Video Analysis Platform

Pablo Flores[1], Federico Lecumberry[1], Pablo Arias[1], Alvaro Pardo[1,2]

[1]IIE, Facultad de Ingeniería [2]DIE, Fac. de Ingeniería y Tecnología

Universidad de la República Univ. Católica del Uruguay

gmm@fing.edu.uy, http://iie.fing.edu.uy/vap

1. Abstract

In this article we present the Video Analysis Platform (VAP) which is an open source software framework for video analysis, processing and description. The main goals of VAP are: to provide a multiplatform system which allows the easy implementation of video algorithms, provide structures and algorithms for the segmentation of video data in its different levels of abstraction: shots, frames, objects, regions, etc, permit the generation and comparison of MPEG7-like descriptors, and develop testing applications for shot detection, shot matching, object segmentation and tracking, etc.

2. Introduction

In recent years the amount multimedia data (text, sound, image, and video) has grown remarkably. In this context, several problems arise regarding analysis, compression, transmission, manipulation, search and organization of such media. The main goal of VAP project is to study and develop useful tools for the analysis, compression, transmission and search of video sequences. With this goal in mind, we worked on the design and development of data structures suitable for describing video sequences and their contents.

Due to its enormous amount of data, video analysis imposes important requirements in memory and computational resources; therefore it is mandatory to be very conservative in their use. This restriction applies also to the overhead that is introduced in the processing due to repetitive data structure construction and software procedures invocation.

Another important issue when working with video data is the fact that video is usually compressed. That means that we need to decode it every time we want to process it. Unfortunately, there are a lot of different codecs. Although the process of decoding may seem a simple task, it is a time-consuming one when you do not have the proper tools at hand. One of the aims of VAP was to make the process of video decoding transparent to the end user and to the algorithms programmed. In this way, if the system supports the video codec, you just need to connect to the proper *Video Reader* to access the video information.

Finally, inspired in the MPEG7 standard [6], VAP includes a description module that handles XML to provide content description capabilities.

In summary the main goals for VAP design were to:

- Develop a multiplatform framework for easy and efficient implementation of video algorithms.
- Allow the segmentation of video data in its different levels of abstraction: shots, frames, objects, regions, etc.
- Generate and compare MPEG7-like descriptors.
- Develop testing applications for shot detection, shot matching, object segmentation and tracking, etc.

In the next section some concepts and constrains of video processing are introduced. In section 4, the approach to video processing applied to the development of VAP is presented. Section 5 presents the Architecture overview of VAP; this section contains many examples to show the use of VAP. Finally in section 6 we present the conclusions and the lines of future work.

3. Video processing

Video is processed to extract information from it. For instance, shot detection (the partition of a video into continuous scenes) may be done by analyzing the similarity of subsequent frames pairs. This is an example of event detection, which could be accompanied with some outputs, like sounding a beep or annotating the frame number in a file. In other cases, like in noise reduction, the objective is to obtain a modified version of the input video. A general video processing method schema is shown in Figure 1. The process can be categorized in five stages: Acquisition, Extraction, Analysis, Representation and finally Recognition.

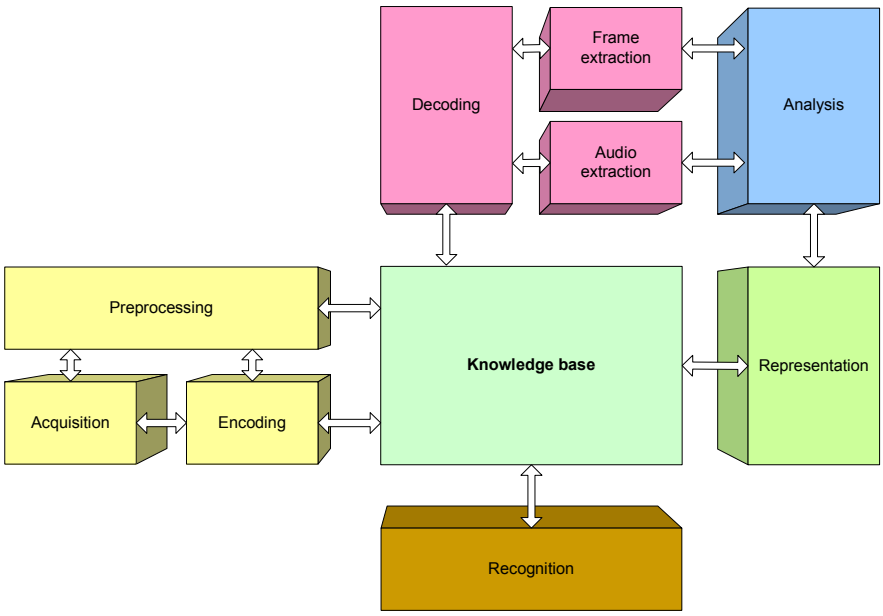

Figure 1: Video processing

First of all, video has to be acquired by a camera or artificially created. Regardless the technologies used for acquisition or generation, audio and image get digital. In the next step coding algorithms are applied to compress the data; most times modifying once more the original contents. Coded information can then be stored, for further use or for immediate transmission.

The opposite process consists in decoding the video to extract frames and audio information. Once decoded, the frames are the main working unit for video analysis.

The analysis process consists in executing different algorithms to the video image and audio information.

Some algorithms results may be used by other ones forming an algorithm pipeline. Each algorithm could represent different abstraction level needed for event detection and video description. In the case of shot detection algorithms, they may calculate individual frames histograms and then use them to obtain accumulative histograms useful for segment description. Both results may be used to guess whether shot has changed or not having two different views of the scene, one of them instantaneous and the other one accumulative.

It is important to distinguish between progressively accessing the video from a file or accessing it from a streaming source. First of all, not all the algorithms access video information sequentially. For example, some shot detection algorithms look ahead in video windows consisting of sets of frames, for example to detect fade-in and fade-out effects [7]. Within a file, algorithms can go all

around the video looking for the requested information, but the only way for going back and forth when receiving a streaming is having a buffer and waiting until the requested frame comes into it. Therefore, working with small windows of frames is mandatory for effective live analysis of streamed video. In addition, algorithms have to be computationally lightweight enough in order to be run without having to leave frames out.

The results of every algorithm can be stored in temporal memory structures, or can be persisted for further uses. This process is called representation.

The MPEG7 standard [6] is conceived for long-term persistence of analysis results and can be applied to all kinds of multimedia material. It establishes extensible rules for XML-based representations of metadata and defines a binary format to be attached to multimedia elements. Unfortunately, MPEG7 is not widely used, and there is not much literature about practical implementations and applications. In addition to textual descriptors, other types of persistence may be needed, like modified video, for instance highlighting tracked objects or applying filters, or selected frames and sound sequences.

In the recognition process, stored or streamed video is taken as input and an algorithm tries to detect sets of properties in the sequences which are interpreted as events. This is the case of many challenges in video analysis, for instance detecting the presence of an actor in a film, a goal in a soccer game or an intruder in a security camera transmission.

Persistent information extracted from analysis processes can be used for recognition algorithms in which features described from an initial video have to be detected in other videos. Particular cases of this are signature finding algorithms, like the commercials detection ones. In these cases, an information set – the signature – is extracted from a video fragment, and then it can get matched within other videos, using some distance measure. These algorithms usually use MPEG7 descriptors for the signature applied to a pre-established video fragment and feature set.

4. VAP's approach for video processing

VAP is a software framework developed with the goal of making the development of video processing software easy. With this purpose, it has to make some trade-offs between execution speed, memory usage efficiency and ease of programming.

It is not a library; developers who want to use VAP have to learn its architecture and how to work upon it. This approach permits VAP to automate many common tasks of video processing, having the drawback that its learning curve may be somehow steper.

As a starting point, VAP provides a *Video Reader*, which allows the access to coded video files and connects it with different toolkits for image processing, mathematical and statistical operations etc.

To accomplish the objective of efficiently handling the large amounts of video data keeping computation and memory requirements under control we decided the implementation of *Descriptors*. *Descriptors* are lightweight data structures to hold analysis results of a given video unit. For example, we can have *Frame Descriptors*, *Shot Descriptors*, *Region Descriptors*, etc.

As it was discussed above, in several video processing and analysis applications we need to access its descriptors several times. For instance, when computing the interframe histogram difference, dh(frame_a, frame_b), we may need the histogram of a given frame several times. Furthermore, if we are running several algorithms at the same time, it may happen that more than one algorithm uses the same description. This could lead to the same descriptor being computed and stored several times. To solve this problem, we included a descriptor cache to hold the last computed descriptors. Then, once a descriptor's cache is created, analysis results will be reused, saving computation time.

To provide tools for making MPEG7 and other kinds of persistent descriptions, VAP integrated XML generation and parsing tools in a subset we called MP7. This subset provides the tools for

making a duality between memory-stored descriptors and XML descriptors that can be saved and restored.

VAP is conceptually divided into three subsystems; two of which have been explained above: Descriptors subsystem, MP7 subsystem and Analyzers subsystem. The last one provides the tools for automating video analysis with specific objectives, like shot detection and signature detection.

5. Architecture overview

VAP is completely developed using C++ language. Since it was conceived as an open-source system, we used some pieces of open source libraries. The decoding and coding of video data is performed with the library ffmpeg [3] using a customized version of the C++ wrapper fobs [4]. Ffmpeg is able to decode a wide range of codecs and for this reason was used. In some cases we use the fobs wrapper while in others, such as for color conversion and reading in YUV format we use ffmpeg functions directly. All the video input/output methods are encapsulated in two classes, *VideoReader* and *VideoWriter*.

In order to give VAP good image processing capabilities we decided to provide a interconnection module with ITK (Insight Toolkit of Kitware) [1,2]. ITK is a well known and extend image processing library, mainly focused in medical images treatment, that provides a comprehensive tools set for image analysis.

VAP has four decoupable software units, which are represented in figure 2: *vapCore* which contains the central elements of the system. *vapMP7* contains MPEG7-like functionalities and descriptors; *vapITK* provides an interface to integrate ITK image processing functionalities; and finally *vapAlgorithms* implements algorithms for video processing based on previous units.

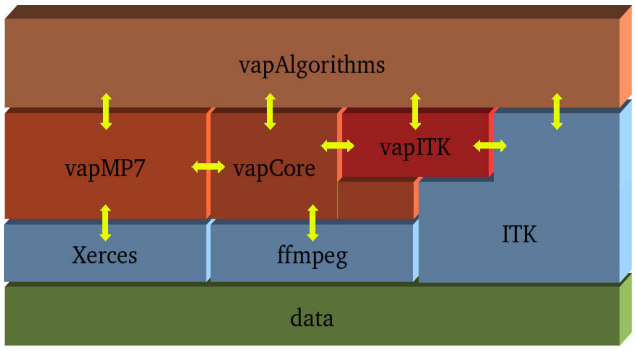

Figure 2. VAP software units and interactions

vapCore

Figure 3 overviews *vapCore's* design. *vapCore* provides access to video data through ffmpeg library. The class *VideoReader* encapsulates this logic allowing easy access to video data. This class uses the singleton design pattern; therefore there is only one input video in the system which is easily accessed. Through the *VideoReader* it is possible to access the raw data corresponding to frames, identifying each one by its *FrameNumber*, an integer value.

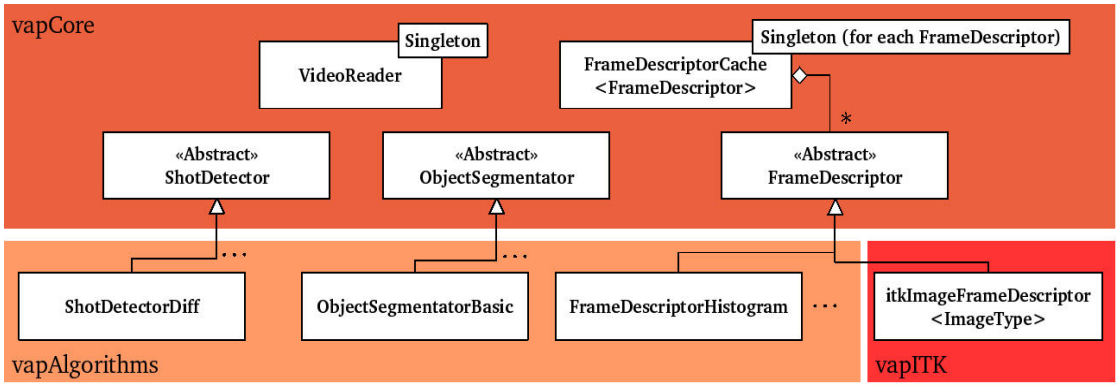

Figure 3: Main elements of vapCore.

In Source 1 we show how to use the video reader to obtain a pointer to the decoded video frame. First, in line 1, we declare the input file. In line 2 we declare the *Frame Number*. In line 3 we get the *VideoReader* instance and assign it to *myVideoReader* pointer. Finally, we use *GetRGB()* to obtain the raw RGB values of the first frame (f=0) in the video file. As we can see accessing the video data is very easy.

```
1- vap::VideoReader::SetVideoFileName("filename.avi");
2- vap::FrameNumber f = 0;
3- vap::VideoReader* myVideoReader = VideoReader::GetInstance();
4- unsigned char* data = myVideoReader->GetRGB(f);
```

Source 1: VideoReader example.

The main concepts introduced by *vapCore* correspond to the *Descriptors* and their corresponding caches. *Descriptors* run algorithms over a part of the video to get the desired information and store it. The procedure for analyzing a video consists on creating one *FrameDescriptor* for each frame, or a *ShotDescriptor* for continuous sequences of frames. For that end, two interfaces are defined. Each descriptor has to inherit from its corresponding interface class, *FrameDescriptor* or *ShotDescriptor* (see Figure 3). Figure 4 shows an example on how different *FameDescriptors* hold information obtained from video frames. It is important to notice that each frame has a corresponding *FrameDescriptor* object for each type of *FrameDescriptor* (in the example, *FrameDescriptorHistogram* and *FrameDescriptorMeanColor*). The example also shows how different *ShotDescriptors* can refer to different framesets depending on the shot detection algorithm that was used.

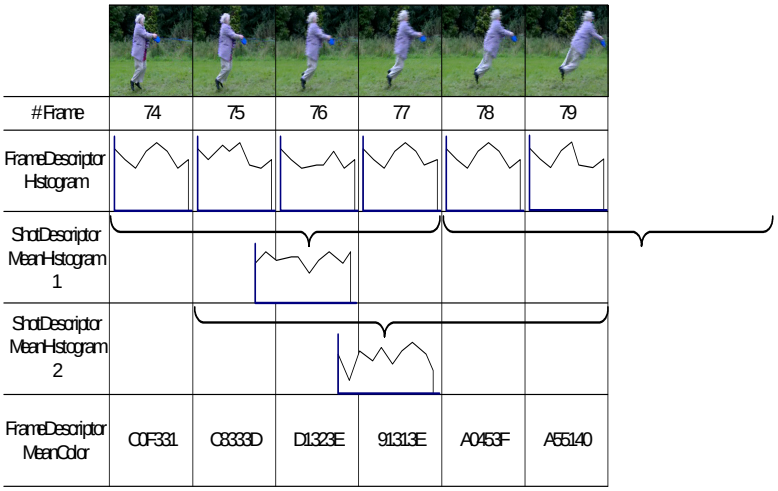

Figure 4: Use of FrameDescriptors.

The *FrameDescriptor* interface only defines one method to be implemented, *ComputeFrame*, which receives a *FrameNumber*. When it is called, implementations can access raw video data or other descriptors. This information can later be accessed by suitable "getter" methods implemented. An example is the *ItkImageFrameDescriptor* (see Figure 3) together with the *FrameReader*. For each frame that it is computed, *FrameReader* accesses the raw video data through the *VideoReader* to generate an *itkImage* (an image stored in the format handled by the ITK library). FrameReader inherits from *ItkImageFrameDescriptor* (and therefore also from *FrameDescriptor*). Then, *FrameReader* is a special case of *FrameDescritor*. Other descriptors may run algorithms taking advantage of the new functionalities available, calling its *GetImage()* method to obtain a pointer to a ITK image.

In Source 2 we present an example of the use of *FrameReader* to connect *VideoReader* with ITK. In line 3 we declare the ITK image type. Once we defined the ITK image type we define, based on it, the *FrameReader* type in line 4. When defining the *FrameReader* type we also declare the type of output, in this case RGB. Then, in line 5, we define and create a cache of frame *FrameReaders* (remember that *FrameReader* is a Descriptor). For details on caches see below. Finally, in line 6, we read the first ten frames from the video file and store the result in *itkImage*. Once we have the *itkImage* we can proceed to use all ITK functionalities.

```
1- vap::VideoReader::SetVideoFileName("filename.avi");
2- vap::FrameNumber f;

   // Define ITK image type. In this case a 2-dimensional RGB image.
   // Each color channel is stored as a unsigned char.
3- typedef itk::Image<itk::RGBPixel<unsigned char>,2> TColorImage;

   // Define FrameReader type. It will use a TColorImage as defined above.
   // The output of the VideoReader will be stored in RGB format.
4- typedef vap::FrameReader<TColorImage,RGB> TColorFrame;

   // Pointer to a FrameDescriptorCache of FrameReader's.
5- FrameDescriptorCache< TColorFrame >* videoFrame;
   _videoFrame = FrameDescriptorCache< TColorFrame >::Instance();

   for(f=0;f<10;f++) {
6-      TColorImage::Pointer itkImage = videoFrame->GetData(f)->Getimage();
        ...
   }
```

Source 2: Example on how to use *FrameReader* to obtain an ITK image.

ShotDescriptors apply to different kinds of frames sequences. To be strict, they do not necessarily have to describe shots, as the only requirement the system imposes is to hold information relative to a set of frames identified by two particular ones, usually the first and the last one. How does the implementation know which frames to process? For this purpose, a *ComputeFrame* method is defined in the interface, just like it happens in *FrameDescriptor*. In addition to this, *ShotDescriptors* have to implement a *ComputeShot* method, which receives as parameters the two aforementioned *FrameNumbers*. With this information and the information already obtained during consecutive calls to *ComputeFrame* method, *ComputeShot* can obtain a summary description of the analyzed group of frames. We are going to present an example of this together with the description of the MPEG7 subsystem.

To reduce computational costs, descriptors can be used as *flyweight* objects, having as *intrinsic* data the ones obtained after running algorithms and as *extrinsic* data the *FrameNumber* and, in

consequence, other associated information. Then, descriptor objects can be reused, for instance when the information obtained from a past frame is no longer needed. As it will be detailed later, this concept is widely used by caches. Finally, as descriptors define a family of interchangeable algorithms, they follow the *strategy* design pattern.

To optimize the implementation of several algorithms over the same frames a Cache was created. The class *FrameDescriptorCache* allows to temporally save last computed *FrameDescriptors*. In this way it is possible to have universal access to already extracted information that can be re-used by other algorithms. Figure 4 shows this need clearly. For *FrameDescriptorMeanColor* it may be cheaper to obtain its value looking at the histograms, which are also used by *ShotDescriptorMeanHistogram*.

FrameDescriptorCache was designed with the following requirements: a) have different caches for different *FrameDescriptors*; b) keep temporally the last descriptors calculated (most algorithms analyze frames within small windows of frames); c) be able to create and access descriptors in random order; d) be as lightweight as possible. This led us to two main design decisions: 1) Make a templetized singleton class *FrameDescriptorCache*; it means there can be only one instance of the cache for each type of *FrameDescriptor* and 2) keep *FrameDescriptors* in a fixed-length circular array.

Now we explain how *FrameDescriptorCaches* works. Each one is templetized on the specific *FrameDescriptor* it will cache, having a singleton access method *Instance()*. For example, a pointer to a cache of *FrameDescriptorHistogram* can be obtained from anywhere in the system calling *FrameDescriptor<FrameDescriptorHistogram>::Instance()*. If the cache was already used, its pointer is returned, otherwise it is created. The array has two fields: *FrameNumber* and *Descriptor*. The latter one is a pointer to the specific class of descriptors to be cached. Initially, all *FrameNumber* fields have an *UNDEFINED* value. When access to the descriptors is requested through the method *GetFrameDescriptor(f)*, the access position *Module(f, CACHE_LENGTH)* is calculated. If the corresponding *FrameNumber* does not match f, then it is updated as well as the pointed descriptor.

Figure 5 shows an example of the usage of *FrameDescriptorCache<SpecificFrameDescriptor>* with *CACHE_LENGTH=16*. In this case, the access to descriptors of frames 0, 1, 15, 18 and 19 has been requested. Maybe before requesting the access to frame 18, frame 2 was requested, but in this case its value was overwritten. If access to frame 2 is requested again later, it will have to be calculated again. This is not a drawback if we work within windows of frames smaller than *CACHE_LENGTH*. In addition to this, results which are known to be needed later can be retained easily.

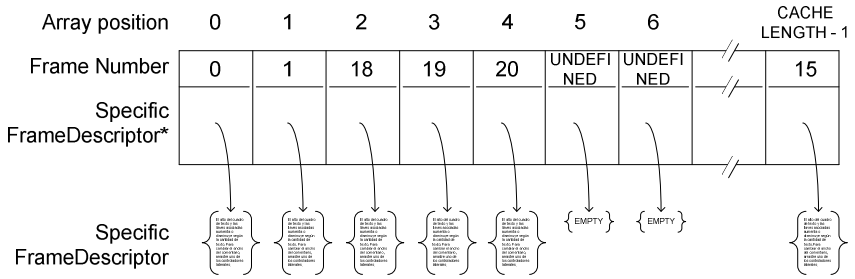

Figure 5: Example on FrameDescriptorCache<SpecificFrameDescriptor>

Considering the nature of video processing algorithms, this design brings some advantages: 1) seeking the access position in the cache is extremely fast; 2) specific descriptors can be created, one

for each of the array positions, all at once, and then they can be reused fast, without having to create structures again.

Source 3 shows how use caches to implement a shot detection based on histogram differences. In line 1 we declare and get the instance of the cache of *FrameDescriptorHistogram*. The in line 2 we request the histograms of frames f-1 and f and compare them using the method *Distance()* of *FrameDescriptorHistogram* class. With this example we can show the simplicity of the resulting code. Once we declared the use of the video *filename.avi* and the *FrameDescriptors* needed, the rest of the code is very simple and easy to read. We don't need to worry about low level details at this stage. So, in each step of the development we concentrate on one specific problem. The framework provides the access to the video data and some descriptors. The used can easily then create its own descriptors and use them in a similar way as described in Source 3.

```
   vap::VideoReader::SetVideoFileName("filename.avi");
   vap::FrameNumber f;

// Get the needed cache. As it was not previously required, the cache array
// will be created with empty instances of FrameDescriptorHistogram,
1- FrameDescriptorCache<FrameDescriptorHistogram>* fdch =
              FrameDescriptorCache< FrameDescriptorHistogram >::Instance();

   for (f=1; f<maxFrames; f++)
     {
2-     if(fdch->GetFrameDescriptor(f-1)->GetItkHistogram().
     Distance(fdch->GetFrameDescriptor(f)->GetItkHistogram()) > threshold)
     {
          cout << "Shot has changed in frame " << f << endl;
        }
     }
```

Source 3: Descriptors usage with caches

Figure 6 shows the whole descriptors subsystem, which provides as core elements the caches and defines pure interfaces for *FrameDescriptor* and *ShotDescriptor*. In a second level, where abstract classes which implement some functionalities, are defined, *ItkImageFrameDescriptor* was programmed for descriptors which store results as ITK images. The lower level represents concrete descriptors instantiated in the *vapAlgorithms* unit. Finally, *vapCore* includes some specific-purpose base classes to detect shots (*ShotDetector*), segment objects (*ObjectSegmentator*) and look for commercials and other fashions of signatures (SignatureFinder).

vapItk

VAP gives the possibility to use all the tools provided by ITK to process image data. For that end a particular *FrameDescriptor* was implemented (*ItkImageFrameDescriptor*) which converts raw data into an ITK image that can be accessed through the cache like any other descriptor. In the Appendix we present an example that uses vapItk to convert a color image to greyscale and writes it to disk.

vapAlgorithms

This is the upper layer of the system and can use any of the functionalities previously mentioned to implement video analysis algorithms in an easy way.

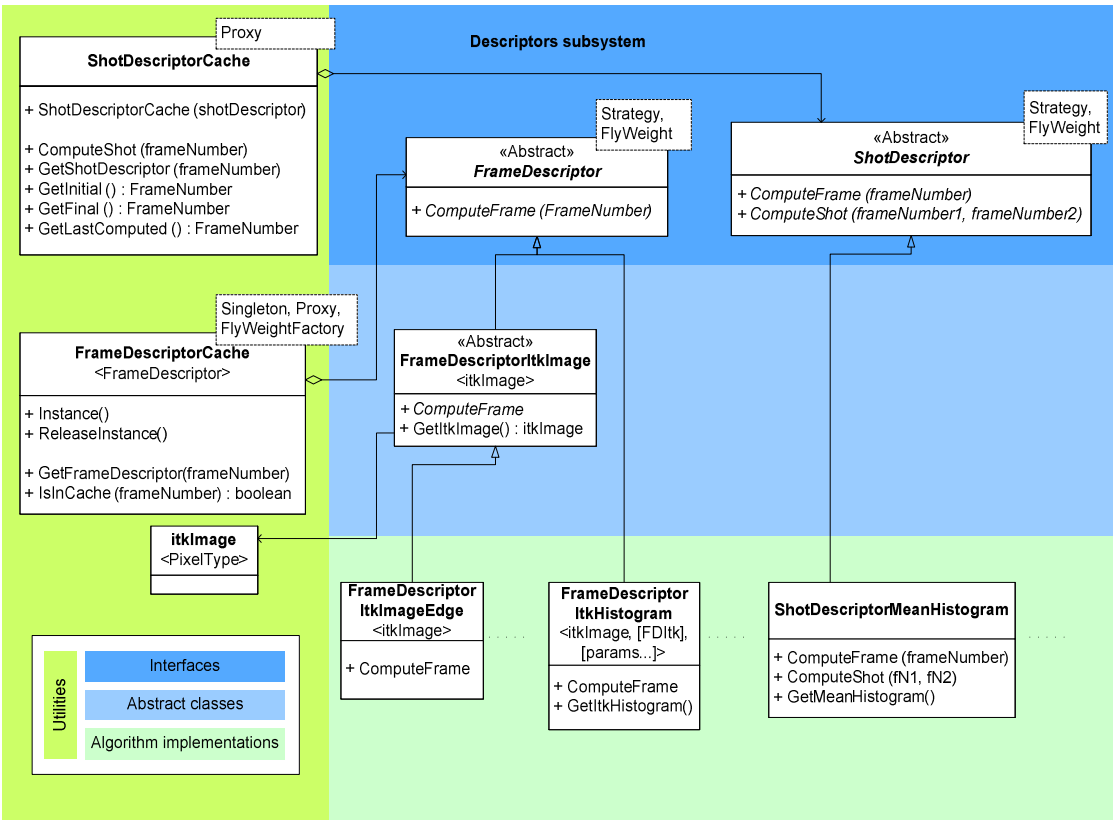

Figure 6: Descriptors subsystem design

vapMP7

This unit complements *vapCore* implementing the logic of generation of XML descriptors extracted from the video, using the Xerces library [5]. We called them *MP7* descriptors, as they are inspired in the MPEG-7 standard. Nevertheless, the platform has all the elements to be adapted to other type of XML formats. As it was previously introduced, the goal of the *MP7* subsystem is to make possible the persistence of already computed descriptors. For instance, if we get 12-bins histogram descriptors for frames and shots, the result would be as shown in Source 4.

```
<MP7>
 <ScalarHistogram FrameNumber="0">
  <Component Channel="R">126 8 13 19 35 34 99 464 952 1503 2172 4588
     </Component>
  <Component Channel ="G">5 18 21 42 97 163 263 294 283 158 102 175
     </Component>
  <Component Channel ="B">30 94 99 173 232 138 396 351 371 345 414 9
     </Component>
  </ScalarHistogram>
…
 <ShotMeanScalarHistogram FramesConsidered="15" Initial="0" NumberOfFrames="13">
  <Component Channel ="R">107 11 10 11 22 29 24 20 15 30 68 145
     </Component>
  <Component Channel ="G">4 14 20 35 52 86 137 168 285 348 365 482</Component>
  <Component Channel ="B">49 96 91 561 826 991 702 493 332 333 49 75
     </Component>
  </ShotMeanScalarHistogram>
</MP7>
```

Source 4: Sample MP7

The system defines a duality between *FrameDescriptor* and *MP7FrameDescriptor*, and between *ShotDescriptor* and *MP7ShotDescriptor*. This means that if a *MP7* descriptor is properly programmed, it will persist correspondent information of the descriptor. Conversely, reading the *MP7* descriptor, the pure descriptor can be reconstructed which is useful for signatures finding, as it will be described later.

XML descriptors can be organized in different structures, often one contained inside another, forming a tree structure. The class *MP7Tree* is defined to store and navigate this structure of MP7 descriptors, and it makes possible its persistence to disk.

In construction time, *MP7Tree* uses an auxiliary class *MP7TreeCreator*, or a child of it, to set up the *MP7* descriptors to include: *MP7FrameDescriptors* as well as *MP7ShotDescriptors* can be subscribed. Then it can be indicated the *MP7* of which frames and shots to be computed and stored in the tree.

In the example shown in Source 4, we use a *ShotDetector* to determine the segmentation of the video into shots which in turn will be described in *MP7*. *MP7TreeCreator* simplifies the *MP7* tree construction, also encapsulating its organizational aspects such as the elements order and other formalities to be accomplished. In line 1 we declare a *MP7TreeCreator*. In lines 2 and 3 we construct the frame and shot descriptors. Lines 4 and 5 attach the descriptors to the *MP7Tree*. Then, for each frame we call the method *ComputeFrame()* of the *MP7Tree*. When a shot change is encountered, the partial information that has been collected so far it is processed with *ComputeShot()*. Once again, the framework developed allows developing video analysis methods easily.

```
    vap::VideoReader::SetVideoFileName("filename.avi");
    vap::FrameNumber f;

   // Shot detector to be used
   vap::ShotDetector shotdetector;

   // MP7 tree creator
1- vap::MP7TreeCreator tc;

   // MP7 descriptors to be computed
2- MP7ShotDescriptor* mp7sd = new MP7ShotDescriptor();
3- MP7FrameDescriptor mp7fd = new MP7FrameDescriptor();

   // Set up the descriptors to be used in the new MP7 tree
4- tc.Include(mp7sd);
5- tc.Include(mp7fd);

   // Process the video detecting shots and building the MP7 tree
   for(f=0; f<lastFrame; f++)
   {
6-    tc.ComputeFrame(f);
      shotdetector.AnalyzeFrame(f);
      // If there is a shot change in this frame, close the shot and store
      // its description calling ComputeShot().
      if(shotdetector.ShotChange()){
7-        tc.ComputeShot(f);
    }
   }
   tc.CloseTree();
   cout << tc.GetTree()->GetString();
   delete tc.GetTree();
```

Source 5: *MP7Tree* creation

MP7 trees can be restored later. This is useful for contents search and other techniques of video comparison. Commercial detection, for instance, is a particular case of it, in which describing aspects of a piece of video are searched into new videos. With this purpose, we developed a *SignatureFinder* class, which defines the main aspects of this kind of searches: First, a piece of video y described with MP7, saving it (we call this description the signature of the video). Then, the MP7 tree can be restored and corresponding descriptors reconstructed. Finally, the same classes of descriptors can be obtained from a new video and try to match them with the signature ones. Signature finding process is not straightforward, as some relaxation has to be made in the comparison algorithms.

6. Conclusions and future works

In this article we presented the development of a software framework for video processing and analysis. The system is very easy to use and it is efficient in memory and CPU use[1]. It also integrates the capabilities of ITK making it very attractive. So far we implemented some testing applications which showed the benefits of the system. It was successfully tested in MS Windows with Visual C++ 6.0 and in GNU Linux with GCC 3.3.3-7. We believe the system can be useful to develop video analysis algorithms prototypes, proving good computational performances and small memory requirements. For future work we are going to implement more testing applications to obtain a stable version of the software that will be released as open source. Also, video streaming is a line for future developments.

Appendix Example - Working with pixels

Here we present a small example using the proposed framework to read a color frame from a video sequence, convert it to gray values (a way to process each pixel or a region of the frame) and save it to disk. We use the ITK library to access and handle the frames. First, the image format can be define by selecting the types of pixels to be used, in this case a three-component pixel (`itk::RGBPixel`) and a one-component pixel (gray pixel). Also pointers to these images are defined.

```
typedef itk::Image<itk::RGBPixel<unsigned char>,2> TColorImage;
typedef itk::Image<unsigned char,2> TGrayImage;
TColorImage::Pointer _frameImage;
TGrayImage::Pointer _grayImage;
```

In the same way as the example presented in Source 2, we use *FrameReader* to obtain a pointer to the ITK image `_frameImage`.

```
typedef vap::FrameReader<TColorImage,RGB> TColorFrame;
vap::VideoReader::SetVideoFileName ("filename.avi");
vap::FrameNumber f = 0;
FrameDescriptorCache< TColorFrame >* _videoFrame;
_videoFrame = FrameDescriptorCache< TColorFrame >::Instance();
_frameImage = _videoFrame->GetData(f)->GetImage();
```

In order to work with the pixel values of the image is necessary to access to the pixel level of the image, we can use the iterators provided by ITK. Each iterator is associated with an image and defined in a particular region, in this case the whole image. The image can be processed in an intermediate level using the regions defined with the iterators (For details see [1]).

[1] We don't present results on CPU usage in this paper due to lack of space. However, it is easy to verify that the overhead of the systems is very low.

```
typedef itk::ImageRegionIterator< TColorImage > TColorImageIterator;
typedef itk::ImageRegionIterator< TGrayImage > TGrayImageIterator;
TColorImageIterator iterColor(_frameImage, _frameImage->GetRequestedRegion());
TGrayImageIterator iterGray (_grayImage, _grayImage->GetRequestedRegion());
```

Using both iterators each pixel in the region of interest is accessed sequentially in the color image, processed and set in the gray image.

```
TColorImage::PixelType colorPixel;
TGrayImage::PixelType grayPixel;
for (iterColor.GoToBegin(), iterGray.GoToBegin(); !iterColor.IsAtEnd();
                                                   ++iterColor, ++iterGray) {
  colorPixel = iterColor.Get();     // Get the pixel RGB values...
  grayPixel = (colorPixel[0] +
               colorPixel[1] +
               colorPixel[2])/3;    // ...compute the gray level...
  iterGray.Set(grayPixel);          // ...and set the pixel.
}
```

Finally, to save the gray image we use the writer filter provided with ITK

```
typedef itk::ImageFileWriter< TGrayImage >::Pointer grayImageWriter;
grayImageWriter->SetFileName("grayImage.png");
grayImageWriter->SetInput(_grayImage);
grayImageWriter->Update();
```

7. References

[1] Insight Toolkit. www.itk.org

[2] Ibanez, Schroeder, Ng, Cates. The ITK Software Guide. Kitware, Inc. ISBN 1-930934-15-7.

[3] www.ffmpeg.org

[4] fobs.sourceforge.net/

[5] xml.apache.org/xerces-c/

[6] B. S. Manjunath (Editor), Philippe Salembier (Editor), Thomas Sikora (Editor), Phillipe Salembier (Editor). Introduction to MPEG 7: Multimedia Content Description Language. John Wiley & Sons, 2002. ISBN: 0471486787.

[7] A. Hanjalic: Shot-Boundary Detection: Unraveled and Resolved?, IEEE Transactions on Circuits and Systems for Video Technology, February 2002