *HgIvMoviePlayer::openFile(const char* filename)* is used to play a local MPEG file. This is implemented by directly calling the functions *load()* and *browse()*, which are normally called by the Session Manager.

Finally there are a number of member functions to change the display size (half, normal, or double), the turn on or off the playing of the film during loading, or to turn on or off the anchors, whose meaning and implementation is straightforward.

8. Summary

Moving pictures have become an important part of our daily life, for entertainment as well as business. Television's advance from analogue to digital technology further increases its possibilities.

On the other hand new communication technologies are arising, which, in some time, will surely be as widespread and natural as television is today. Every household will be connected to a global communication network. But as information systems will be used by more and more people (and not only experts) it will become essentially that these systems are easy to use. Hypermedia systems could meet these expectations.

Hyper-G is the first second-generation hypermedia system, which is flexible enough to be of use for a wide range of applications. Harmony is the Unix/X11 client for Hyper-G. It consists of the Session Manger, which allows users to navigate through the information space, and a native viewer for each type of document, which currently are text, image, film, audio, scene and postscript. It is the viewer's responsibility to display a document.

The Harmony Film Player fully integrates digital video into the hyperlink structure of Hyper-G/Harmony. It supports source and destination anchors in film documents as well as it provides the functionality to define new anchors. The most important features of the Film Player are:

- The film can be played *live* during loading.
- The Film Player provides a *VCR-like user interface* and a *scrollbar*, allowing the user to pause the film or to jump to any picture of the film.
- The Film Player allows *synchronised playback* of the film, with any user chosen frame rate.
- By defining rectangles in key frames, any part of the film (both spatial and temporal) can be marked either as source or destination anchor of a *hyperlink*.

The Harmony Film Player is far from being complete. It was a first step to show the possibilities of digital video in hypermedia systems. Important things to come in the future include:

- *MPEG-2* is now a new international standard, which is especially designed to suit digital television; surely the Film Player will support MPEG-2 in some time.
- Other existing formats for digital video should be integrated into the Film Player, namely *Quicktime* and *AVI*, which are already widely used, the first on Apple platforms, the later on Microsoft Windows platforms.

- Up to now the Film Player does not support *audio* (one can say it is still in the silent era). Audio immediately introduces the problem of absolute synchronisation of video and audio (which makes it difficult to combine the Harmony Film Player and the Harmony Audio Player, since they are independent processes and cannot be synchronised easily).
- Decoding of digital video needs a lot of time, especially, when also audio should be decoded and played, synchronously and in real time. Therefore Film Player should let *MPEG hardware*, if present, do the decoding (and displaying) in the future.

Appendix A: The Harmony Colormap

The Harmony Colormap, which is used by all Harmony processes, contains 148 colours. These are: 6 primary colours (red, green, blue, cyan, magenta and yellow), 16 grayscales (black, #111111, #222222, ..., #EEEEEE, white), and 128 colours equally spaced in $YCbCr$ colour space, with 8 values of Y (16, 48, 80, 112, 144, 176, 208, 240) and 4 values of C_b and C_r respectively (32, 96, 160, 224). Note that 2 of these colours are double in RGB colour space, making totally 148 unique colours.

The colours are sorted and allocated by priority, so that Harmony gets colours across the whole colour space, even if it cannot allocate all desired colours. If Harmony is not able to allocate a colour, it takes the closest colour to be found in the colormap.

Table A.1: The Harmony Colormap

	RGB	$YCbCr$
0	(255/255/255)	white
1	(0/0/0)	black
2	(0/147/255)	(112/224/32)
3	(255/108/0)	(144/32/224)
4	(41/255/5)	(176/32/32)
5	(214/0/250)	(80/224/224)
6	(119/119/119)	gray
7	(150/0/0)	(16/96/224)
8	(195/255/69)	(240/32/96)
9	(105/255/255)	(240/160/32)
10	(150/0/0)	(16/32/224)
11	(255/150/151)	(208/96/224)
12	(0/127/0)	(48/96/32)
13	(0/137/136)	(80/160/32)
14	(41/255/119)	(176/96/32)
15	(214/0/136)	(80/160/224)
16	(220/120/255)	(176/224/160)
17	(92/14/104)	(48/160/160)
18	(255/0/0)	red
19	(0/255/255)	cyan
20	(99/133/255)	(144/224/96)
21	(156/122/0)	(112/32/160)
22	(92/0/218)	(48/224/160)
23	(195/255/183)	(240/96/96)
24	(131/231/5)	(176/32/96)
25	(255/255/0)	yellow
26	(131/209/119)	(176/96/96)
27	(0/73/72)	(16/160/32)
28	(255/86/87)	(144/96/224)

	RGB	$YCbCr$
29	(67/145/55)	(112/96/96)
30	(255/64/200)	(144/160/224)
31	(35/69/250)	(80/224/96)
32	(163/197/255)	(208/224/96)
33	(92/58/0)	(48/32/160)
34	(0/191/55)	(112/96/32)
35	(156/56/255)	(112/224/160)
36	(255/204/69)	(240/32/224)
37	(9/201/200)	(144/160/32)
38	(188/132/87)	(144/96/160)
39	(220/186/5)	(176/32/160)
40	(170/170/170)	gray
41	(156/78/168)	(112/160/160)
42	(105/255/69)	(240/32/32)
43	(105/255/183)	(240/96/32)
44	(67/123/168)	(112/160/96)
45	(255/184/255)	(240/224/160)
46	(0/0/255)	blue
47	(68/68/68)	gray
48	(150/0/72)	(16/160/224)
49	(150/0/186)	(16/224/224)
50	(255/228/183)	(240/96/160)
51	(41/211/255)	(176/224/32)
52	(214/44/0)	(80/32/224)
53	(195/251/255)	(240/160/96)
54	(60/4/0)	(16/96/160)
55	(67/167/0)	(112/32/96)
56	(0/213/0)	(112/32/32)
57	(255/42/255)	(144/224/224)

	RGB	YC_bC_r
58	(0/71/0)	(16/32/96)
59	(204/204/204)	gray
60	(99/177/87)	(144/96/96)
61	(35/91/136)	(80/160/96)
62	(99/155/200)	(144/160/96)
63	(124/46/136)	(80/160/160)
64	(188/110/200)	(144/160/160)
65	(156/100/55)	(112/96/160)
66	(34/34/34)	gray
67	(0/83/218)	(48/224/32)
68	(220/164/119)	(176/96/160)
69	(163/241/151)	(208/96/96)
70	(60/0/72)	(16/160/160)
71	(9/223/87)	(144/96/32)
72	(246/32/168)	(112/160/224)
73	(246/54/55)	(112/96/224)
74	(0/27/72)	(16/160/96)
75	(255/250/69)	(240/32/160)
76	(124/24/250)	(80/224/160)
77	(3/37/218)	(48/224/96)
78	(252/196/151)	(208/96/160)
79	(99/199/0)	(144/32/96)
80	(163/255/37)	(208/32/96)
81	(73/255/37)	(208/32/32)
82	(73/255/151)	(208/96/32)
83	(0/105/104)	(48/160/32)
84	(0/169/168)	(112/160/32)
85	(67/101/255)	(112/224/96)
86	(131/165/255)	(176/224/96)
87	(60/0/186)	(16/224/160)
88	(188/88/255)	(144/224/160)
89	(182/0/218)	(48/224/224)
90	(182/0/104)	(48/160/224)
91	(255/96/232)	(176/160/224)
92	(255/118/119)	(176/96/224)
93	(255/172/37)	(208/32/224)
94	(124/90/0)	(80/32/160)
95	(188/154/0)	(144/32/160)
96	(35/113/23)	(80/96/96)
97	(0/255/0)	green
98	(255/0/255)	magenta
99	(0/159/23)	(80/96/32)
100	(255/138/255)	(240/224/224)
101	(0/51/186)	(16/224/32)
102	(3/59/104)	(48/160/96)

	RGB	YC_bC_r
103	(255/182/183)	(240/96/224)
104	(252/218/37)	(208/32/160)
105	(73/243/255)	(208/224/32)
106	(182/12/0)	(48/32/224)
107	(9/179/255)	(144/224/32)
108	(246/76/0)	(112/32/224)
109	(0/115/250)	(80/224/32)
110	(255/140/5)	(176/32/224)
111	(35/135/0)	(80/32/96)
112	(0/181/0)	(80/32/32)
113	(41/233/232)	(176/160/32)
114	(131/187/232)	(176/160/96)
115	(220/142/232)	(176/160/160)
116	(255/74/255)	(176/224/224)
117	(214/22/23)	(80/96/224)
118	(124/68/23)	(80/96/160)
119	(17/17/17)	gray
120	(51/51/51)	gray
121	(85/85/85)	gray
122	(102/102/102)	gray
123	(136/136/136)	gray
124	(153/153/153)	gray
125	(187/187/187)	gray
126	(221/221/221)	gray
127	(238/238/238)	gray
128	(255/106/255)	(208/224/224)
129	(3/103/0)	(48/32/96)
130	(0/49/0)	(16/96/96)
131	(0/149/0)	(48/32/32)
132	(163/219/255)	(208/160/96)
133	(195/229/255)	(240/224/96)
134	(255/206/255)	(240/160/160)
135	(255/160/255)	(240/160/224)
136	(60/26/0)	(16/32/160)
137	(92/36/0)	(48/96/160)
138	(9/245/0)	(144/32/32)
139	(246/10/255)	(112/224/224)
140	(73/255/255)	(208/160/32)
141	(182/0/0)	(48/96/224)
142	(3/81/0)	(48/96/96)
143	(252/174/255)	(208/160/160)
144	(0/117/0)	(16/32/32)
145	(255/128/255)	(208/160/224)
146	(0/95/0)	(16/96/32)
147	(252/152/255)	(208/224/160)

References

- [Alberti92] B. ALBERTI, F. ANKLESARIA, P. LINDNER, M. MCCAHERN, D. TORREY: *The Internet Gopher Protocol: A Distributed Document Search and Retrieval Protocol*. March 1992. Available by anonymous ftp from boombox.micro.umn.edu: /pub/gopher/gopher_protocol.
- [Andrews93] K. ANDREWS, J. FASCHINGBAUER, M. GAISBAUER, F. KAPPE, H. MAURER M. PICHLER, J. SCHIPFLINGER: *Hyper-G: A New Tool for Distributed Hypermedia*. Proc. Distributed Multimedia Systems and Applications, Honolulu 1994, pp 209-214.
- [Andrews94a] KEITH ANDREWS, FRANK KAPPE: *Soaring through hyperspace: A snapshot of Hyper-G and its Hamony client*. Proc. of Eurographics Symposium and Workshop on Multimedia: Multimedia/Hypermedia in Open Distributed Environments, Graz, June 1994.
- [Andrews94b] KEITH ANDREWS, FRANK KAPPE., HERMANN MAURER, KLAUS SCHMARANZ: *On Second Generation Hypermedia Systems*. Journal of Universal Computer Science J.UCS Vol 0, 0, online via Hyper-G (host hyperg.iicm.tu-graz.ac.at) or WWW (http://www.iicm.tu-graz.ac.at/Cjucs_root).
- [Apple93] APPLE COMPUTER, INC.: *Inside Macintosh, Volume 4: Quicktime*; Addison-Wesley 1993.
- [Berk91] EMILY BERK, JOSEPH DEVLIN (ED.): *Hypertext/Hypermedia Handbook*; Intertext Publications, McGraw-Hill Publishing Company 1991.
- [Berners92] T. BERNERS-LEE, R. CAILLIAU, J. GROFF., B. POLLERMANN: *World-Wide-Web: The Information Universe*. Electronic Networking: Research, Applications and Policy, 2(1):52-58, Spring 1992.
- [Color94] The Color-Space FAQ; available by anonymous ftp (for example) from rtfm.mit.edu:/pub/usenet/comp.graphics/Color_space_FAQ.
- [Gonzales92] R. C. GONZALES , R. E. WOODS: *Digital Image Processing*; Addison-Wesley 1992.
- [Günther92] CARSTEN GÜNTHER: *Andy's Erbe - Apples Multimedia-Erweiterung QuickTime*; c't Oktober 1992, p 186-194.
- [Hill90] FRANCIS S. HILL, JR.: *Computer Graphics*; Macmilan Publishing Company, New York 1990.

- [ISO10918] ISO/IEC 10918 (JPEG); *Digital Compression and Coding of Continuous-tone Still Images*.
- [ISO11172] ISO/IEC-11172 (MPEG): *Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 MBit/s*.
- [ISO13818] ISO/IEC 13818 (MPEG-2): *Coding of Moving Pictures and Associated Audio Information*.
- [ITU90] ITU/CCITT: Recommendation H.261: *Video Codec for Audiovisual Services at px64 kbit/s*. Geneva 1990.
- [Kappe91] FRANK KAPPE: *Aspects of a Modern Multi-Media Information System*; PhD thesis, Graz University of Technology, Austria, June 1991. Also available as IIG Report 308, IIG, Graz University of Technology, Austria, June 1991 and by anonymous ftp from ftp.iicm.tu-graz.ac.at:/pub/Hyper-G/doc.
- [Kappe93] FRANK KAPPE: *Hyper-G: A Distributed Hypermedia System*. in Leiner B. (editor): Proc. INET '93, San Francisco, California, pp DCC-1 - DCC-9, Internet Society, August 1993.
- [LeGall91] DIDIER LE GALL: *MPEG: A Video Compression Standard for Multimedia Applications*; Communications of the ACM Vol. 34(4):47-58, April 1991.
- [Lennon94] JENNIFER LENNON., HERMANN MAURER: *You Believe You Know What Multimedia is? And What Internet Will Do For You? Well... Think Again!*; Journal of Universal Computer Science J.UCS Vol 0, 0, online via Hyper-G (host hyperg.iicm.tu-graz.ac.at) or WWW (<http://www.iicm.tu-graz.ac.at>).
- [Linton89] MARK A. LINTON, JOHN M. VLISSIDES, PAUL R. CALDER: *Composing User Interfaces with InterViews*. IEEE Computer Vol. 22(2):8-22, Feb. 1989.
- [Maurer92] HERMANN MAURER: *Why Hypermedia Systems are Important*. Proc. ICCAL 92, Wolfville, LNCS 602, Springer Heidelberg/New York (1992), pp 1-15.
- [Maurer94] HERMANN MAURER, KLAUS SCHMARANZ: *J.UCS - The Next Generation in Electronic Publishing*. J.UCS 0, 0 1994; online via Hyper-G (host hyperg.iicm.tu-graz.ac.at) or WWW (<http://www.iicm.tu-graz.ac.at/Cjucs>).
- [Murray94] J. D. MURRAY, W. VAN RYPER: *Encyclopedia of Graphics File Formats*. O'Reilly&Associates, Inc., 1994.
- [Nielson90] JAKOB NIELSEN: *Hypertext & Hypermedia*. Academic Press 1990.
- [Nye89] ADRIAN NYE, O'Reilly & Associates, Inc: *XLIB Programming Manual (for Version 11) Volume One*. 1989.
- [Nye88] ADRIAN NYE, O'Reilly & Associates, Inc: *XLIB Reference Manual (for Version 11), Volume Two*. 1988.
- [Patel93] K. PATEL, B. C. SMITH, L. A. ROWE: *Performance of a Software MPEG Video Decoder*. available by anonymous ftp from: mm-ftp.cs.berkeley.edu:/pub/multimedia/papers/Mpeg93.ps.Z.

- [Pins91] MARKUS PINS: *Extensions of the Color-Cell-Compression*. in Proc. Computer Animation '91, Geneva, Switzerland, pp 241-251, Springer 1991.
- [Pöpsel94] JOSEF PÖPSEL: *Multimediale Klippen - Format von RIFF- und AVI-Dateien*. c't November 1994, pp 327-332.
- [Rowe94] L. A. ROWE, K. D. PATEL, B. C. SMITH, K. LIU: *MPEG Video in Software: Representation, Transmission, and Playback*. High Speed Networking and Multimedia Computing, IS&T/SPIE Symp. on Elec. Imagaging Sci. & Tech., San Jose, CA, Februray 1994; also available by anonymous ftp from: mm-ftp.cs.berkeley.edu: /pub/multimedia/papers/CMPEG-SPIE94.ps.Z.
- [Stein91] RICHARD M. STEIN: *Browsing Through Terabytes - Wide-area Information Servers Open a New Frontier in Personal and Corporate Information Services*. Byte, 16(5):157-164, May 1991.
- [Stroustrup91] BJAREN STROUSTRUP: *The C++ Programming Language*. 2nd edition, Addison Wesley 1991.
- [Wallace91] G. K. WALLACE.: *The JPEG Still Picture Compression Standard*. Communications of the ACM Vol. 34(4):31-44, April 1991.
- [Welch84] TERRY A. WELCH: *A Technique for High-Performance Data Compression*. IEEE Computer Vol. 17(6):8-19, June 1984.

