IDIAP COMMUNICATION

A video package for Torch

Julien Tiphaigne Dr. Sebastien Marcel

IDIAP–Com 04-02

June 2004

Dalle Molle Institute for Perceptual Artificial Intelligence • P.O.Box 592 • Martigny • Valais • Switzerland

phone +41 − 27 − 721 77 11
fax +41 − 27 − 721 77 12
e-mail secretariat@idiap.ch
internet http://www.idiap.ch

Acknowledgements

Firstly, I would like to thank my Supervisor, Dr. Sebastien Marcel, for his useful suggestions and IDIAP for its welcome. He was always available and able to answer to my many questions.

I also thanks Mr. Pierre-Marie Allioux and ENIC who allowed me to do this internship. All what I learned during the last five years, and especially with him, enable me to complete my work.

I would also like to thank all the rest of researchers, administrative and system staff of IDIAP. Everybody help me when it was necessary.

Thanks to Switzerland and Canton du Valais for allowing me to work in their country.

Finally, I thanks the authors of Torch, FFmpeg and Avilib for their works.

Contents

Introduction

My internship took place at IDIAP, a Swiss research institute specialized in speech, computer vision and machine learning. I was integrated into the computer vision group to develop a video coding/decoding C++ library.

IDIAP researchers use several tools to experiment their algorithms. Of course, they do not use the same if they work on videos or on speech. However, they have developed a very powerful tool for all mathematics and statistics operations which is used by all and named Torch. This library works with some specialized packages, one for statistics, one for neural networks, one for speech tools... Currently, there is no package for video and researchers have to convert a video file into several pictures in order to be able to analyze them.

In a first part, I will present IDIAP Research Institute and the current tools used by researchers.

In a second part, I will introduce video technologies, starting with a basic introduction, a presentation of the AVI format and finishing with a description of the mpeg4 compression method. Then, I will do an overview of available open source libraries to work on video, and especially FFmpeg.

Finally, I will present my proposed video package, with details about the source code and a commented example.

Chapter 1

Presentation

1.1 IDIAP Presentation

IDIAP is a semi-private research institute located in Martigny, Valais, and affiliated with the Swiss Federal Institute of Technology (EPFL) at Lausanne, and the University of Geneva. IDIAP carries out research in the areas of speech and speaker recognition, computer vision and machine learning.

1.1.1 Machine learning

Statistical Machine Learning is a research domain mostly related to statistical inference, artificial intelligence, and optimization. Its aim is to construct systems able to learn to solve tasks given a set of examples that were drawn from an unknown probability distribution, and given some a priori knowledge of the task. Another important goal of Statistical Machine Learning is to measure the expected performance of these systems on new examples drawn from the same probability distribution

1.1.2 Speech processing

The overall goals of the IDIAP speech processing group are to research and develop robust recognition and understanding techniques for realistic speaking styles and acoustic conditions, as well as robust speaker verification and identification techniques. This includes advanced research activities, maintenance of language resources for the training and testing of recognition systems, and development of real-time prototypes. The group has been involved in speech research projects for several years and is today at the leading edge of technology. The IDIAP Speech Processing group is also involved in numerous national and European collaborative projects, as well as industrial projects.

Domains of application

The IDIAP Speech Group aims at developing systems to perform:

- Speaker independent speech recognition with small and medium vocabulary sizes. This includes word spotting for the detection of vocabulary words and rejection of Out Of Vocabulary (OOV) words.
- Speaker recognition (verification and identification) for secure voice servers applications. IDIAP takes part in several European projects and participates to the NIST evaluations. Multimodal systems are also built, combining the speaker verification techniques from both Speech and Vision groups.
- Large Vocabulary Continuous Speech Recognition (LVCSR), applied to the automatic transcription of natural speech data. Research is performed at the different modeling layers: acoustic, lexical and linguistic.
- Voice thematic indexing. In this domain, the recognition system permits to archive and retrieve audio material. This is a challenging application due to the random (generally bad) quality of the speech data to archive.
- Speech processing in smart meeting rooms

1.1.3 Vision Group

The computer vision group studies problems in machine visual perception, such as media annotation, people detection and human gesture tracking and recognition. Research activities center on multi-modal interpretation of visual and multimedia data, and improvement of basic detection and classification measures and algorithms. This improvement may be achieved by enhancing and extending existing algorithms, or by creating new algorithms and measures. This frequently involves collaboration across research groups, as complementary expertise is brought to bear on a problem.

There is strong expertise within the vision group in areas of text processing from both documents and video, object tracking and recognition of gesture, and domain based video annotation. The group is active in all of these areas under a number of collaborative European and Swiss national projects.

Research Area

Document Analysis and Recognition: Work in this area involves applying non-traditional methods to improve both preprocessing and recognition steps. In particular, methods from the speech processing area are being examined for use to improve recognition. Improved pre-processing: Statistical methods and HMM techniques are applied to improve the normalization and word modeling process, continuing improvements in the preprocessing step for hand written and cursive text.

Sentence recognition: Speech recognition methods such as language models are being applied to improving sentence recognition in script recognition. This is made possible by the statistical word modeling approach taken in earlier processing.

Image and video annotation: Multimedia annotation involves both visual and audio data, and the deduction of higher level, semantic annotation which must be conducted under a machine learning framework. The work in this area thus brings together all three groups at IDIAP.

Accretive annotation for video: Accretive annotation uses low level feature information from a number of modalities, such as audio and video, to work towards semantic annotation. For this work domain specific information is used to provide a framework for interpreting the low level data, to direct progressively higher level processing for increasingly detailed annotation.

Text Detection and Recognition in Images and Videos: The vision group is involved in text detection and segmentation algorithms, and also examination of new paradigms in video text recognition. The goal of current research is to reduce false positive detection, and move away from explicit segmentation as a preprocessing step.

Face Algorithms: Face algorithm can be divided into four different areas.

- Face detection : The goal of face detection is to identify and locate human faces in images at different positions, scales, orientations and lighting conditions.
- Face localization : Face localization is a simplified face detection problem with the assumption that the image contain only one face.
- Face verification : Face verification is concerned with validating a claimed identity based on the image of its face, and either accepting or rejecting the identity claim.
- Face recognition : The goal of face recognition is to identify a person based on the image of its face. This face image has to be compared with all the registered persons. Therefore, face recognition is computationally expensive regarding the number of registered persons.

Gesture Recognition: Gestural interfaces based on the image is the most natural way for the construction of advanced man-machine interfaces. Thus, machines would be easier to use by associating the gestural command with the vocal command. It is necessary to distinguish two aspects of hand gestures:

- the static aspect is, for instance, characterized by a posture of the hand in an image. Hand postures for example can be used to execute commands.
- the dynamic aspect is defined either by the trajectory of the hand, or by the sequence of hand postures in a sequence of images.

Application Examples

The purpose of image and video annotation is to provide access to the ever increasing digital archives of such data. Whether these archives are within a television station or publicly available web documents, the sheer volume of data being produced at any moment is beyond human ability to annotate. In addition there are large historical archives that contain priceless data recording important moments. Television stations will use such technology to provide a method of accessing their archives, such as sports and news, and to access historical footage to enrich current programs, and for documentary pieces. Video and image text recognition is obviously a key part of this technology, as captions and in-vision text contain useful information.

Hand drawn character and cursive writing recognition is useful for such tasks as automated address reading for postal services, and interface to such devices as PDAs. In addition, notes taken in meetings and during other discussions are predominantly handwritten. The ability to read such sources of information would be very useful in many cases.

1.2 Problem description and existing work

Problem description

At IDIAP, the computer vision group works on videos to study problems in machine visual perception, such as person detection, face and gesture tracking and recognition, within meeting room videos and sports videos. Thus, researchers need tools to decode, extract pictures and encode videos. In fact, you cannot work directly on a compressed video because you need to decode it to extract a raw picture before and process it.

The current solution used at IDIAP is not optimal. The first step is to extract each frame to a PPM picture. PPM is a very simple picture format, without compression, where pixels are simply stored in RGB format. Researchers commonly use it because they can quickly load a matrix of pixels, process it and save the result as a new picture. Of course, this format use too many space on disk : for example, if a compressed video source has 1000 frames and takes one megabyte, the extracted pictures will takes about 100 megabytes. As you can see on figure 1.1, processing these frames will generate 1000 more temporary images, and then 100 more Mb, before re-encoding an output video file.

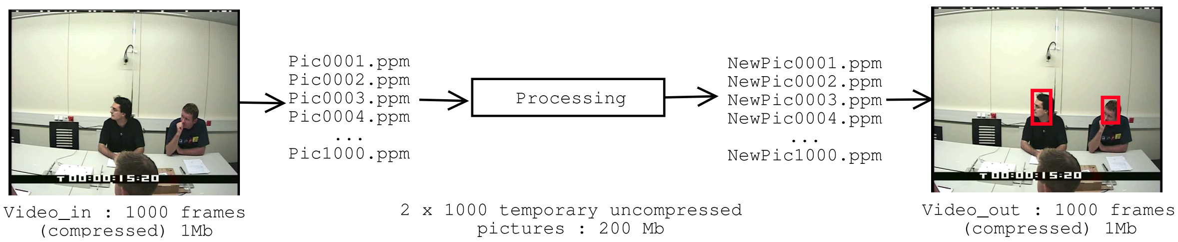

Figure 1.1: Currently solution to work on videos at IDIAP

Finally, to analyze a video, researchers need a tool to extract frames, another one to build a video from pictures and their own program to work on pictures. The complete process is slow, use large disk space and is not optimal from a computational point of view.

1.2.1 VisionLib and MPlayer

A first solution was proposed with VisionLib. VisionLib is a library build by the computer vision group to work with videos. It contains many classes to analyze pictures and a video class to read video files. This solution permits to delete the first step, the frame extraction. Unfortunately, this library is not able to make an output video file.

The VisionLib video package is based on MPlayer. MPlayer is one of the best video player on Linux. This player is able to read nearly all knowed formats. Although it works fine, this solution has a lot of limitations.

- First, this is program and not a library with API, so, an hacked version of MPlayer have been used.
- The source code is very complex to interface and needs many external libraries to works fine.
- MPlayer supports too many formats for IDIAP needs. The consequence is the size of final binaries.
- MPlayer is not able to encode videos. For that, it needs MEncoder which is not integrated.
- Finally, MPlayer cannot read several files at the same time. This is a very important limitation because researchers often need to output the result of their processing as new video files.

VisionLib and its video class offer a good temporary solution. But there is too many limitations and the vision group need a better tool to works on videos.

1.2.2 Torch

Torch[1] is a machine learning library, written in C++, which is under BSD license. The ultimate objective is to contain all the state-of-the-art machine learning algorithms, for both static and dynamic problems. Currently, it contains all sorts of Artificial Neural Networks (including convolutional network and time-delay neural networks), Support Vector Machines for regression and classification, Gaussian Mixture Models, Hidden Markov Models, Kmeans, K nearest… Several additional package exist to do speech recognition, speaker verification, face detection and face recognition.

Main Concepts of Torch

Torch has been developed using an object-oriented paradigm and implemented in `C++`. In order to simplify the modification of existing algorithms or the design of new algorithms or methods, a modular strategy was chosen through the definition of the following broad classes:

- `DataSet`: this class handles data. Several subclasses provide ways to handle static or dynamic data, data that can fit into memory or which could be accessed "on-the-fly" from disk (for very large data sets for example), *etc...*

- `Machine`: this class represents any black-box that, given an (optional) input (again, either static or dynamic) and some (optional) parameters, returns an output. It could be for instance a neural network, a support vector machine, a hidden Markov model, *etc...*

- `Trainer`: this class is used to select an optimal set of parameters of a machine according to a given criterion and a given `DataSet`, and test it using another (or the same) `DataSet`.

- `Measurer`: objects of this class print in different files various measures of interest. It could be for example the classification error, the mean-squared error or the log-likelihood.

Thus, the general idea of Torch is very simple: first the `DataSet` produces one or several "training examples". The `Trainer` gives them to the `Machine` which computes an output, which is used by the `Trainer` to tune the parameters of the `Machine`. During this process, one or more `Measurer`(s) can be used to monitor the performance of the system. Note however that some machines can only be trained by specific trainers:

- various "gradient machines" (including multi-layer perceptrons) can be trained by gradient descent,

- support vector machines, for classification or regression can be trained by a trainer specialized on constrained quadratic problems,

- distributions (such as Gaussian mixture models or hidden Markov models) are usually trained using an Expectation-Maximization (or its Viterbi approximation) trainer but can also be trained by gradient descent.

Torch vision package

A vision package have been developed by the vision group to work easiest on images and videos. This package provides a lot of methods to open, read or write files, to process them (resizing, filtering, converting…) and to analyse them (DCT, gaussian, PCA…)

[1] Torch is available at http://www.torch.ch.

Here is a non-exhaustive list of features:

- **Image I/O :** The vision package permits to open, read, write and close basics images formats (PGM, PPM, GIF, TIF, JPEG). The purpose is to get a matrix of RGB pixels, that is easy to manipulate, and save results quickly.
- **Image Processing :** Images often need to be normalized for a futur processing. The package provides the basic operations like edges, motions, color filtering, rotation, photometric normalisation, flip…
- **Features extraction :**A lot of features extraction are also available, like PCA (eigenfaces), LDA (fisherfaces), DCT, objects detection (image search and pattern recognition), face detection and verification…
- **Geometry :** In order to show the results, the package provides some methods to display some 2D geometry.

This vision package is fully compatible with Torch and is also under BSD license.

IDIAP researchers need a video package, integrated in Torch, which would be easy to install and easy to use, which could read and write basics video formats. If possible, a library using an external video coder and decoder.

Chapter 2

Overview

2.1 Introduction to video

Basis

A video is a sequence of video and audio frames. Audio frames could be at the end of file or interlaced between video frames. The frame rate is the number of pictures per second.

A picture (or a video frame) is a structured array of pixels. The size of a picture is named resolution and gives the total number of pixels. For example, for a picture of size 100x200, the width is 100 pixels and the weight is 200 pixels. In case the picture is RGB[1] (three bytes per pixels) the image size will be 60000 Bytes (58.5 KB).

The pixel [5] (a word invented from "picture element") is the basic unit of programmable color on a computer display or in a computer image. Up to three bytes of data are allocated for specifying a pixel's color, one byte for each major color component. A true color or 24-bit color system uses all three bytes. However, many color display systems use only one byte (limiting the display to 256 different colors).

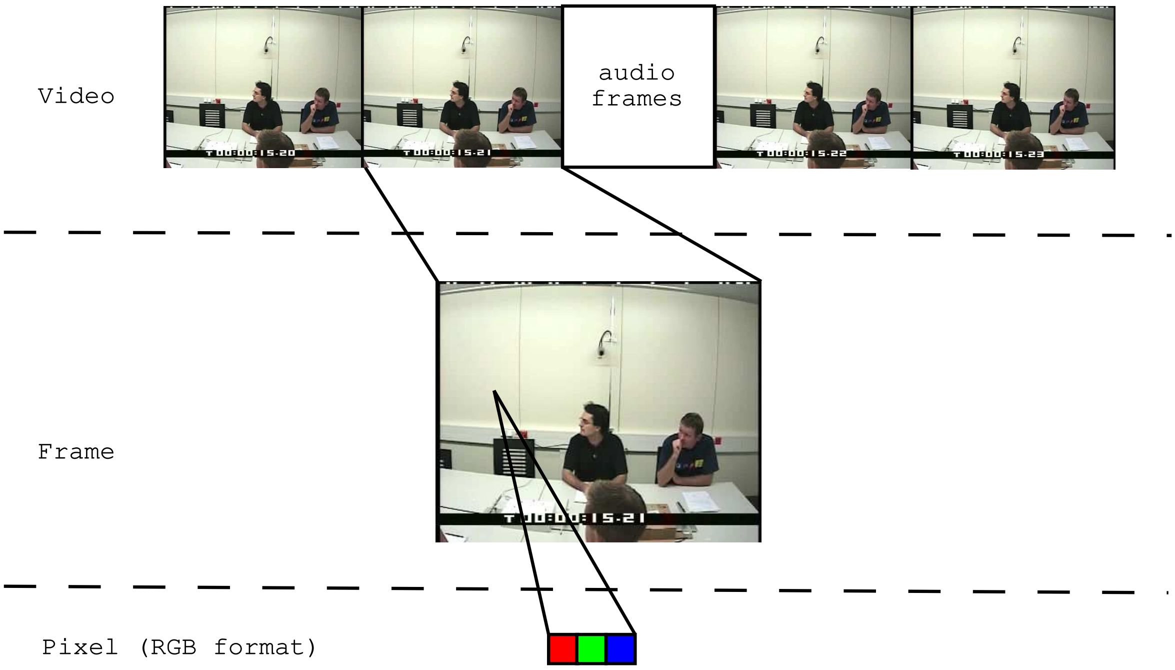

Figure 2.1: Video basis

2.1.1 Format and codec

Currently, the type of files the most used on computers are multimedia files. A multimedia file is a file who contains video and audio. In this part, I will focus on the video. Usually, when people speak about a video file, they tell "This is a AVI file" or "This is a DivX", which is completely different. They confuse the format and the codec. The format is a container (a type of file) whereas the codec is a compression method. The confusion is due to some formats and codecs that have the same name (mpeg for example). Knowing the format of a video file is very easy because it's generally given by

[1] RGB stands for Red-Green-Blue colorspace

the extension of the file, but the codec can't be known so easily. The list of formats and codec is very long and there's no point in knowing all because only about ten are often used. There is two types of formats : the container formats (like AVI), they can contain many different kinds of compression and other formats who are linked with a specific compression method (like mpeg).

What is a format? The format is the structure of the video file. It contains all needed parameters to read the video, such as size, length, used codec, number of frames per second, codec parameters... There are often an index of keyframes[2].

What is a codec? A codec is a compression method. So, what is a compression? It's a method to save space, with a given quality.

Why compression? First, lets have a example to understand benefits of compression. Imagine a 10 minutes video where the picture size is 720x576 (classical digital camcorder video size). So, this video contains 10 minutes * 60 seconds * 25 frames = 15 000 frames. In this case, a non-compressed frame size is 720 * 576 * 3 = 1215 kBytes. Then, the video file size is 15000 * 1215 = 18 225 000 kBytes (17 Giga Bytes, more than 25 cd !). Actually, a good compression on this video, with nearly the same quality, can create a 60 MBytes video file. By compressing a video file, you reduce its size, but you also reduce its bitrate. The biterate is the number of bits per second necessary to read, or transmit the video. Due to the newest compression method, you can more and more watch videos over Internet, in real time (streaming). The lost is the biterate, the smaller is the video file but the worst is the quality of the image. In the next chapter, we will see in details how works video compression.

2.1.2 AVI

One of the oldest formats in the x86 computer world is AVI[2]. The abbreviation 'AVI' stands for 'Audio Video Interlaced'. This video format was created by Microsoft, which was introduced along with Windows 3.1. AVI, the proprietary format of Microsoft's "Video for Windows" application, merely provides a framework for various compression algorithms such as Cinepak, Intel Indeo, Microsoft Video 1, Clear Video or IVI. In its first version, AVI supported a maximum resolution of 160 x 120 pixels with a refresh rate of 15 frames per second. The format attained widespread popularity, as the first video editing systems and software appeared that used AVI by default.

AVI is a file format, like MP3 or JPG. But unlike these formats, AVI is a container format, meaning it can contain video audio compressed using many different combinations of codecs. So while MP3 and JPG can only contain a certain kind of compression (MPEG Audio Layer 3 and JPEG), AVI can contain many different kinds of compression (DivX video + WMA audio or Indeo video + PCM audio), as long as a codec is available for encoding/decoding. AVI all look the same on the "outside", but on the "inside", they may be completely different. There is no such thing as a "normal" AVI file, but the closest you can get is probably an AVI file that contains no compression. One of the main problem with the standard AVI format is the file size limitation. The current RIFF[3] file format implies a maximum chunk size of 4 GB because the size is stored as a 32-bit value. Note that AVI files without file limits are usually referred to as OpenDML AVI files.

AVI RIFF file structure

Please note that this section has been produced based on text and image material from Microsoft developers website[1].

[2] A keyframe is a non compressed frame, used for seeking.

[3] RIFF is a device control interface and common file format native to the Microsoft Windows system. It is used to store audio (wav files), video(avi files), and graphics information used in multimedia applications .

FOURCCs A FOURCC[4] (four-character code) is a 32-bit unsigned integer created by concatenating four ASCII characters[1]. For example, the FOURCC 'abcd' is represented on a Little-Endian system as 0x64636261. FOURCCs can contain space characters, so ' abc' is a valid FOURCC. The AVI file format uses FOURCC codes to identify stream types, data chunks, index entries, and other information.

RIFF File Format The AVI file format is based on the RIFF (resource interchange file format) document format. A RIFF file consists of a RIFF header followed by zero or more lists and chunks.

- The RIFF header has the following form:

  ```
  'RIFF' fileSize fileType (data)
  ```

  where `RIFF` is the literal FOURCC code `RIFF`, `fileSize` is a 4-byte value giving the size of the data in the file, and `fileType` is a FOURCC that identifies the specific file type. The value of `fileSize` includes the size of the `fileType` FOURCC plus the size of the data that follows, but does not include the size of the 'RIFF' FOURCC or the size of `fileSize`. The file data consists of chunks and lists, in any order.

- A chunk has the following form:

  ```
  ckID ckSize ckData
  ```

  where `ckID` is a FOURCC that identifies the data contained in the chunk, `ckSize` is a 4-byte value giving the size of the data in `ckData`, and `ckData` is zero or more bytes of data. The data is always padded to nearest WORD boundary. `ckSize` gives the size of the valid data in the chunk; it does not include the padding, the size of `ckID`, or the size of `ckSize`.

- A list has the following form:

  ```
  'LIST' listSize listType listData
  ```

  where `LIST` is the literal FOURCC code `LIST`, `listSize` is a 4-byte value giving the size of the list, `listType` is a FOURCC code, and `listData` consists of chunks or lists, in any order. The value of `listSize` includes the size of `listType` plus the size of `listData`; it does not include the `LIST` FOURCC or the size of `listSize`.

The remainder of this section uses the following notation to describe RIFF chunks:

```
ckID ( ckData )
```

where the chunk size is implicit. Using this notation, a list can be represented as:

```
'LIST' ( listType ( listData ) )
```

AVI RIFF Form AVI files are identified by the FOURCC 'AVI ' in the `RIFF` header. All AVI files include two mandatory LIST chunks, which define the format of the streams and the stream data, respectively. An AVI file might also include an index chunk, which gives the location of the data chunks within the file. An AVI file with these components has the following form:

```
RIFF ('AVI '
      LIST ('hdrl' ... )
      LIST ('movi' ... )
      ['idx1' (<AVI Index>) ]
     )
```

[4]see http://www.fourcc.org

The hdrl list defines the format of the data and is the first required LIST chunk. The movi list contains the data for the AVI sequence and is the second required LIST chunk. The 'idx1' list contains the index. AVI files must keep these three components in the proper sequence.

Note The OpenDML extensions define another type of index, identified by the FOURCC indx.

The hdrl and movi lists use subchunks for their data. The following example shows the AVI RIFF form expanded with the chunks needed to complete these lists:

```
RIFF ('AVI '
      LIST ('hdrl'
            'avih'(<Main AVI Header>)
            LIST ('strl'
                  'strh'(<Stream header>)
                  'strf'(<Stream format>)
                  [ 'strd'(<Additional header data>) ]
                  [ 'strn'(<Stream name>) ]
                  ...
                 )
             ...
           )
      LIST ('movi'
            {SubChunk | LIST ('rec '
                                SubChunk1
                                SubChunk2
                                ...
                               )
               ...
            }
            ...
           )
      ['idx1' (<AVI Index>) ]
     )
```

AVI Main Header The hdrl list begins with the main AVI header, which is contained in an avih chunk. The main header contains global information for the entire AVI file, such as the number of streams within the file and the width and height of the AVI sequence.

AVI Stream Headers One or more strl lists follow the main header. A strl list is required for each data stream. Each strl list contains information about one stream in the file, and must contain a stream header chunk (strh) and a stream format chunk (strf). In addition, a strl list might contain a stream-header data chunk (strd) and a stream name chunk (strn).

A stream format chunk (strf) must follow the stream header chunk. The stream format chunk describes the format of the data in the stream. The data contained in this chunk depends on the stream type.

If the stream-header data (strd) chunk is present, it follows the stream format chunk. The format and content of this chunk are defined by the codec driver. Typically, drivers use this information for configuration. Applications that read and write AVI files do not need to interpret this information; they simple transfer it to and from the driver as a memory block.

The optional strn chunk contains a null-terminated text string describing the stream.

The stream headers in the `hdrl` list are associated with the stream data in the `movi` list according to the order of the `strl` chunks. The first `strl` chunk applies to stream 0, the second applies to stream 1, and so forth.

Stream Data ('movi' List) Following the header information is a `movi` list that contains the actual data in the streams. The data chunks can reside directly in the `movi` list, or they might be grouped within 'rec ' lists. The 'rec ' grouping implies that the grouped chunks should be read from disk all at once, and is intended for files that are interleaved to play from CD-ROM.

The FOURCC that identifies each data chunk consists of a two-digit stream number followed by a two-character code that defines the type of information in the chunk.

Two-character code	**Description**
db	Uncompressed video frame
dc	Compressed video frame
pc	Palette change
wb	Audio data

For example, if stream 0 contains audio, the data chunks for that stream would have the FOURCC '`00wb`'. If stream 1 contains video, the data chunks for that stream would have the FOURCC '`01db`' or '`01dc`'. Video data chunks can also define new palette entries to update the palette during an AVI sequence.

AVI Index Entries An optional index (`idx1`) chunk can follow the `movi` list. The index contains a list of the data chunks and their location in the file. It consists of an structure with entries for each data chunk, including `rec` chunks.

Other Data Chunks Data can be aligned in an AVI file by inserting `JUNK` chunks as needed. Applications should ignore the contents of a `JUNK` chunk.

OpenDML or AVI2.0

The OpenDML File Format Subcommittee is defining an AVI-compatible file format that addresses the particular needs of professional video.

2.1.3 Fundamentals of MPEG Video Compression Algorithms

Please note that this section has been produced based on text and image material from a book in [7]

Generally speaking, video sequences contain a significant amount of statistical and subjective redundancy within and between frames. The ultimate goal of video source coding is the bit-rate reduction for storage and transmission by exploring both statistical and subjective redundancies and to encode a "minimum set" of information using entropy coding techniques. This usually results in a compression of the coded video data compared to the original source data. The performance of video compression techniques depends on the amount of redundancy contained in the image data as well as on the actual compression techniques used for coding. With practical coding schemes a trade-off between coding performance (high compression with sufficient quality) and implementation complexity is targeted.

Source model The MPEG digital video coding techniques are statistical in nature. Video sequences usually contain statistical redundancies in both temporal and spatial directions. The basic statistical property upon which MPEG compression techniques rely is inter-pixel correlation, including the assumption of simple correlated translatory motion between consecutive frames. Thus, it is assumed that the magnitude of a particular image pixel can be predicted from nearby pixels within the same frame (using Intra-frame coding techniques) or from pixels of a nearby frame (using Inter-frame techniques). Intuitively it is clear that in some circumstances, i.e. during scene changes of a video sequence, the temporal correlation between pixels in nearby frames is small or even vanishes - the video scene then assembles a collection of uncorrelated still images. In this case Intra-frame coding techniques are appropriate to explore spatial correlation to achieve efficient data compression. The MPEG compression algorithms employ Discrete Cosine Transform (DCT) coding techniques on image blocks of 8x8 pixels to efficiently explore spatial correlations between nearby pixels within the same image. However, if the correlation between pixels in nearby frames is high, i.e. in cases where two consecutive frames have similar or identical content, it is desirable to use Inter-frame DPCM coding techniques employing temporal prediction (motion compensated prediction between frames). In MPEG video coding schemes an adaptive combination of both temporal motion compensated prediction followed by transform coding of the remaining spatial information is used to achieve high data compression (hybrid DPCM/DCT coding of video).

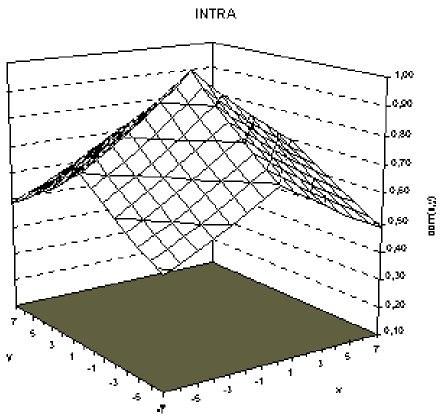

Figure 2.2: Spatial inter-element correlation of "typical" images as calculated using a AR(1) Gauss Markov image model with high pixels correlation. Variables x and y describe the distance between pixels in horizontal and vertical image dimensions respectively.

Subsampling and Interpolation The basic concept of subsampling is to reduce the dimension of the input video (horizontal dimension and/or vertical dimension) and thus the number of pixels to be coded prior to the encoding process. It is worth noting that for some applications video is also subsampled in temporal direction to reduce frame rate prior to coding. At the receiver the decoded images are interpolated for display. This technique may be considered as one of the most elementary compression techniques which also makes use of specific physiological characteristics of the human eye and thus removes subjective redundancy contained in the video data. The human eye is more sensitive to changes in brightness than to chromaticity changes. Therefore the MPEG coding schemes first divide the images into YUV components (one luminance and two chrominance components). Next the chrominance components are subsampled relative to the luminance component with a Y:U:V ratio specific to particular applications (with the MPEG-2 standard a ratio of 4:1:1 or 4:2:2 is used).

Motion Compensated Prediction Motion compensated prediction is a powerful tool to reduce temporal redundancies between frames and is used extensively in MPEG-1 and MPEG-2 video coding standards as a prediction technique for temporal DPCM coding. The concept of motion compensation is based on the estimation of motion between video frames. If all elements in a video scene are approximately spatially displaced, the motion between frames can be described by a limited number of motion parameters (by motion vectors for translatory motion of pixels). However, encoding one motion information with each coded image pixel is generally neither desirable nor necessary. Since the spatial correlation between motion vectors is often high it is sometimes assumed that one motion vector is representative for the motion of a "block" of adjacent pixels. To this aim images are usually separated into disjoint blocks of pixels (i.e. 16x16 pixels in MPEG-1 and MPEG-2 standards) and only one motion vector is estimated, coded and transmitted for each of these blocks. In the MPEG compression algorithms the motion compensated prediction techniques are used for reducing temporal redundancies between frames and only the prediction error images - the difference between original images and motion compensated prediction images - are encoded. In general the correlation between pixels in the motion compensated Inter-frame error images to be coded is reduced compared to the correlation properties of Intra-frames due to the prediction based on the previous coded frame.

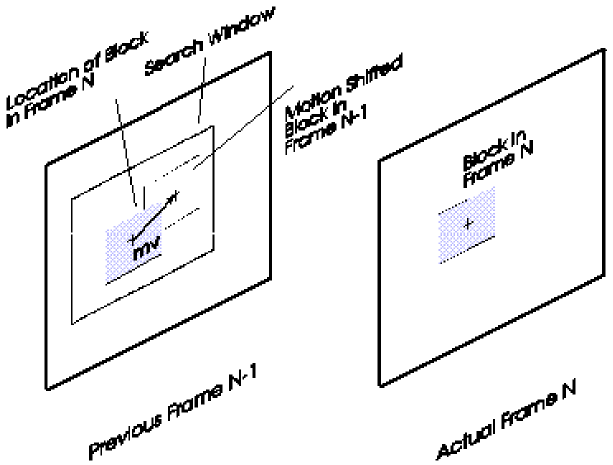

Figure 2.3: Block matching approach for motion compensation: One motion vector (mv) is estimated for each block in the actual frame N to be coded. The motion vector points to a reference block of same size in a previously coded frame N-1. The motion compensated prediction error is calculated by subtracting each pixel in a block with its motion shifted counterpart in the reference block of the previous frame.

Transform Domain Coding Transform coding has been studied extensively during the last two decades and has become a very popular compression method for still image coding and video coding. The purpose of Transform coding is to de-correlate the Intra- or Inter-frame error image content and to encode Transform coefficients rather than the original pixels of the images. To this aim the input images are split into disjoint blocks of pixels b. The transformation can be represented as a matrix operation using a NxN Transform matrix A to obtain the NxN transform coefficients c based on a linear, separable and unitary forward transformation

c = A b AT.

Here, AT denotes the transpose of the transformation matrix A. Note, that the transformation is reversible, since the original NxN block of pixels b can be reconstructed using a linear and separable inverse transformation

b = ATc A.

Upon many possible alternatives the Discrete Cosine Transform (DCT) applied to smaller image blocks of usually 8x8 pixels has become the most successful transform for still image and video coding[4]. In fact, DCT based implementations are used in most image and video coding standards due to their high decorrelation performance and the availability of fast DCT algorithms suitable for real time implementations.

A major objective of transform coding is to make as many Transform coefficients as possible small enough so that they are insignificant (in terms of statistical and subjective measures) and need not be coded for transmission. At the same time it is desirable to minimize statistical dependencies between coefficients with the aim to reduce the amount of bits needed to encode the remaining coefficients. Figure 2.4 depicts the variance (energy) of a 8x8 block of Intra-frame DCT coefficients based on the simple statistical model assumption already discussed in Figure 1. Here, the variance for each coefficient represents the variability of the coefficient as averaged over a large number of frames. Coefficients with small variances are less significant for the reconstruction of the image blocks than coefficients with large variances. As may be depicted from Figure 2.4, on average only a small number of DCT coefficients need to be transmitted to the receiver to obtain a valuable approximate reconstruction of the image blocks. Moreover, the most significant DCT coefficients are concentrated around the upper left corner (low DCT coefficients) and the significance of the coefficients decays with increased distance. This implies that higher DCT coefficients are less important for reconstruction than lower coefficients. Also employing motion compensated prediction the transformation using the DCT usually results in a compact representation of the temporal DPCM signal in the DCT-domain - which essentially inherits the similar statistical coherency as the signal in the DCT-domain for the Intra-frame signals in Figure 2.4 (although with reduced energy) - the reason why MPEG algorithms employ DCT coding also for Inter-frame compression successfully[6].

The DCT is closely related to Discrete Fourier Transform (DFT) and it is of some importance to realize that the DCT coefficients can be given a frequency interpretation close to the DFT. Thus low DCT coefficients relate to low spatial frequencies within image blocks and high DCT coefficients to higher frequencies. This property is used in MPEG coding schemes to remove subjective redundancies contained in the image data based on human visual systems criteria. Since the human viewer is more sensitive to reconstruction errors related to low spatial frequencies than to high frequencies, a frequency adaptive weighting (quantization) of the coefficients according to the human visual perception (perceptual quantization) is often employed to improve the visual quality of the decoded images for a given bit rate.

The combination of the two techniques described above - temporal motion compensated prediction and transform domain coding - can be seen as the key elements of the MPEG coding standards. A third characteristic element of the MPEG algorithms is that these two techniques are processed on small image blocks (of typically 16x16 pels for motion compensation and 8x8 pixels for DCT coding).

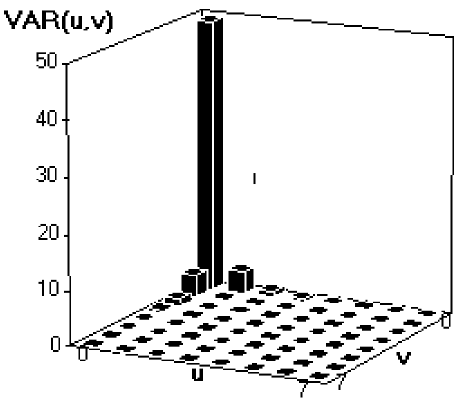

Figure 2.4: The figure depicts the variance distribution of DCT-coefficients "typically" calculated as average over a large number of image blocks. Most of the total variance is concentrated around the DC DCT-coefficient (u=0, v=0).

To this reason the MPEG coding algorithms are usually refered to as hybrid block-based DPCM/DCT algorithms.

2.1.4 MPEG4 Video Algorithm

Please note that this section has been produced based on text and image material from a book in [7]

MPEG4 is now the new standard on computers, but not yet on other multimedia supports (DVD, DVB...). The norm provides new features and suggest compression methods. Peoples who want create an MPEG4 codec must respect this norm but don't have to include each features (because some of them are simply not yet developed). They must choose compression methods, for intra (dct or wavelet) or predicted frames for example. Most popular codec are actually divx, xvid, sorenson (used on quicktime movies), rm9 (the newest real codec) and Windows Media 9. H264 will probably be the next standard, but it is actually under development and need important hardware requirements.

The MPEG-4 Video coding algorithm support all functionalities already provided by MPEG-1 and MPEG-2, including the provision to efficiently compress standard rectangular sized image sequences at varying levels of input formats, frame rates and bit rates as well as provisions for interlaced input sources.

Content-Based Scalability Furthermore, one of the main "content"-based MPEG-4 Video functionalities, is the support for the separate encoding and decoding of content. This feature is the ability to identify and selectively decode and reconstruct video content of interest. It provides the most elementary mechanism for interactivity and manipulation of content of video without the need for further segmentation.

VOP To enable the content based interactive functionalities, the MPEG-4 Video Verification Model introduces the concept of Video Object Planes (VOP's). Each frame of an input video sequence is segmented into a number of shaped image regions (video object planes). Each of them may cover particular image or video content of interest, describing physical objects or content. In contrast to the MPEG-1 and MPEG-2 standards, the video input is thus no longer considered a rectangular region.

Coding of Shape, Motion and Texture Information for each VOP

The information related to the shape, motion and texture information for each VO, is coded into a separate VOL-layer in order to support separate decoding of VO's. The shape information is not transmitted if the input video to be coded contains only standard images of rectangular size. In this case the MPEG-4 Video coding algorithm has a structure similar to the MPEG-1/2 coding algorithms.

The MPEG-4 compression algorithm employed for coding each VOP image sequence is based on the successful block-based coding technique already employed in the MPEG coding standards[3]. The MPEG-4 coding algorithm encodes the first VOP in Intra-Frame. Each subsequent frame is coded using Inter-frame prediction. Only data from the nearest previously coded VOP frame is used for this prediction. In addition the coding of B-directionally predicted is also supported.

The MPEG-4 Video Verification Model supports the coding of both forward predicted (P) as well as bi-directionally (B) predicted VOP's. Motion vectors are predictively coded using standard MPEG-1/2 code tables including the provision for extended vector ranges. Notice, that the coding of standard MPEG I-frames, P-frames and B-frames is still supported by the Verification Model (for the special case of image input sequences (VOP's) of rectangular shape).

Texture Coding The Intra VOP's as well as the residual errors after motion compensated prediction are coded using a DCT on 8x8 blocks similar to the standard MPEG standard. Again, the adaptive VOP window Macroblock grid is employed for this purpose. For each Macroblock a maximum of four 8x8 Luminance blocks and two 8x8 Chrominance blocks are coded. Scanning of the DCT coefficients followed by quantization and run-length coding of the coefficients is performed using

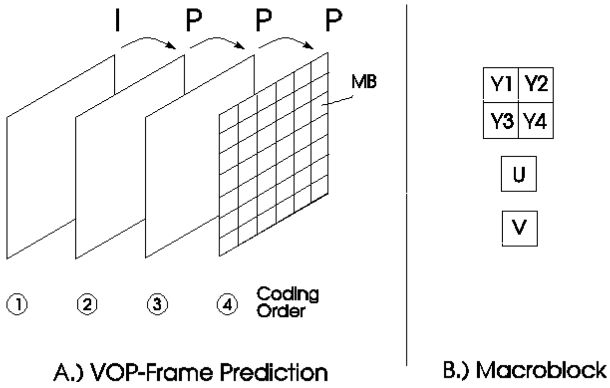

Figure 2.5: Illustration of an I-picture VOP and P-picture VOP's in a video sequence. P-VOP's are coded using motion compensated prediction based on the nearest previous VOP frame. Each frame is divided into disjoint "Macroblocks" (MB). With each Macroblock, information related to four luminance blocks (Y1, Y2, Y3, Y4) and two chrominance blocks (U, V) is coded. Each block contains 8x8 pixels.

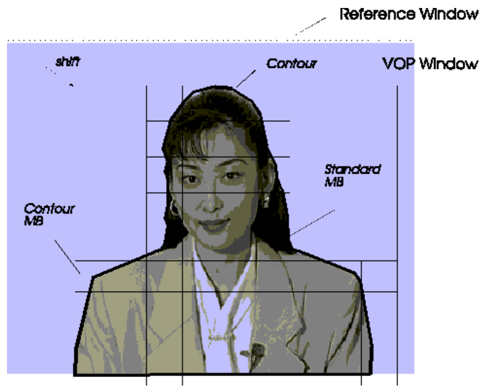

Figure 2.6: Example of a MPEG-4 VM Macroblock grid for a foreground VOP image. This Macroblock grid is used for alpha plane coding, motion estimation and compensation as well as for block based DCT-based texture coding. A VOP window with a size of multiples of 16 pixels in each image direction surrounds the foreground VOP of arbitrary shape and specifies the location of the Macroblocks, each of size 16x16 pixels. This window is adjusted to collocate with the most top and most left border of the VOP. A shift parameter is coded to indicate the location of the VOP window with respect to the borders of a reference window (original image borders).

techniques and VLC tables defined with the MPEG-1/2 standards. An efficient prediction of the DC- and AC-coefficients of the DCT is performed for Intra coded VOP's.

Multiplexing of Shape, Motion and Texture Information Basically all "tools" (DCT, motion estimation and compensation...) defined in the MPEG-1 standards are currently supported by the MPEG-4 Video Verification Model. The compressed alpha plane, motion vector and DCT bit words are multiplexed into a VOL layer bitstream by coding the shape information first, followed by motion and texture coding based on the MPEG definitions.

The Verification Model defines two separate modes for multiplexing texture and motion information: A joint motion vector and DCT-coefficient coding procedure based on standard H.263. This guarantees that the performance of the VM at very low bit rates is at least identical to the H.263 standard. Alternatively, the separate coding of motion vectors and DCT-coefficients is also possible, to eventually incorporate new and more efficient motion or texture coding techniques.

Spatial and Temporal Scalability

An important goal of scalable coding of video is to flexibly support receivers with different bandwidth or display capabilities or display requests to allow video database browsing. Another important purpose of scalable coding is to provide a layered video bit stream which is amenable for prioritized transmission. The techniques adopted for the MPEG-4 Video Verification Model allow the "content-based" access or transmission of arbitrarily-shaped VOP's at various temporal or spatial resolutions. Receivers either not capable or willing to reconstruct the full resolution arbitrarily shaped VOP's can decode subsets of the layered bit stream to display the arbitrarily shaped VOP's content/objects at lower spatial or temporal resolution or with lower quality.

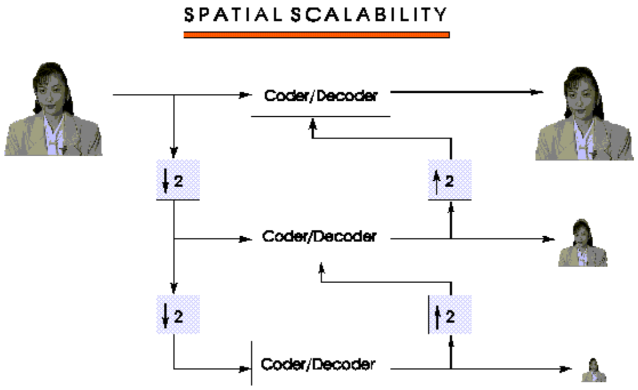

Figure 2.7: Spatial scalability approach for arbitrarily shaped VOP's.

Spatial Scalability The figure 2.7 depicts the MPEG-4 general philosophy of a content-based VOP multiscale video coding scheme. Here three layers are provided, each layer supporting a VOP at different spatial resolution scales. The downscaled version is encoded into a base layer bit stream with reduced bit rate. The upscaled reconstructed base layer video is used as a prediction for the coding of the original input video signal. The prediction error is encoded into an enhancement layer bit stream. If a receiver is either not capable or willing to display the full quality VOP's, downscaled VOP signals can be reconstructed by only decoding the lower layer bit streams. It is important to notice that the display of the VOP at highest resolution with reduced quality is also possible by only decoding the lower bit rate base layers. Thus scaleable coding can be used to encode content-based video with a suitable bit rate allocated to each layer in order to meet specific bandwidth requirements of transmission channels or storage media. Browsing through video data bases and transmission of video over heterogeneous networks are applications expected to benefit from this functionality.

Temporal Scalability Different frame rates can be supported with a layered bit stream. Using the MPEG-4 "content-based" VOP temporal scalability approach, it is possible to provide different display rates for different VOL's within the same video sequence (a foreground person of interest may be displayed with a higher frame rate compared to the remaining background or other objects).

Error Resilience

A considerable effort has been made to investigate the robust storage and transmission of MPEG-4 video in error prone environments. To this end an adaptive Macroblock Slice technique similar to the one already provided with the MPEG-1 and MPEG-2 standards has been introduced into the MPEG-4. The technique provides resynchronization bit words for groups of Macroblocks and has been optimized in particular to achieve efficient robustness for low bit rate video under a variety of severe error conditions (for the transmission over mobile channels).

2.2 Libraries available

The first job to do to replace MPlayer was to find a better multimedia decoder and encoder. Here, better means more suitable to Torch, faster and easier to implement. It must be free and open source.

2.2.1 overview

transcode (official overview) transcode[5] is a linux text-console utility for video stream processing, running on a platform that supports shared libraries and threads. Decoding and encoding is done by loading modules that are responsible for feeding transcode with raw video/audio streams (import modules) and encoding the frames (export modules). It supports elementary video and audio frame transformations, including de-interlacing or fast resizing of video frames and loading of external filters. A number of modules are included to enable import of DVDs on-the-fly, MPEG elementary (ES) or program streams (VOB), MPEG video, Digital Video (DV), YUV4MPEG streams, NuppelVideo file format and raw or compressed (pass-through) video frames and export modules for writing DivX;-), OpenDivX, DivX 4.xx or uncompressed AVI files with MPEG, AC3 (pass-through) or PCM audio. Additional export modules to write single frames (PPM) or YUV4MPEG streams are available, as well as an interface import module to the avifile library. It's modular concept is intended to provide flexibility and easy user extensibility to include other video/audio codecs or file types.

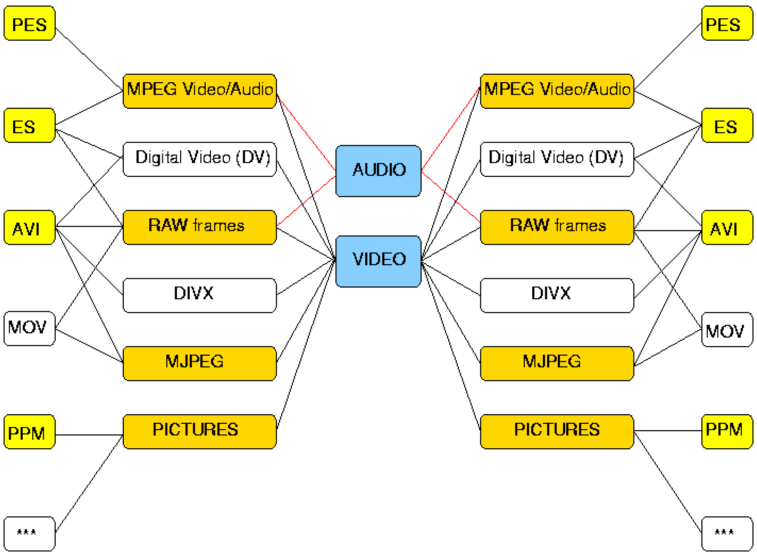

Figure 2.8: **overview of formats and codecs supported by transcode .** Colored boxes are supported without additional packages.

Avilib is a small library to read and write avi files. Avilib does not contain decoder, so the read frame must be decoded by an external decoder. The only case where avilib could be used alone is when video file is Raw. Avilib could also be used if you only want to know some parameters about the video (like size, coder...) included in header. Avilib could not be used alone to replace MPlayer because it couldn't decode a frame. Its features are very limited but the big advantage is it doesn't need external library and it is very easy to link within your own program. So, I don't reject this program and I keep it to build a very small and easy to use Raw avi writer/reader.

[5] See http://zebra.fh-weingarten.de/ transcode/

FFmpeg By looking for a good decoder, I notice all video player on Linux world are based on ffmpeg (mplayer and transcode for example). FFmpeg is a very fast library who can read and write nearly all video and audio formats. It provides an API(Application Program Interface) to handle video files and decode or encode them.

Other libraries are available to work on compressed videos. For example, Xine library is an excellent framework that can read nearly all formats and provide a clean API. However, these libraries are always based on FFmpeg and other small libraries for exotic formats and codecs.

Another solution is to use official libraries, but it is too complex to interface because each of them have a specific API. Moreover, there is not an open source library available for each codec.

2.2.2 FFmpeg

FFmpeg is a complete solution to record, convert and stream audio and video. It includes libavcodec, the leading audio/video codec library. FFmpeg is developed under Linux, but it can be compiled under most OS, including Windows.

Global description

FFmpeg is a project, started by Fabrice Bellard and developed by about twenty peoples, that can code and decode most of video files formats. Another interesting feature is the possibility to work on streamed videos, or build a stream server. This project is under LPGL [6] license and is always on development under CVS [7] server.

FFmpeg is composed of libraries and binaries programs. Libraries can be used independently in your own program to work on multimedia files. Three programs are also provided to show all features and how to use these libraries.

- **ffmpeg** is a command line tool to convert one video file format to another. It also supports grabbing and encoding in real time from a TV card.
- **ffplay** is a simple media player based on SDL and on the ffmpeg libraries.
- **ffserver** is an HTTP (RTSP is being developped) multimedia streaming server for live broadcasts. Time shifting of live broadcast is also supported.

These libraries are used in many projects and in all video player on Linux. They offer the possibility to work on many kind of multimedia files very easily and they are often up to date.

Main projects are using FFmpeg: MPlayer, Xine, VideoLan, XboxMediaPlayer, transcode, Avifile...

FFmpeg structure

FFmpeg is divided into two libraries : **libavformat** and **libavcodec**. First, libavformat is the library containing the file formats to handle (mux and demux code for several formats). It can open and close a file, work on header and index, and read or write a frame. Secondly, libavcodec is the library containing the codecs (both encoding and decoding).

These libraries are structured with a lot of C structures. Only a part of them are available from the API. The first part of my work on FFmpeg was to understand how these structures are compiled and how to use them.

Now, I will detailed the libraries described by the figure 2.9:

Libavformat

- Each format have its own structure with the same functions to read and write header, index, frames... Functions names are stored into a structure, `AVInputFormat` for reading and `AVOutputFormat` for writing.
- `AVFormatContext` is the main structure with API. It means that all its functions will work independently from the format.
  - `AVInputFormat` and (`AVOutputFormat`) links API to specifics format functions.
  - There is an array of `AVStream` (generally, there is 2 streams, one for audio and one for video).

[6] see http://www.gnu.org/licenses/lgpl.html
[7] last version on : cvs -z9 -d:pserver:anonymous@mplayerhq.hu:/cvsroot/ffmpeg co ffmpeg

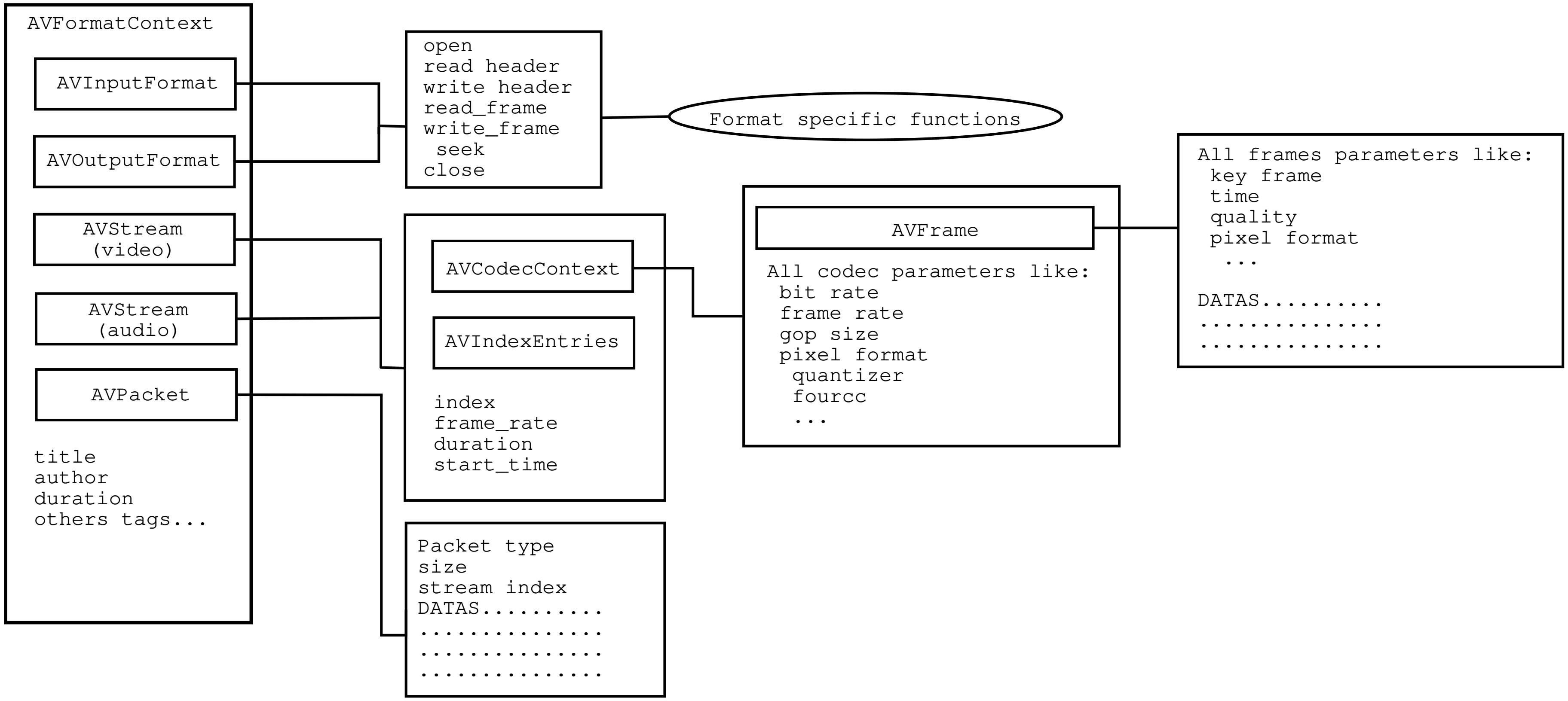

Figure 2.9: FFmpeg libraries structure

  * An `AVCodecContext` is placed here.
  * `AVIndexEntries` is a structure where index entries are stored before writing.
  * This structure also contains all parameters necessary for reading a stream, like bitrate, framerate, duration, size...

- `AVPacket` is return by a read function contains the current compressed read frame and its details.
- **Tags** about the file (author, date...)

Libavcodec

- Each codec have its own structure with the same function to code and decode a frame.
- `AVCodecContext` is the main structure. It contains the codec name and all parameters about the current coding or decoding algorithm.
- `AVFrame` is a structure that could contain a picture on several pixel formats.
- All coding and decoding functions of API are stored in this library. They need an `AVCodecContext` to choose the right algorithm, an `AVFrame` and an `AVPacket`.

Chapter 3

The proposed video package

The libavformat and libavcodec libraries that come with FFmpeg are a great way of accessing to a large variety of video file formats. Unfortunately, there is no real documentation on using these libraries in your own programs (at least I couldn't find any), and the example programs aren't really very helpful either. So, the first step was to understand the internal structure of FFmpeg by developing small programs using main methods. Finally, don't forget FFmpeg is developed in C whereas VisionLib and Torch are developed in C++.

Operational method The first step in the development part was to understand the structure and the principle of VisionLib, Torch3 and FFmpeg. For that, I made some small programs which used a limited number of functions, and I tried to use these libraries at the same time. Then, I modified the VisionLib :

1. Deleting the MPlayer part.
2. Copying FFmpeg sources files in the VisionLib structure (a separate folder for sources files, includes files, binaries files...)
3. Re-building Makefiles in order to compile FFmpeg in its new structure.
4. Doing some tests with my small programs.
5. Creating a new class based on FFmpeg, using the same API.

Finally, when this new VisionLib class was approximately operational, I transposed it into Torch3. A long period of debugging followed where I deleted some mistakes and memory leaks.

3.1 Class hierarchy

In this section, I will try to explain you how I developed my package, with commented examples of source files. Note you can't use directly this source code. To see detailed source files, please see appendix C and D.

3.1.1 VideoFile

`VideoFile` is only a class with virtual methods. That means methods are empty and must be defined in another class (see figure 3.1). It is a good solution to provide an API because users only know the name of the method (read() for example) but in fact this function could be different depending on the context. In this case, there is two possible context : using a compressed video or a raw video. Moreover, it is now very easy to integrate a new video class (audio support, new codec...) into the package.

3.1.2 ffmpegVideoFile

constructor

```
ffmpegVideoFile::ffmpegVideoFile() {

addIOption("width", &width, 720, "output width");
addIOption("height", &height, 576, "output height");
addIOption("bitrate", &bitrate, 1500000, "output bitrate");
addROption("framerate", &frate, 25, "output framerate");
addIOption("gop", &gop, 100, "output gop size");
```

These Torch functions allow the developer to change encoding parameters. The defaults parameters are commonly used at IDIAP.

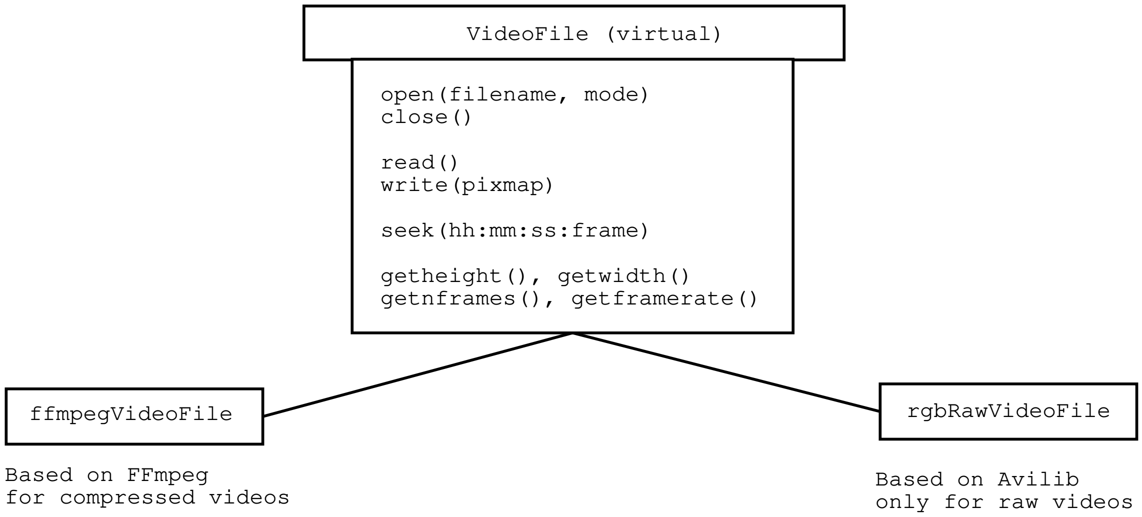

Figure 3.1: Video package structure

```
    n_instance++;
    if (n_instance==1)
    {
        avcodec_init();
        av_register_all();
    }
```

The first step is to register formats and codecs. We only have to do this one time (for example, when you open two files), this is why there is a counter on the number of instances of the class (static variable).

open in read mode Open method allows two parameters (second is optional): filename and open mode (read or write)

```
        av_open_input_file(&ic, filename_, NULL, 0, NULL);
        av_find_stream_info(ic);

        st = ic->streams[0];

        frate = st->r_frame_rate/st->r_frame_rate_base;
        nframes = (int)(frate * st->duration / AV_TIME_BASE);
        width = st->codec.width;
        height = st->codec.height;
```

The function which open the file also extract all header parameters. In case of mpeg file (no header) `av_find_stream_info` extract the first frame to get them. `st` represent the first stream, so the video stream. Parameters like number of frame have to be computed.

```
        codec = avcodec_find_decoder(st->codec.codec_id);
```

```
c = avcodec_alloc_context();
avcodec_open(c, codec);
avframe = avcodec_alloc_frame();
```

Then, we find the decoder with its fourcc identification number, we allocate memory for it, and we open it. We also must allocate memory for the decoded frame.

```
avpicture_alloc(pict,PIX_FMT_RGB24, width,height);
```

Finally, we must allocate memory for a new RGB picture, that will be used when YUV to RGB conversion.

open in write mode Writing initialization is set by two different parts. The first is while opening and the second is while encoding the first frame. Encoding parameters are not necessary known when opening the file, that is why I finish this step after.

```
oc = av_alloc_format_context();
oc->oformat = guess_format(NULL, filename_, NULL);
oc->priv_data = av_mallocz(oc->oformat->priv_data_size);
```

After allocating memory for the format context, a function guess the format with the extension of the filename, and initialize it. We also need allocate memory for the codec. `priv_data` is used for index.

```
st = av_new_stream(oc,0);
c = &st->codec;

codec = avcodec_find_encoder(oc->oformat->video_codec);
url_fopen(&oc->pb, filename_, URL_WRONLY);
```

Then, a stream must be added to the format context. In this case, we there is only one video stream, but an audio stream could be also added.

`avcodec_find_encoder` find the default encoder with the format. For example, if output video is AVI, the default codec will be DivX. Last function only create a new file.

read The read function always return the next picture.

```
while (!frame_created)
{
read_frame:
    av_read_frame(ic, pkt);
    len = pkt->size;
    ptr = pkt->data;
    if (pkt->stream_index != 0)
    {
        av_free_packet(pkt);
        goto read_frame;
    }
```

`av_read_frame` return a packet (`pkt` memory is allocated at each time, so don't forget to free it) which contains one video frame or several audio frames. So we test it and if it is an audio frame, we read a new frame. If a future audio support must be developed, put your audio decoding function here. `len` represent the size of the read packet.

```
        while (len > 0)
            {
                rot = avcodec_decode_video(c, avframe, &got_picture, ptr, len);
                if (got_picture)
                {
                    img_convert(pict,PIX_FMT_RGB24,
                        (AVPicture*)avframe, c->pix_fmt,width, height);
                    frame_created=true;
                    cframe++;
                }
                len -= rot;
            }
```

Now, we can decode the packet. `avcodec_decode_video` do this and return an YUV picture into an `AVFrame`. For IDIAP needs, it is better to convert it to an RGB picture.

Important note : we seeking, frame is already read by the seek function, so we can directly decode it. That is why there is a test of seeking and a jump. (see detailed source code).

```
                av_free_packet(pkt);
```

Finally, you must free the packet at each time.

write

```
    if (cframe == 0)
    {
        c->bit_rate = bitrate;
        c->width = width;
        c->height = height;
        c->frame_rate = (int)frate;
        c->gop_size = gop;

        avcodec_open(c, codec)
        av_write_header(oc);

        outbuf_size = 1000000;
        outbuf = new uint8_t[outbuf_size];
    }
```

This step is the end of initialization and is only do at the first frame. We just give basic parameters and open the codec. Then, we can write the header and start encoding. `outbuf` is a buffer needed by the codec.

```
    avframe->data[0] = pixmap_;

    img_convert(pict,PIX_FMT_YUV420P, (AVPicture*)avframe,
                                PIX_FMT_RGB24,c->width, c->height);

    out_size = avcodec_encode_video(c, outbuf, outbuf_size, (AVFrame*)pict);

    av_write_frame(oc, 0, outbuf, out_size)

    cframe++;
```

Encoder need a YUV picture, so we must convert it. Encoding function just need the codec context (c) and the input picture (pict) and return a coded frame in `outbuf`. Finally, we write the frame.

seek A seek function already exists but is not accurate. In fact, this function seeks on the previous key frame and researchers needs a better tool. I don't give source code here because it is too long, but the following is the algorithm:

1. User wants to seek to HH:MM:SS:FF
2. FFmpeg seek function seeks to the previous key frame (hh:mm:ss:ff)
3. N is the number of frames between HH:MM:SS:FF and hh:mm:ss:ff
4. Read N frames.

3.1.3 rgbRawVideoFile

The `rgbRawVideoFile` class is based an Avilib and can only read or write frames, but is not able to code or decode them. If the frame is a standard RGB picture, you can create directly read it.

The RGB codec is very easy to implement because there is not compression. As a convention, pictures must be horizontally flipped and colors channels swapped (RGB to BGR) before writing.

```
swap_(unsigned char *a, unsigned char *b)
{
  unsigned char tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}

  for(unsigned int h = 0; h < height/2 ; h++)
   for(unsigned int w = 0; w < width ; w++)
     {
     int base_in = w + h * width;
     int base_out = w + (height - h - 1) * width;

     base_in *= 3;
     base_out *= 3;

     swap_(&data_in[base_out], &data_in[base_in+2]);
     swap_(&data_in[base_out+1], &data_in[base_in+1]);
     swap_(&data_in[base_out+2], &data_in[base_in]);
    }
```

Figure 3.2: RGB to BGR convertion and horizontal flip

Avilib function are very limited and always start with `AVI_`.

open in read mode

```
if(strcmp(open_flags_, "r") == 0)
    {
        avifile_input = AVI_open_input_file(filename_, 1);
        readMode = true;

        width = AVI_video_width(avifile_input);
        height = AVI_video_height(avifile_input);
        frate = AVI_frame_rate(avifile_input);
        nframes = AVI_video_frames(avifile_input);

        size = width * height * 3;
        if(pixmap == NULL)
         pixmap = (unsigned char *)
```

```
        allocator->alloc(size * sizeof(unsigned char));
    pixmap = (unsigned char *)
      allocator->realloc(pixmap, size * sizeof(unsigned char));
```

I just use Avilib functions to open the file and get parameters. Pixmap is allocated with Torch functions.

open in write mode

```
else if(strcmp(open_flags_, "w") == 0)
    {
        avifile_output = AVI_open_output_file(filename_);
        writeMode = true;
    }
```

read

```
    AVI_read_frame(avifile_input, (char *) pixmap, &key);
    rgb24_to_bgr24_flip(pixmap, NULL, width, height);
    cframe++;
```

`AVI_read_frame` read the next frame and fill up the pixmap. The integer `key` is set to one if the current frame is a keyframe. In this case, there is only keyframes. Then, we need to convert the picture as described by figure 3.2.

write

```
if(cframe == 0) // This is the first frame
    {
        AVI_set_video(avifile_output, width, height, frate, "RGB" );
    }

    rgb24_to_bgr24_flip(pixmap_, pixmap, width, height);
    int ret = AVI_write_frame(avifile_output, (char *)pixmap, size, 1);

    cframe++;
```

At the first frame, we must initialize Avilib and write the header with `AVI_set_video`. The last parameter is the Fourcc code, here `RGB`[1] or 0.

3.2 Commented example

In the following example, the program read two video, compose them and write the output into a new video file. This simple example show possibilities of my package : work on multiple video files, without external programs. Of course, replace the compose function by your own processing. Read this example like the package manual.

```
#include "ffmpegVideoFile.h"
#include "rawRgbVideoFile.h"

void compose(int width, int height, unsigned char *pixmap1,
            unsigned char *pixmap2, unsigned char *imagecomposite)
```

[1] See http://www.fourcc.org/rgb.php

```
{
    unsigned char *test = imagecomposite;

    for (int h=0;h<height;h++)
    {
        for (int l=0;l<width*3;l++)
            *test++ = *pixmap1++;
        for (int l=0;l<width*3;l++)
            *test++ = *pixmap2++;
    }
}
```

`compose()` simply takes two pictures, and create a new picture with both. The new picture size is (width * 2) x height.

```
int main() {
    int width;
    int height;
    int nframes;
    unsigned char *composed;

    VideoFile *video_in1 = NULL;
    VideoFile *video_in2 = NULL;
    VideoFile *output = NULL;

    video_in1 = new ffmpegVideoFile();
    video_in2 = new rawVideoFile();
    output = new ffmpegVideoFile();
```

First step is to create `VideoFile` pointers and instance them with `ffmpegVideoFile` or `rawRgbVideoFile` depending of your needs.

```
    video_in1->open("video1.mpg");
    video_in2->open("video2.avi", "r");
    output->open("video_out.avi", "w");

    int nframes = video_in1->getnframes();
    int height = video_in1->getheight();
    int width = video_in1->getwidth();

    output->setIOption("width", 2 * width);
    output->setIOption("height", height);
    output->setROption("framerate", 25);
    output->setIOption("bitrate", 20000);
```

Then, use `open` method to open files. First argument is the filename and second is the mode (r for read and w for write). If you read a video, this second argument is optional.

You can get some parameters of your input files. Initialize encoder with `setOption` methods of Torch. By defaults, parameters are the same as meeting room video used at IDIAP, with a good encoding quality .

```
    composed = new unsigned char[height * width * 6];
```

```
for(int i = 0 ; i < nframes ; i++)
{
    video_in1->read();
    video_in2->read();
    compose(width, height, video_in1->pixmap,
            video_in2->pixmap, composed);

    output->write(composed);
}
```

Now, you can start a loop and process pictures like you used to do.

```
video_in1->close();
video_in2->close();
output->close();

delete output;
delete video_in1;
delete video_in2;
delete composed;
```

Finally, you must close all video files and delete objects. If you forget to close your output video, index will not be written and your video file will not be readable.

Conclusion

My proposed video package is now completed and works fine. It permits to read almost all video formats and to write in real-time videos in divx or in others formats. Experiments shown that it is faster than the VisionLib package and use few memory. Of course, developments are still possible, like the audio support or direct grabbing from a digital camera.

This work have been presented during a seminar at IDIAP and seems to be adopted already by some researchers. The video package will be used in several projects on face recognition, gestures detection and sports events detection.

This internship learned me many things. First, by working in an international research institute, I discovered the research world, its working methods, its goals and its people. Doctors, PhD students or simple students like me come from several countries (about 20 nationalities at IDIAP), work on different projects and supply the institute with new knowledge and new ideas. Although it was in a french speaking country, the official language at IDIAP is english, thus I have improved it and discovered new cultures.

By using complex libraries, and by developing a video package, I also improve my knowledge on C++ and video compression methods. Furthermore, thanks to weekly seminar at IDIAP, I had the opportunity to attend presentations on speech and speaker recognition, computer vision and machine learning.

Bibliography

[1] Microsoft Corporation. Avi riff file reference. Internet.

[2] Bernd Fischer. Video formats and compression methods. The AVI Format, September 1999.

[3] MPEG Video Group. Peg-4 video verification model version 2.1. *ISO/IEC JTC1/SC29/WG11*, 1996.

[4] T.Natrajan N.Ahmed and K.R.Rao. Discrete cosine transform. *IEEE Trans. on Computers*, C-23(1):90–93, December 1984.

[5] Winn L. Rosch. *The Winn L. Rosch Hardware Bible.* Que Publishing, 1999.

[6] R.Schfer and T.Sikora. Digital video coding standards and their role in video communications. *Proceedings of the IEEE*, 83:907–923, 1995.

[7] T.Sikora. *MPEG Digital Video Coding Standards.* In Digital Electronics Consumer Handbook, 1997.

Appendix

Appendix A : Used Avilib functions.

Appendix B : Used FFmpeg functions.

Appendix C : ffmpegVideoFilesource code.

Appendix D : rawRgbVideoFile source code.

Used Avilib functions

- `AVI_open_input_file`: Open a AVI file in read mode.
- `AVI_open_output_file`: Open a AVI file in write mode.
- `AVI_close`: Close an opened file.
- `AVI_video_frames`: Return the total number of frames.
- `AVI_video_width`: Return the width size.
- `AVI_video_height`: Return the height size.
- `AVI_frame_rate`: Return the frame rate.
- `AVI_video_compressor`: Return the fourcc code.
- `AVI_read_frame`: Read a frame and return it in vidbuf.
- `AVI_seek_start`: Set the video position to the first frame.
- `AVI_set_video_position`: Set the video position to the specified frame.
- `AVI_set_video`: Initialize the write mode and write the header with size, frame rate and fourcc code.
- `AVI_write_frame`: Write a frame.

Used FFmpeg functions

- Memory functions
  - `av_register_all`: Read the list of available codecs.
  - `avcodec_alloc_context`: Allocate memory for the codec.
  - `avcodec_open`: Open the codec.
  - `avcodec_alloc_frame`: Allocate memory for a coded picture.
  - `avpicture_alloc`: Allocate memory for a decoded picture.
  - `av_alloc_format_context`: Allocate memory for the format (in order to write the index).
  - `av_free_packet`: Free a packet.
  - `av_free`: Free.
- Format functions
  - `av_open_input_file`: Open a file in read mode.
  - `av_find_stream_info`: Read header and provides video parameters.
  - `av_new_stream`: Add a stream to the current input file.
  - `av_read_frame`: Read a frame and return a packet.
  - `av_write_header`: Write the header.
  - `av_write_frame`: Write a frame with a packet.
  - `av_write_trailer`: Write the index and update the header.
- Codec functions
  - `avcodec_find_encoder`: Return the fourcc code of the default encoder associated with the output filename extension.
  - `avcodec_find_decoder`: Return the fourcc code of the input file.
  - `avcodec_decode_video`: Decode a packet and return an YUV picture.
  - `avcodec_encode_video`: Encode an YUV picture and return a frame.
  - `img_convert`: Convert the pixel format of a picture.
  - `avcodec_close`: Close the codec.