Media handling for visual information retrieval in VizIR

Horst Eidenberger[*]
Vienna University of Technology,
Institute of Software Technology and Interactive Systems,
Favoritenstrasse 9-11, 1040 Vienna, Austria

ABSTRACT

This paper describes how the handling of visual media objects is implemented in the visual information retrieval project VizIR. Essentially, four areas are concerned: media access, media representation in user interfaces, visualisation of media-related data and media transport over the network. The paper offers detailed technical descriptions of the solutions developed in VizIR for these areas. Unified media access for images and video is implemented through class *MediaContent*. This class contains methods to access the view on a media object at any point in time as well as methods to change the colour model and read/write format parameters (size, length, frame-rate). Based on this low-level-API class *VisualCube* allows accessing spatio-temporal areas in temporal media randomly. *Transformer*-classes allow to modify visual objects in a very simple but effective way. Visualisation of media object is implemented in class *MediaRenderer*. Each *MediaRenderer* represents one media object and is responsible for any aspect of its visualisation. In the paper examples for reasonable implementations of *MediaRenderer*-classes are presented. Visualisation of media-related data is strongly connected to *MediaRenderer*. *MediaRenderer* is to a large extent responsible for displaying visual panels created by other framework components. Finally, media object transport in VizIR is based on the Realtime Transfer Protocol (for media objects) and XML-messaging (for XML-data).

Keywords: Visual Information Retrieval, Content-based Image Retrieval, Content-based Video Retrieval, Media Handling, Media Processing, Video Visualisation, Java Media Framework

1. INTRODUCTION

This work describes how the handling of visual media is implemented in the visual information retrieval[1, 9] (VIR) project VizIR. The VizIR project[3, 2] aims at developing and collecting components for an extendible general resource framework for VIR research. The major components of VizIR are two class frameworks: one for VIR querying (including feature extraction, query definition, etc.) and one for VIR user interface design. Media handling in VizIR can be split in four areas: (1) media access, (2) media visualisation and time representation, (3) visualisation of media-related data (metadata, descriptions, etc.) and (4) media transport over the network.

In a modern VIR system media access must be unified for images and videos. Even though both types of media are conceptually similar, there are enough differences (e.g. the temporal dimension, colour models) that make unification on the technical level a non-trivial problem. Representation of the temporal dimension of video is a well-known research problem. The solutions implemented in VizIR are based on the 3D-user interfaces used for querying. They make use of the classes for unified media access. Visualisation of media-related data is a problem that occurs in various situations: extracted features (e.g. MPEG-7-descriptions) have to be visualised to give the user an impression of what he is querying on and metadata have to be visualised in a browsable, clearly structured way. In VizIR, each component is responsible for the visualisation of "its" media-related data. VizIR offers a distributed environment for VIR. Therefore, not only media objects have to be transferred over the network but, as well metadata and XML-query descriptions. Media transport in VizIR is based on the Realtime Transfer Protocol (RTP).

The paper is organised as follows. Section 2 gives background information on the VizIR project and media handling in other VIR prototypes. Section 3 addresses the problem of media access (on the technical and the semantic level). Section 4 sketches the methods implemented in VizIR for media (especially video) representation in user interfaces. Section 5 is dedicated to visualisation of media-related information. Finally, Section 6 explains how network transport of media data and any kind of metadata is implemented in VizIR.

[*] eidenberger@ims.tuwien.ac.at; phone 43 1 58801-18853; fax 43 1 58801-18898; www.ims.tuwien.ac.at

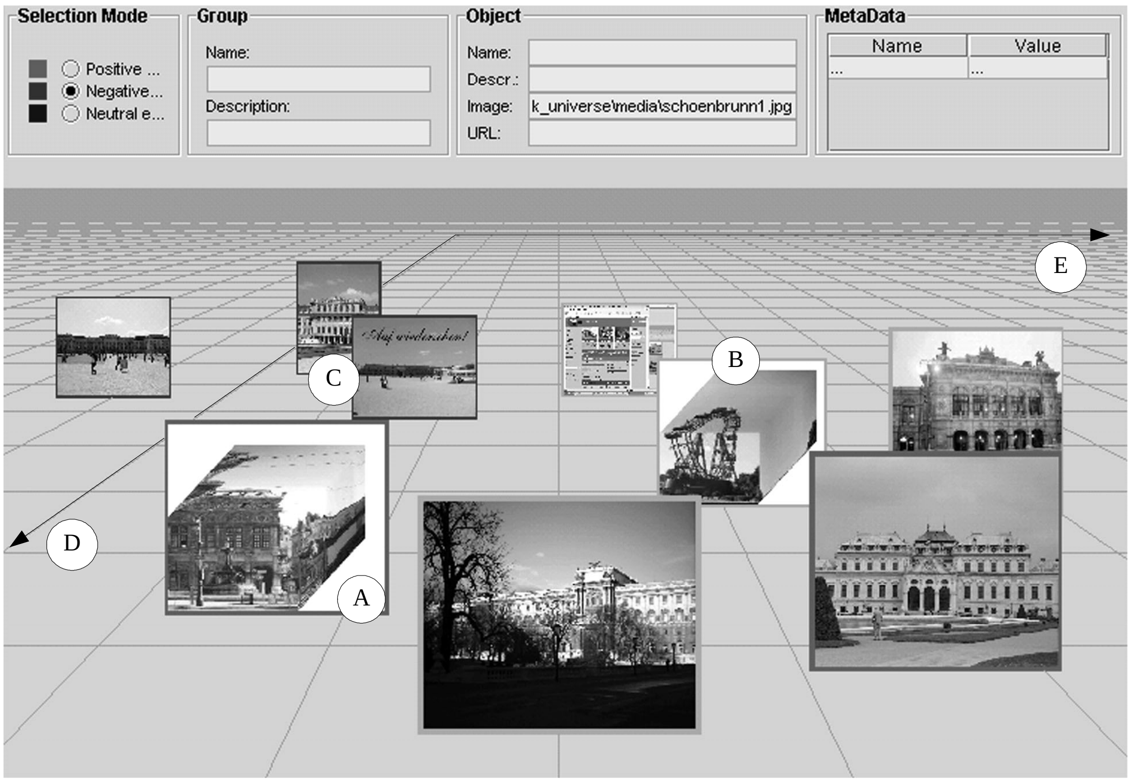

Figure 1: Screenshot of the VizIR user interface.

2. BACKGROUND

2.1 The VizIR project

The overall goal of the VizIR project is the implementation of a modern, open *class framework* for content-based retrieval of visual information. The major result should be a workbench-like implementation platform for further research on successful methods for automated information extraction from images and video streams, the definition of similarity measures that can be applied to approximate human similarity judgment and new, better concepts for the user interface aspect of visual information retrieval, particularly for human-machine-interaction, for query definition and refinement, and video handling. On top of this framework working prototypes are implemented that are fully based on the visual part of the MPEG-7-standard for multimedia content description.

Querying in VizIR is done in a 3D environment. The user is enabled to view distance space (the result of distance measurement for feature vectors extracted with multiple descriptors) and to define example clusters. In the retrieval process the example clusters are filled with similar media objects. Figure 1 shows a screenshot. The user interface shows the distance-relationships of media objects for two descriptors (marked as D and E). Thus, at any time the user sees the media objects (on the image plane) and works (on the X-Z-plane) in a two-dimensional subspace of distance space. The descriptors for the X- and Z-axis can be changed interactively. Elements A to C show different media objects: A and B are videos (represented as Micons[5]), C are images. Sections 4 and 6 explain, how the media objects are integrated into the user interface.

2.2 Media access and visualisation in VIR prototypes

To the author's knowledge not a single VIR research prototype exists that could deal with both image and video content. Usually, image retrieval systems use very simple interfaces for media access. Photobook[8], for example, works exclusively with the simple PPM-image format. This format stores pixel data as uncompressed RGB-values and can easily be exploited and visualised. The QBIC system[4], even though it was one of the earliest image retrieval systems, offers the most convenient interface. It recognises most common image formats (JPG, GIF, etc.), converts them to RGB-data and offers the image content in a dedicated data-class. For video retrieval systems the situation is equally bad. Most environments can only deal with one particular format. For example, the MPEG-7-reference implementation supports only MPEG-2 video.

Visualisation of images is easy but visualisation of the temporal dimension of videos in static user interfaces is a non-trivial task. There are three principal solutions: (1) integration of the full video with player controls into the environment (CPU power and network bandwidth consuming), (2) creation and usage of animated icons (CPU power consuming) and (3) creation of still images that represent the video content. The third solution is the most widely applied one (in VIR). The simplest form of the third type is an image matrix of all keyframes in a video clip. Another approach is the Micon, a 3D cube showing the first frame of a video clip as well as the first line and the last column of all consecutive frames (see element A and B in Figure 1 for examples). Another type is the Hierarchical Video Browser, a tree-structured view of a video clip. An overview on video visualisation techniques can be found in [5].

3. MEDIA ACCESS

3.1 Low-level media access

As part of VizIR the visual MPEG-7 descriptors are implemented. At least for these descriptors it must be possible to apply them to both types of visual media. For this purpose it is necessary to make the media access in VizIR independent from the underlying media type. In VizIR the class *MediaContent* is responsible for low-level media access. *MediaContent* is a wrapper for two Java technologies that are actually used: the Java Media Framework (JMF) for video access and Java2D for image access[7]. Luckily, both JMF and Java2D use common data structures that allow for unifying media content representation. The API provided by *MediaContent* follows an idea of the developers of the SMIL multimedia authoring standard. In SMIL, each media (including image and text) is assumed to have a temporal dimension and to be of indefinite length. Adopting this idea, *MediaContent* offers just a simple API that allows the VizIR framework programmer to access the view of a media object (as it is perceived) at any point in time. The following code segment shows the declaration of *MediaContent*.

```
class MediaContent {
   MediaContent(String) throws NotFoundException;

   Pixel[][] getViewAtTime(Time);
   Pixel[][] getFirst();
   Pixel[][] getNext();

   void setKeyFrameInterval(int);
   int getKeyFrameInterval();

   void setColorModel(ColorModel);
   ColorModel getColorModel();

   Dimension getSize();
   long getLengthInFrames();

   double getFramesPerSecond();
}
```

MediaContent is initialised with an URL to the media object. *getViewAtTime()* returns the view on the media content as it would be rendered at the specified point in time. *getFirst()* and *getNext()* offer an *Iterator*-interface to media content. The increment of *getNext()* is specified with *setKeyFrameInterval()*. Additionally, *MediaContent* offers methods to read and change the colour model. This is important because, for example, the MPEG-7-descriptors alone use five different colour models (RGB, YCrCb, HSV, HMMD, CIE). The additional get methods allow to read media size and length. The view on the content is always returned as a two-dimensional *Pixel*-array. *Pixel* is defines as follows.

```
class Pixel {
   // RGB, YCrCb, HSV, HMMD, CIE
   final int R_Y_H_X=0,
             G_Cr_S_M_Y=1,
             B_Cb_V_D_Z=2;

   int getComponent(int);
}
```

Pixel defines names for the colour components of the five implemented colour models and the method *getComponent()* to access the independent colour channels. If media content is monochrome, only the first component can be accessed. Colour models are defined in class *ColorModel*. The public description of the class contains classifiers for the models and a get method to read the selected model.

```
class ColorModel {
   final int RGB=0, YCrCb=1, HSV=3, HMMD=4, CIE=5;

   ColorModel(int);
   int getColorModel();
}
```

Media content can be accessed by instantiating *MediaContent* and selecting the appropriate colour model. Views can be accessed with the three get methods that return *Pixel* arrays. The framework does not contain methods to change the colour model for a "rendered" pixel array because this is not required for the usual workflow in VIR systems. Because JMF and Java2D are used to access media data, content in most common formats can be accessed (e.g. AVI, MOV) and most codecs can be used (e.g. MPEG-1 & MPEG-4 video, MJPEG, Cinepak).

3.2 High-level media access

Based on this low-level API VizIR includes high-level classes and methods to work with media content on a more semantic level. Even though *MediaContent* gives the programmer all methods he needs to implement visual descriptors, this API does not provide the degree of freedom that would be desired for the exploitation of completely new directions in descriptor design. This freedom should be provided by the class *VisualCube*. *VisualCube* defines a digital 3D-pixel-cube of arbitrary size that is available through random access. X- and Y-dimension represent the spatial dimension and the Z-axis represents the temporal dimension. The following code segment shows the public declaration of *VisualCube*.

```
class VisualCube extends MediaContent {
   VisualCube(String);
   SpatioTemporalLocator getSize();

   void setSwapping(Boolean);
   Boolean getSwapping();

   Pixel getPixelAt(int x, int y, long t);
   Pixel[][][] copyToArray();
}
```

VisualCube is initialised with an URL to a media object. The size of the contained media object can be read with *getSize()*. This method returns an object of class *SpatioTemporalLocator*. Basically, this class contains the size of the visual cube in all three dimensions. *getSwapping()* and *setSwapping()* can be used to read respectively set, if *VisualCube* should hold the whole media object in direct access memory (resource-consuming, faster) or read on demand (default, slower).

The central method *getPixelAt()* allows to access any point within the content. With *getPixelAt()* spatio-temporal descriptors can easily be extracted from arbitrary media objects. *VisualCube* is not intended to be a changeable (temporary) data structure. Therefore, the declaration does not include a set method for pixel values. For this purpose

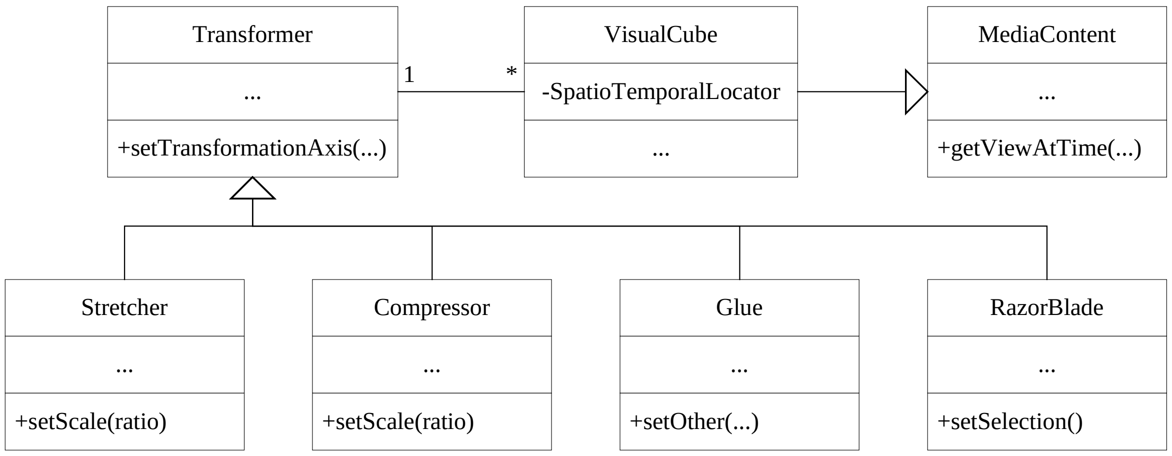

Figure 2: Static structure of transformation classes.

VisualCube can be copied to a three-dimensional *Pixel*-array. The helper class *SpatioTemporalLocator* is declared as follows.

```
class SpatioTemporalLocator {
   Dimension getFrameSize();
   void setFrameSize(Dimension);
   long getLength();       // in frames per second
   void setLength(long);

   double getFrameRate();
   void setFrameRate(double);
}
```

SpatioTemporalLocator contains methods to read and write the size of a media object. Additionally, for the temporal dimension the increment (in frames per second) can be set. *VisualCube* can be used to represent any kind of visual media. If *VisualCube* contains image content, the temporal dimension becomes meaningless. If used on video, it can contain entire video objects but, as well motion trajectories of macro-blocks, rectangular regions, etc. To manipulate *VisualCube*-objects (e.g. extract regions of a video) VizIR provides the class *Transformer*:

```
class Transformer {
   final int X=0, Y=1, T=2;

   Transformer(VisualCube);

   int getTransformationAxis();
   void setTransformationAxis(int);
   VisualCube transform();
}
```

Transformer takes a *VisualCube* as input and performs a pre-defined transformation. After instantiation, each *VisualCube* encapsulates the whole content of a media object. *Transformer* allows the programmer to cut, concatenate, compress and stretch visual cubes. For each transformation, a subclass is defined (see Figure 2): *RazorBlade*, *Glue*, *Compressor*, *Stretcher*. Because of the complexity of transformations each transformation can only be applied to a single dimension. Complex operations have to be serialised to multiple transformations. The selected dimension can be set with *setTransformationAxis()*. *transform()* performs the transformation and returns a new *VisualCube* with the result.

The transformation subclasses (Figure 3 shows examples) need additional parameters. For class *RazorBlade* the selected axis defines the dimension that is cut. Additionally, the point where the media should be cut can be provided with

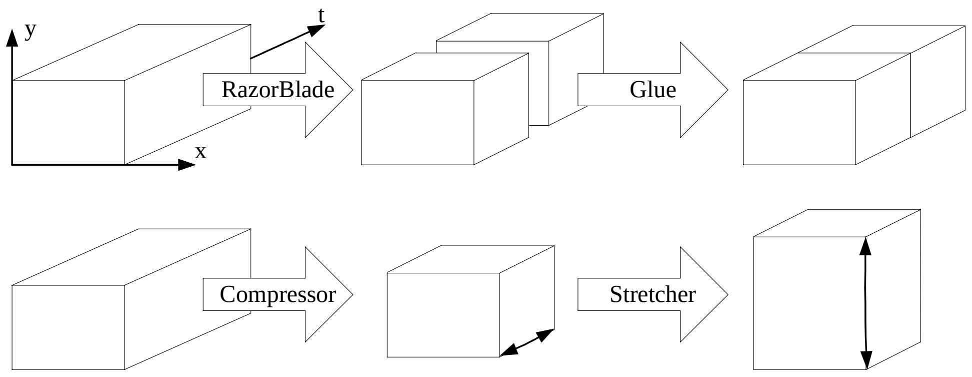

Figure 3: Transformations on *VisualCube*-Objects.

method *setCut(int)*. *setSelection({First|Second})* defines whether the first resulting media object or the second (seen from the origin) should be returned. Class *Glue* requires a second *VisualCube*. This visual cube can be given with *setOther(VisualCube)*. Additionally, *setPosition({LeftTopBegin|RightBottomEnd})* defines, where the second media should be added (depending on the selected axis). For successful gluing both media objects must have equal length on the selected dimension. This can be provided through *Compressor* and *Stretcher*. In method *setScale(double)* both classes take the ratio to which the selected dimension should be compressed or stretched. The reasons why different classes are responsible for compression and stretching are the different algorithms that are used for shrinking and interpolation.

One major limitation exists for transformed *VisualCube*-objects. They can only exist on the heap, swapping is not allowed. Therefore, normally, transformations can only be used on fractions of short clips. Still, this functionality is a valuable extensions for descriptor extraction in visual information retrieval. Especially, because feature extraction is usually done on shot-level. Additionally, *VisualCube*-objects can be used as input for video visualisation techniques.

4. TIME VISUALISATION

The problem of time representation is showing the dynamic content of video objects in a way that is intuitive for the user, easy to render and applicable in the static context of a user interface. VizIR user interfaces make use of 3D-technology (the implementation is based on Gl4Java, a Java-package to access OpenGL[2]). Therefore, in VizIR depth is an additional degree of freedom for time visualisation. The next section explains the basic architecture of media visualisation in VizIR. Section 4.2 describes possible methods for video representation.

4.1 MediaRenderer

In VizIR objects of class *MediaRenderer* and its subclasses are responsible for the generation of (dynamic) representations of media objects (especially video content) in the 3D-user interface. Figure 4 sketches the static structure and relationships of *MediaRenderer*. The visual content taken as input is represented as *MediaContent*. Therefore, *MediaRenderer* is independent of the implemented media access methods. For example, a dynamic renderer that changes its view with the frame rate of the visualised video could be implemented on the basis of a *VideoCube*-object that stores the media data in the direct access memory. *MediaRenderer* has the following public declaration.

```
class MediaRenderer {
   MediaRenderer(MediaContent);
   MediaRenderer(String);
   setViewContextHandle(...);

   void draw();
   void updateModel(InteractionEvent);
   int[] getPosition();
   void setPosition(int, int, int);
```

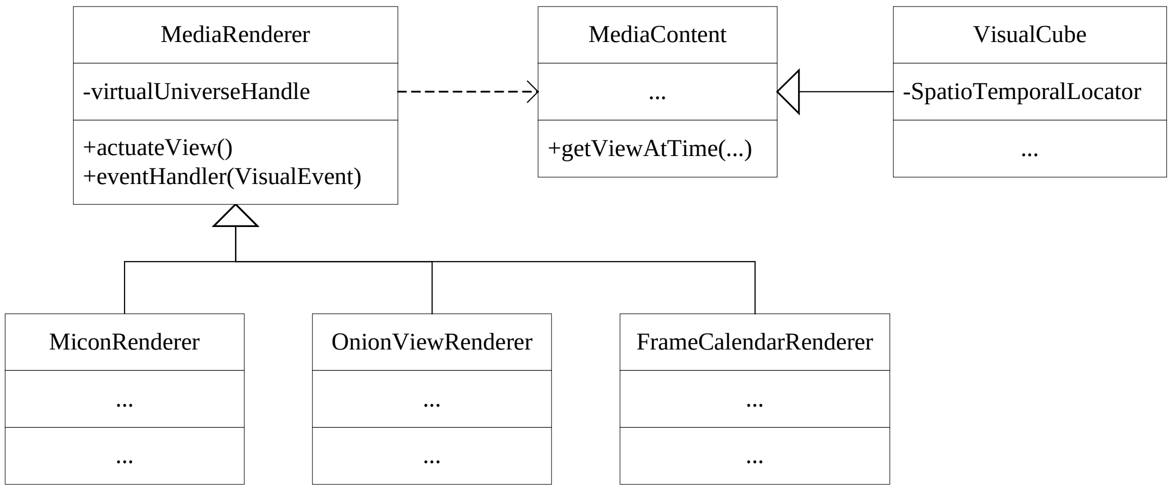

Figure 4: Static structure of media renderer classes.

```
   int[] getSize();
   void setSize(int, int, int);
   String modelToString();
}
```

setViewContextHandle() allows to set a handle to the 3D-user interface. Then, the *MediaRenderer* is able to draw its output directly to the 3D-user interface. Drawing is initiated trough method *draw()*. Method *updateModel()* is implemented by all non-static *MediaRenderer*-classes. This method can be registered in the 3D-user interface as event-handler for interaction events. Then, the VizIR 3D-user interface redirects all interaction events in the visual area of the *MediaRenderer* to *updateModel()*. All events (e.g. *click*, *mouseover*, *mousedown*, *mouseout*) are subclasses of *InteractionEvent*. *serializeViewTo()* is a helper method that converts the *MediaRenderer*-model to an XML-description. For example, this method can be used to store a *MediaRenderer* on the client (with the user interface) and recreate it later with constructor *MediaRenderer(String)*.

The position of a *MediaRenderer* can be read and set with *getPostion()* and *setPostion()*. *setPosition()* takes the X-, Y-, and Z-value of the lower left corner of the visual output. *getSize()* and *setSize()* read respectively set the size of the *MediaRenderer* output. Usually, a *MediaRenderer* object is instantiated for a certain media object and handed over to the 3D-user interface. The 3D-user interface sets its position and size and renders it. If interaction events occur within its area, it calls *updateModel()* and *draw()* to update the view. Thus, arbitrary implementations of *MediaRenderer* with completely different interaction can be used in the VizIR 3D-user interface.

This has one important side-effect. *MediaRenderer* is responsible for all visualisation-related tasks. This includes visual marking of the media object, whether it is part of a group (for example, with a coloured border). Because VizIR querying is based on cluster-definition, the selection of groups is an important task and this functionality must not be forgotten in *updateModel()* of *MediaRenderer*.

4.2 Methods for video representation

At the current point in time, three *MediaRenderer*-classes are implemented: a simple image renderer (for size normalisation, etc.), a renderer for arbitrary XML-data (based on a browser-rendering engine) and a video renderer that generates static Micons. Because of its flexible architecture, *MediaRenderer* can be used to render any kind of content. Using it for images is a very convenient method to create thumbnails, even though this is not the core functionality of *MediaRenderer*.

The Micon-renderer generates 2D-textures of video clips and maps these textures to rectangle-objects. Element A and B in Figure 1 are examples of this kind of Micon. The first frame is shown entirely and the first row and last column are depicted for all consecutive frames. This form of Micon is fast and easy to compute and computationally inexpensive to

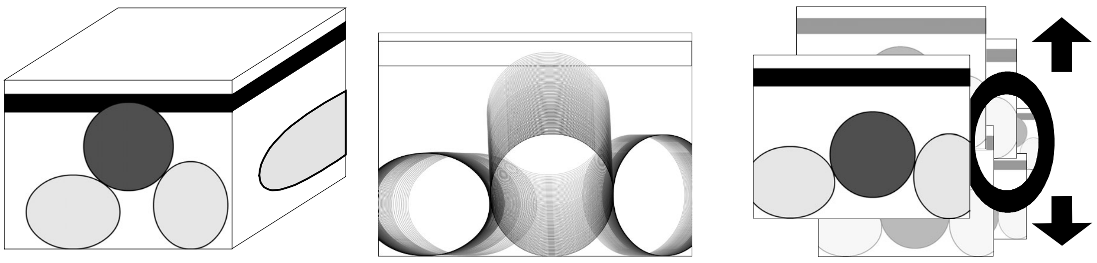
Figure 5: Media visualisation approaches. Left: Micon, middle: onion view, right: key-frame calendar view.

use. Additionally, because it is static it can be created once (e.g. by a server-component) and used multiply (e.g. on multiple clients). On the other hand, it does not make use of the capabilities of the 3D-user interface (the texture looks the same from any point of view) and is not interactive.

Figure 5 shows alternatives that will partially be implemented in VizIR. The first is a cubic Micon. This *MediaRenderer* is based on *VisualCube* and extracts six textures that are laid over a cube. The textures are generated in the same way as for the static Micon. Because the output is a real 3D-object it looks differently from different points of view. In an active version of this *MediaRenderer*, *Transformer*-objects can be instantiated through the context-menu. Therefore, by changing the *VisualCube* that is the basis for the rendering process, the resulting Micon can be examined interactively.

The second example in Figure 5 is an onion-view as it is used in Macromedia Flash to visualise animations. Basically, the resulting image-object consists of the edge maps of all key-frames in the video that are overlaid with a variable alpha-channel. The alpha-channel is minimal for the first frame and increases linearly to the last frame in the video. The major advantage of this renderer is the low complexity of the output. The resulting image can be used in the 3D-user interface in the same way as static Micons.

The third example is like a desktop calendar or organiser for key-frames. It shows the selected frame in a clip with maximum resolution and all other in the background, smaller and with less detail. This *MediaRenderer* gives an impression of the selected frame in the context of its neighbouring key-frames. The active frame can be switched by rotating the calendar by the up and down arrows.

5. METADATA VISUALISATION

Next to media access and time visualisation another – information retrieval-specific – problem is the visualisation of metadata. In VIR systems, metadata occur mainly in three ways: descriptions, querying parameters and media metadata. The better the visualisation of these media-related items is, the more intuitive and easier to use is the resulting system. Subsection 5.1 describes how metadata can be visualised and Subsection 5.2 sketches the methods implemented in VizIR.

5.1 Visualisation of descriptions, querying parameters and metadata

When defining a query one of the most important things the user has to do is specifying the visual properties (descriptors, e.g. colour histograms, texture moments, shapes) he is interested in. All retrieved objects should be similar to the chosen examples according to these properties. If features are spatial (e.g. silhouettes) this is an easy job. But if features are abstract (e.g. statistical) it is important to give the user a feeling for the visual properties of his examples. Therefore, it is necessary to visualise feature data and connect these visualisations to the media objects.

The problem of finding the best visualisation has to be solved for each descriptor independently. Figure 6 shows examples for eight visual MPEG-7-descriptors. For Color Layout a simple image with six regions is suggested that shows the extracted combinations of transformation coefficients in YCrCb-colour space in different colours. Thus, the similarity of two images can easily be judged. Color Structure, Group-of-Frames/Group-of-Pictures Color and Scalable Color can be visualised straight-forward through a colour histogram. Dominant Color could be visualised with five bars in the colours of the extracted dominant colours. The length of the bars represents the calculated percentage-value.

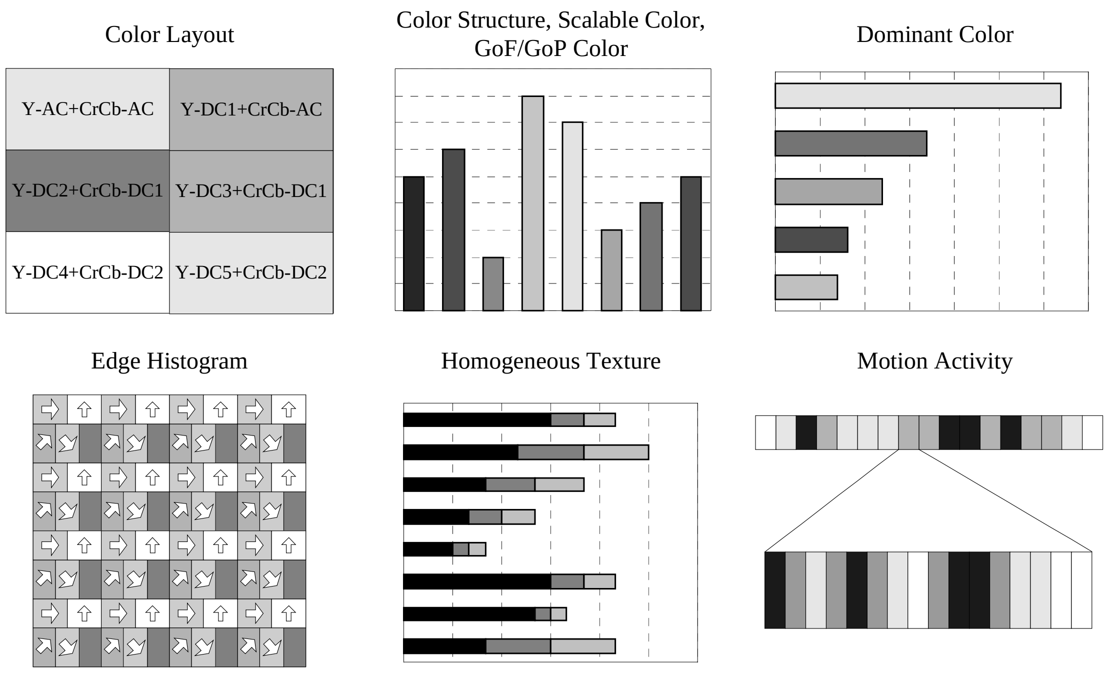

Figure 6: Description visualisation for eight visual MPEG-7-descriptors.

Edge Histogram could be visualised as a 16x16 matrix of small icons. Each icon consists of five bins that are filled with different grey-levels depending on the number of edges that could be extracted for the orientation depicted in the bin. Humans should be able to estimate the similarity of two media objects based on these feature vector visualisations easily. This should help users to understand the impact of the used descriptors on the querying process.

The description elements of Homogeneous Texture could be visualised as bars with three fractions (in different grey-levels). The first fraction of a bar shows the energy value minus its deviation, the first and second show the energy value and all three show the energy value plus the energy deviation. Finally, Motion Activity could be visualised frame-wise by showing the activity extracted in a frame with grey-values. Because not all frames can be depicted at the same time, an interactive hierarchical structure could be used to visualise the whole video clip.

Querying parameters include weights for descriptors and description elements (e.g. weights for histogram entries) and parameters of the used similarity measures (e.g. to control the discriminancy). Usually, parameters and parameter settings are visualised through widgets (see, for example, the panels on top of Figure 1) and organised in Java panels. Metadata in VizIR are always given as XML text. Therefore, state-of-the-art XML browsing tools can be used to visualise elements, attributes and values. It is not planned to develop a tailor-made XML browser for the VizIR project.

5.2 Implementation in VizIR

In VizIR, each descriptor (implementing the interface *Descriptor*) and each query engine (implementing the interface *QueryEngine*) has to implement the interface *Visualize*. This interface forces the implementing class to give a visual representation of the feature data. This representation can either be accessed as a Java *Panel* or as a Java2D *Canvas* and is always static (it does not change, because the object it is associated with cannot change either).

Descriptor visualisations display what the visual descriptor for a certain media object looks like. Therefore, they should be attached to the media objects they belong to. Consequently, in VizIR descriptor panels are handled and displayed by *MediaRenderer*-objects. Whenever the user selects new descriptors for the two dimensions of the 3D-user interface the viewed descriptor visualisations have to be changed by the *MediaRenderer* objects. For this communication process (and in general) the VizIR framework implements the Delegation-Event-Model[2]: Whenever the selection of descriptors for the X- or Z-axis changes, the responsible panel sends an event to all concerned *MediaRenderer*-objects. To be notified, each object must register a listener-method at the event sender. In case of a change this method is called and exchanges the visualised *Descriptor*-object in the view.

Querying parameters and metadata are visualised in stand-alone panels. Normally, querying parameter panels should be static. Visualised metadata have to change whenever a new media object or group is select in the 3D-user interface. The metadata-panel (e.g. an XML-browser) has to register a listener-method at the 3D-user interface to guarantee that. Whenever a new media object is selected an event is fired, caught by the listener and metadata associated to the selected item are displayed.

6. MEDIA TRANSPORT

The last problem of media handling in VIR is the transport of media-related data over the network (e.g. from the query engine or the media database to the user interface). Data that has to be transferred includes media objects (videos and images), descriptions and other metadata and query descriptions. All data types except media data are XML-structured and therefore, in VizIR, XML-messaging is used to transmit these kinds of data. The messaging functionality is encapsulated in the responsible classes *XMLReader* and *XMLWriter* and fully transparent to the VizIR programmer.

The problem of media loading is: The VizIR user interface may run on a different computer far away from the server with the query engine. Each media object is represented by an instance of *MediaRenderer*. As soon as *MediaRenderer* has read the media, it is able to create a view and cache it locally (through *modelToString()*). As long as it has not read the media data, it displays its URL instead. While the dummy text is shown, the media data is transferred in the background (in a separate thread) using the Realtime Transfer Protocol[6] (RTP, JMF implementation).

This functionality is integrated in *MediaRenderer*. If *MediaRenderer* recognises that it has to load a media object over the network (e.g. by the hostname in the URL) it investigates, whether an RTP-client is already running in the application hosting the 3D-user interface and eventually creates one. Then it registers itself as an event-listener at the RTP-receiver and sends an XML-message, containing the URL of the desired media object to the responsible server. As soon as the server has sent the media object through a newly created RTP-stream, the RTP-receiver notifies the *MediaRenderer* with an *MediaObjectArrivedEvent*. Then, the visualisation can be started. Alternatively (if the media representation is static), the visualisation can be created and stored on the server-side and transferred in an XML-message.

To implement the RTP-procedure in the Java Media Framework, custom *DataSource*-objects have to be implemented for every media type (for video and images already existing!) and appropriate RTP-components (*Packetizer* and *Depacketizer*) have to be implemented[7]. The implementation of these components is straight-forward.

7. CONCLUSION

This paper describes how handling of visual media objects is implemented in the visual information retrieval project VizIR. Essentially, four areas are concerned: media access, media representation in user interfaces, visualisation of media-related data and media transport over the network. The paper offers detailed technical descriptions of the solutions developed in VizIR for these areas.

Unified media access for images and video is realised through class *MediaContent*. This class contains methods to access the view on a media at any point in time as well as methods to change the colour model and read/write format parameters (size, length, frame-rate). Based on this low-level-API class *VisualCube* allows accessing spatio-temporal areas in temporal media randomly. *Transformer*-classes allow to modify visual objects in a very simple but effective way. Visualisation of media object is realised in class *MediaRenderer*. Each *MediaRenderer* represents one media object and is responsible for any aspect of its visualisation. In the paper examples for reasonable implementations of *MediaRenderer*-classes were shown. Visualisation of media-related data is strongly connected to *MediaRenderer*. *MediaRenderer* is mainly responsible for displaying visual panels created by other framework components. Finally, media object transport in VizIR is based on the Realtime Transfer Protocol (for media objects) and XML-messaging (for XML-data).

Partially, the methods described in this paper have already been implemented (e.g. *MediaRenderer* for Micons, media transport in the background). The missing components will be implemented as specified in this paper. VizIR is an open project. Interested groups are invited to contact the author and join the design and implementation process.

ACKNOWLEDGEMENTS

The author would like to thank Christian Breiteneder for his valuable comments and suggestions for improvement. The VizIR project is supported by the Austrian Scientific Research Fund (FWF) under grant no. P16111-N05.

REFERENCES

1. A. Del Bimbo, *Visual Information Retrieval*, Morgan Kaufmann Publishers, San Francisco, 1999.

2. H. Eidenberger, C. Breiteneder, "A Framework for User Interface Design in Visual Information Retrieval", *Proceedings IEEE International Symposium on Multimedia Software Engineering*, 255-262, IEEE, Newport Beach, 2002.

3. H. Eidenberger, C. Breiteneder, "A Framework for Visual Information Retrieval", *Proceedings Visual Information Systems Conference*, 105-116, LNCS 2312, Springer Verlag, HSinChu 2002.

4. M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, P. Yanker, "Query by Image and Video Content: The QBIC System", *IEEE Computer*, **28/9**, 23-32, 1995.

5. B. Furht, S.W. Smoliar, H. Zhang, "Video and Image Processing in Multimedia Systems", Kluwer Publishers, Boston 1996.

6. Internet Engineering Task Force website, RFC 1889: Realtime Transfer Protocol, http://www.ietf.org/rfc/rfc1889.txt, last visited 2003-04-02

7. Java Media APIs website, http://java.sun.com/products/java-media/, last visited 2003-04-02.

8. A. Pentland, R.W. Picard, S. Sclaroff, "Photobook: Tools for Content-Based Manipulation of Image Databases", *Proceedings SPIE Storage and Retrieval in Image and Video Databases*, San Jose, 34-47, 1994.

9. A.W.M. Smeulders, M. Worring, S. Santini, A. Gupta, R. Jain, "Content-based image retrieval at the end of the early years", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **22/12**, 1349-1380, 2000.