The Extended Framework Structure for MUVIS

MUVIS Team Proposal for Cost 292 Members,

MUVIS Team:

- Moncef Gabbouj
- Esin Guldogan
- Ahmad Iftikhar
- Mari Partio
- Miguel Ferreira
- Serkan Kiranyaz

CONTENTS

Tampere, 18.10.2005

I. INTRODUCTION

In this document, we propose an extended framework structure designed for MUVIS multimedia indexing and retrieval scheme in order to achieve the dynamic integration and run-time execution for the following:

1. The contemporary *FeX* and *AFeX* framework (Description only)
2. *SBD* (Shot Boundary Detection) framework (proposal)
3. *SEG* (Spatial SEGmentation) framework (proposal)

The main purpose of the extended framework is to use MUVIS as a common platform to develop and test novel techniques on spatial segmentation and shot boundary detection along with the existing (visual and aural) feature extraction. Using such techniques within MUVIS over the multimedia databases yields the necessary basis to perform further (multi-modal) analysis and to achieve a better indexing and retrieval performance.

The following section presents an overview on MUVIS v 1.8. Section III presents an overview on contemporary *FeX* and *AFeX* framework for dynamic visual and Aural Feature eXtraction schemes and file formats. Section IV presents the proposed framework structure for Shot Boundary Detection (*SBD*) for video collections in MUVIS databases and a simple file format for storage. Section V presents the other proposal for spatial SEGmentation (*SEG*) framework and a standard file format for segmentation masks. Finally in Section VI we give some conclusive remarks.

II. MUVIS v1.8 – An Overview

MUVIS [1] aims to achieve a unified and global solution to the media (image, video and audio) capturing, recording, indexing and retrieval combined with browsing and various other visual and semantic capabilities. Figure 1 shows the new system block diagram with underlying applications. Similar to earlier versions MUVIS v 1.8 is also based upon three applications, each of which has different responsibilities and functionalities. The extension mainly occurs on the indexing part, where further support for the two new frameworks, namely Segmentation (*SEG*) and Shot Boundary Detection (*SBD*), is provided. Similar to *FeX* API, the proposed *SEG* and *SBD* APIs provide the necessary means to develop one or more *SEG* and *SBD* modules, which can be dynamically integrated and tested within MUVIS system. The main outcome of *SEG* modules are Segmentation Masks (SMs), which are stored into image files and appended into the MUVIS multimedia database where performed. The *SBD* modules, on the other hand, are used to extract shot boundaries and key-frames from the video clips in a database. During this operation using a specific file format, *DbsEditor* stores them into the database for further analysis such as:

- To use key-frame information during a *FeX* operation
- To establish a better video summarization scheme
- To realize multi-modal analysis over video

More structural information and details about MUVIS can be found in [1].

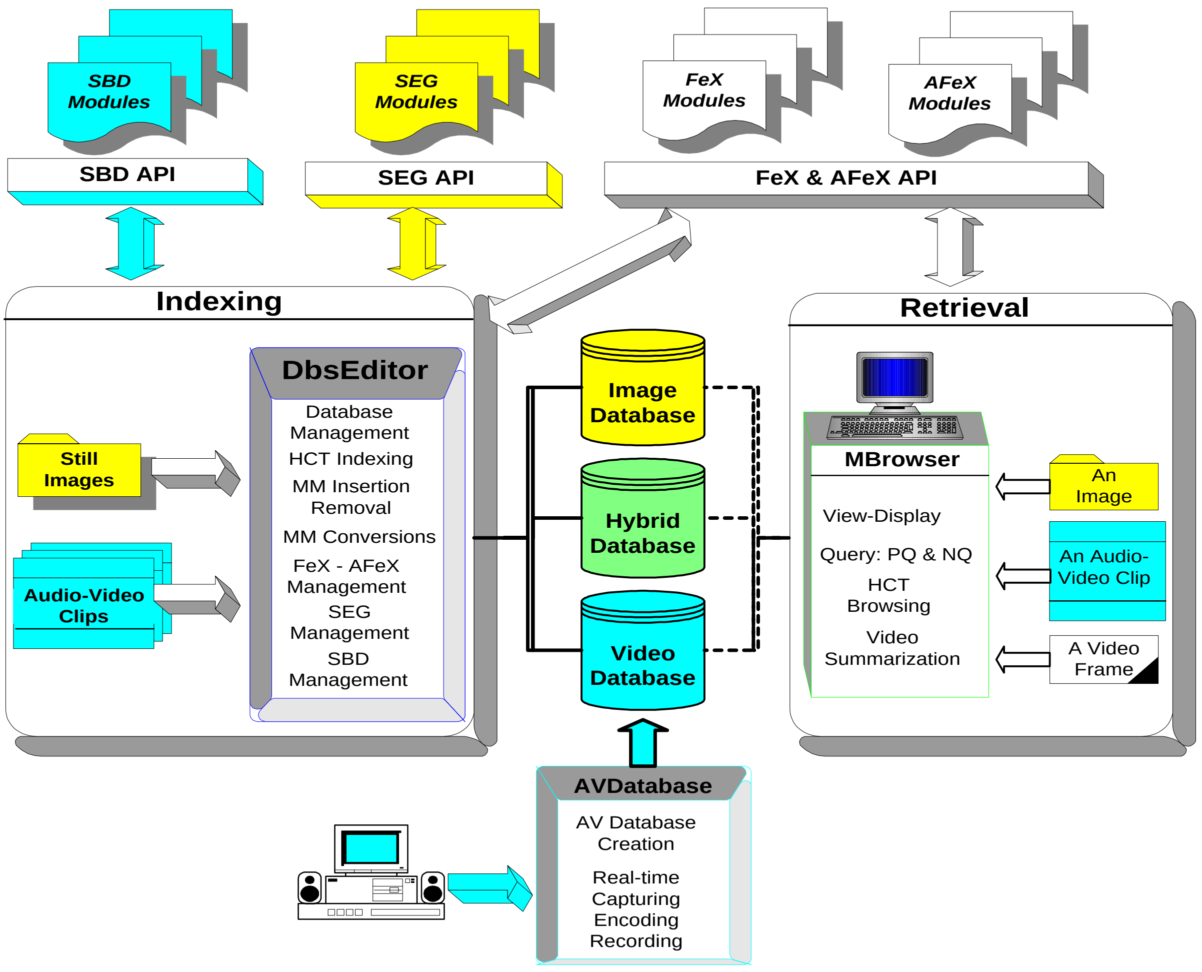

Figure 1: MUVIS Framework Block Diagram.

III. FeX -AFeX Framework in Muvis

Feature vectors of media items in a MUVIS database are represented by normalized array of numbers. Also to support dynamic integration of exclusive *FeX* modules (DLLs), features should be represented in a common and easily supportable format, which is simply the vector representation. These vectors are extracted by dynamic and independent *FeX* modules and they are essentially managed and used by two applications in MUVIS, namely *MBrowser* and *DbsEditor*. *MBrowser* is capable of merging multiple features for querying. The merging scheme used in *MBrowser* requires unit-normalized feature vectors. For this purpose the value of each item in a feature vector is divided by its theoretical maximum value. *DbsEditor* is mainly responsible for the management of a feature extraction operation. Any implemented *FeX* algorithm can be used to extract database features as well as the existing features of any database can be removed. *DbsEditor* also supports multiple sub-feature extraction with different parameters. *MBrowser* is used to query (QBE) a media item within a MUVIS database, if the database contains at least one aural/visual feature and the corresponding *FeX/AFeX* module is available. When necessary (i.e. for an external image query), *MBrowser* uses the same *FeX* module used by *DbsEditor* for extracting the features of the queried media and for obtaining the similarity measure.

A. FeX Framework

There are mainly 2 visual media items in MUVIS framework: video clips and images. Image features are extracted directly from 24bit RGB frame buffer by decoding the image. On the other hand the visual features for video clips are extracted from the key-frames of the clips. In real time video recording case *AVDatabase* may optionally store the uncompressed key-frames of a video clip along with the video bit-stream. If not, *DbsEditor* can extract the key-frames from the video bit-stream and stores them in raw (*YUV 4:2:0)* format. These key-frames are the INTRA frames in MPEG4 or H263 bit-stream. In most circumstances, the encoders use a shot detection algorithm to select the INTRA frames but sometimes a forced-intra scheme might further be applied to prevent a possible degradation. This default key-frame selection scheme is about to change by the integration of the proposed SBD (Shot Boundary Detection) framework into MUVIS v1.8. Upon completion, there will be several techniques to choose better key-frames than the encoder selections and use them during the *FeX* operation.

FeX interface is defined in **Fex_API.h** file. As mentioned before, any *FeX* algorithm should be implemented as a DLL using this API header file. Mainly **Fex_API.h** defines about five different API functions required to manage all feature extraction operations in a dynamic way. It also specifies a certain data structure necessary for feature extraction and communication between the module and application. Figure 2 summarizes the API functions and linkage between MUVIS applications and a sample *FeX* module.

All the *FeX* algorithms should be implemented as a *Dynamic Link Library* (DLL) with respect to a specific *FeX* API, and stored in an appropriate folder (in the same directory with the applications or in “C:\MUVIS\” directory). The naming convention of a *FeX* module should be as follows:

Fex_[feature fourCC code].dll (i.e. “Fex_HSV.dll”)

FeX API provides the necessary handshaking and information flow between a MUVIS application and the *FeX* DLL. The details of the FEX_API and the feature extraction procedure are presented in the following sub-section. Once the features are extracted for any database, the feature vectors for the media primitives of the database are stored in some certain type of files. If the database contains images, then all the image features extracted from a specific *FeX* module are stored in a single file in the same directory with the database file. The naming convention is as follows:

[database former name]_Fex.[FourCC code] (i.e. “MyDatabase_Fex.HSV”).

If the database contains video clips, for each video clip, again a separate feature file for each *FeX* module is created and stored along with the video clip. In this case this feature file contains the feature vectors for all the key-frames in the video. The naming convention in this is as follows:

[indexed video file name]_Fex.[FourCC code] (i.e. “MTV_CLIP_12_Fex.HSV”)

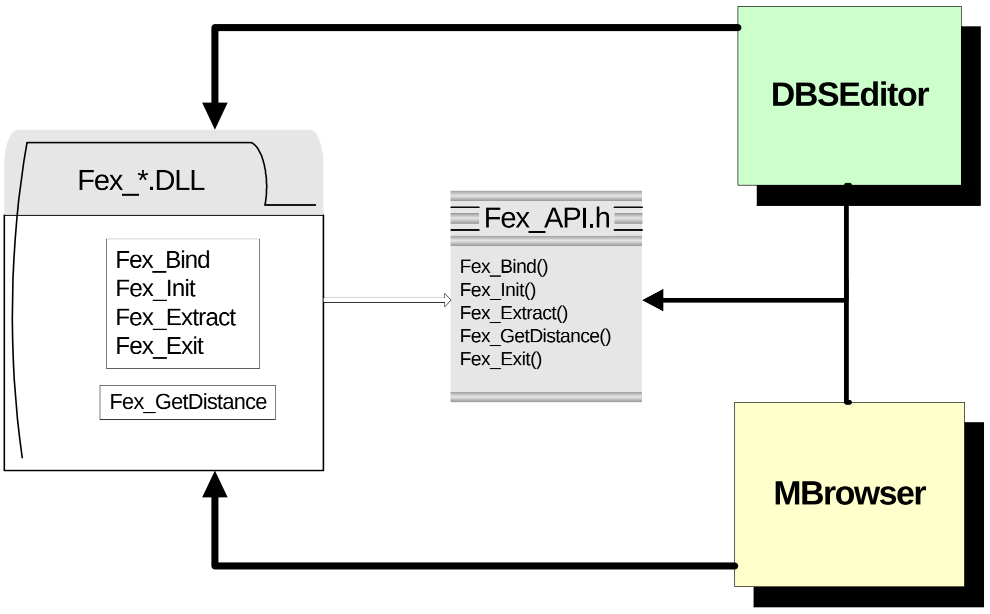

Figure 2: *FeX* Module Integration into MUVIS

1) *Data Structures in FeX Modules*

One enumeration type, two structures and five API function properties (name and types) are declared in **Fex_API.h**. The owner of a *FeX* module should implement all these API functions. There also exists a macro in order to convert a character array to *FourCC* code.

<u>*FrameType* Enumeration*:*</u>
Defines the formats of frames to be used for feature extraction. Currently RGB 24 bit and YUV 4:2:0 frame buffers are supported.

<u>*FexParam* Structure*:*</u>
Created and filled by the *FeX* module for handshaking operation. Any MUVIS application calls **Fex_Bind** function only once with a pointer to this structure. Therefore, a *FeX* module must fill the following members of this structure to introduce itself to the application:

char feat_name[255]	: Description of the feature (i.e. "HSV Histogram"). Used by applications as title.
long feat_fourcc	: Feature fourcc code (i.e. _FourCC('HSV')). Unique identification code for the feature extraction algorithm. Used by applications to identify each FeX module and associated files.
unsigned int feat_param_no	: Number of parameters (i.e. 3 for HSV).
long feat_param_fourcc*	: Array of parameter fourcc codes (i.e. [_FourCC('H'), _FourCC('S'), _FourCC('V')]). Used for display purposes.
double feat_param_default*	: Array of parameter default values (i.e. [8,4,4] for HSV).
FrameType ftype	: Type of frame that should be given to the *FeX* module (i.e. _RGB24)

FrameParam Structure*:*
A MUVIS application calls the **Fex_Extract** function to extract the features of the frame stored inside the given *FrameParam* structure. The format of the frame is the format that is specified by the *FeX* module in the *FexParam* structure.

*unsigned char *buffer* : Raw frame data in a single dimension array. Frame pixels are located in left-to-right top-to-down raster scan order. If the frame is in YUV4:2:0 format, then first part of the buffer keeps the Y values for each pixel, then the part with a quarter size of it keeps the U values each per 4 pixels, and the last part keeps the V values each per 4 pixel. If the frame is in RGB 24bit format, then R, G and B values are all in raster-scan order for each pixel (i.e. RGBRGBRGB…)

int Xs, int Ys : Width and height of the frame.

```
#define _FourCC(x) ((((x)>>24)&0xff) | (((x)>>8)&0xff00) | (((x)<<8)&0xff0000) |
(((x)<<24)&0xff000000))

enum FrameType {
  _YUV420, // I420 or YV12 format ..
  _RGB24, // 3byte frame buffer (For RGBRGBRGB... order)
};

typedef struct {

unsigned char *buffer;
int Xs, int Ys;
} FrameParam;

typedef struct{

char feat_name[255];         // description of the feature (i.e. "HSV Histogram")
long feat_fourcc;            // feature fourcc code (i.e. _FourCC('HSV') )
unsigned int feat_param_no; // No of parameters (i.e. =3 for HSV)
long*feat_param_fourcc;      // each param. fourcc code (i.e. _FourCC('H'))
double* feat_param_default; // each param. default value (i.e. 8 for H);
FrameType ftype;             // Required frame type..
} FexParam;


typedef int      (*FEX_BIND)    (FexParam*);
typedef int      (*FEX_INIT)    (double*);
typedef double*  (*FEX_EXTRACT) (FrameParam, int&);
typedef double   (*FEX_DISTANCE)(double*, double*, int);
typedef int      (*FEX_EXIT)    (FexParam*);

#define  TEXT_FEX_INIT      "Fex_Init"
#define  TEXT_FEX_BIND      "Fex_Bind"
#define  TEXT_FEX_EXTRACT   "Fex_Extract"
#define  TEXT_FEX_EXIT      "Fex_Exit"
#define  TEXT_FEX_DISTANCE  "Fex_GetDistance"
```

Fex_API.h

2) *API Functions in FeX Modules*

int **Fex_Bind***(FexParam*)*
: Used for handshaking operation between a MUVIS application and the *FeX* module (DLL). A pointer to *FexParam* structure is given as a parameter. *FeX* module fills the structure to introduce itself to the application. This function should be called only once at

the beginning, just after the application links the *FeX* module in run-time. This function should return 0 if a problem occurs, and a nonzero value if successfully completed.

int **Fex_Init***(double*)* : Used to initialize the *FeX* module. The feature extraction parameters are given in a double array, which has the size defined in the *FexParam* structure. The *FeX* module performs necessary initialization operations, i.e. memory allocation, table creation etc. This function should be called once at the beginning after the feature extraction parameters are set in the MUVIS application. This function should return 0 if a problem occurs, and a nonzero value if successfully completed.

*double** **Fex_Extract***(FrameParam, int&)*
: Used to extract the features of a frame (buffer) pointed inside the *FrameParam* structure. *FeX* module parameters must be given before when **Fex_Init** is called. This function is called for feature extraction of each frame. It returns the feature vectors, which have the **double** (precision) values. As mentioned before, these vectors should be normalized in such a way that each value in the feature vector and the total length of the vector should be in between 0.0 and 1.0. This normalization is required for merging multiple features while querying in *MBrowser*. The size of the feature vector (number of feature values within the vector) is returned via the given reference parameter.

int **Fex_Exit***(FexParam*)*
: Used for both resetting and terminating the *FeX* module operation. It deallocates the memory spaces for *FexParam* structure allocated in **Fex_Bind** function. Also, if **Fex_Init** has been called already, this function resets the *FeX* module to prepare it for further feature extraction operations. This function should be called at least once while the MUVIS application is closed, but also it can be called at the end of each *FeX* operation to reset (e.g. for temporary clean-up) the *FeX* module.

double **Fex_GetDistance***(double*, double*, int)*
: This is the function to obtain the similarity measure via calculating the distance between two feature vectors. Both feature vectors and their vector size are passed as parameters. Therefore, an appropriate distance metric should be implemented in this function. The computed distance between two vectors is returned as a **double (**precision) value.

Feature values and the distance values are all used as **double** in order to provide generic usage and not to have constraints on the values.

3) Step-by-Step Feature Extraction Processes in MUVIS Applications

The following steps are carried in **DbsEditor** application for feature extraction of any media item (video frame or image):

1- Whenever the *DbsEditor* is started, it looks for the *FeX* modules (DLLs) in proper directories, loads them and links all their functions. It then proceeds by calling their **Fex_Bind** functions to establish handshaking operation. Associated *FexParam* structures are filled by the modules, so that *DbsEditor* has all the necessary information for feature extraction.
2- The adaptive **DbsEditor** user interface lets the user supply the *FeX* parameters and once the user enters the parameters to extract features of a database, it first calls the **Fex_Init** function with the given parameters to initialize the *FeX* module.
3- For each frame in the database (either all the key-frames or images), *DbsEditor* calls the **Fex_Extract** function with the *FrameParam* structure, and the function returns the feature vector along with the vector size.

4- Once all the features with certain parameters are all extracted, **Fex_Exit** function is called to reset the *FeX* module with the *FexParam* structure that was already filled via **Fex_Bind** function.
5- Now in order to extract new features (if required), step 2 to 4 can be repeated if the application is not terminated meanwhile.
6- When *DbsEditor* application is through termination, **Fex_Exit** function is called for final clean-up. This is necessary to deallocate the members inside the *FexParam* structure in case steps 2 to 4 have never been executed during the lifetime of the application.

The following steps are carried in **MBrowser** application for feature extraction and querying of any media item (video frame or image)**:**

1- Whenever the *MBrowser* is started it looks for the *FeX* modules (DLLs) in proper directories, loads them and links all their functions. It then proceeds by calling their **Fex_Bind** functions to establish handshaking operation. Associated *FexParam* structures are filled by the modules, so that *MBrowser* has all the necessary information for feature extraction.
2- Whenever user queries a media, *MBrowser* first calls the **Fex_Init** function with the chosen parameters to initialize the *FeX* module.
3- For the queried frame of the media (either a key-frame of a video clip or image itself), *MBrowser* calls the **Fex_Extract** function with the *FrameParam* structure, and the function returns the feature vector along with the vector size.
4- Once all the features with certain parameters are all extracted, **Fex_Exit** function is called to reset the *FeX* module with the *FexParam* structure that was already filled via **Fex_Bind** function.
5- For all the media primitives in the database, associated feature vectors per frame are read from the appropriate database feature files and the similarity distances between those vectors and the feature vectors of the queried media are obtained using **Fex_GetDistance** function. The distances are sorted and the query results are displayed accordingly.
6- To query a new media, step 2 to 5 can be repeated if the application is not terminated meanwhile.

When *MBrowser* is through termination, **Fex_Exit** function is called for final clean-up. This is necessary to deallocate the members inside the *FexParam* structure in case steps 2 to 4 have never been executed during the lifetime of the application.

4) Image Feature Files

Each image feature file is created for all the images in the database and stored in the same directory along with the database file. A feature file stores all the sub-features' parameters and data (binary) in it. The sub-features are created (extracted) via changing the feature parameters of that particular feature. The following presents a sample (HSV) image feature file:

v 1.8	Version of the feature file
HSV 495 2 3	<FOURCC of feat.> <no. of Images> <no. of sub-feat.> <no. of params>\n
16.000000 8.000000 8.000000 1024	<feat_param1> <fp2> ... <fpN> <feat. vec. size>\n
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
ccccccccccccccccccccccccccccccccccc	
1.000000 8.000000 32.000000 256	
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
ccccccccccccccccccccccccccccccccccc\n	

The binary data contains the feature vectors of each and every image in sequential order. Each feature vector has 6 bytes of header following with the feature vector in double precision. The header contains the image index number (2b), width (2b) and height (2b) of the image.

5) *Visual Feature Files for Video Clips*

Each visual feature file is created per video clip in the database and stored in the same directory with the video file. A feature file stores all the sub-features' parameters and data (binary) in it. The sub-features are created (extracted) via changing the feature parameters of that particular feature. The following presents a sample (HSV) visual feature file:

v 1.8	Version of the feature file
HSV 23 2 3	<FOURCC of feat.> < Key-Frame no> < sub-feat. no> <no. of params>\n
`16.000000 8.000000 8.000000 1024`	<feat_param1> <fp2> ... <fpN> <feat. vec. size>\n
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
cccccccccccccccccccccccccccccccccccc	
`1.000000 8.000000 32.000000 256`	<feat_param1> <fp2> ... <fpN> <feat. vec. size>\n
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
cccccccccccccccccccccccccccccccccccc\n	

The binary data contains the feature vectors of each and every key-frame in sequential order. Each feature vector has 6 bytes of header following with the feature vector in double precision. The header contains the key-frame index number (2b), width (2b) and height (2b) of the key-frame.

B. *AFeX Framework*

AFeX framework mainly supports dynamic audio feature extraction module integration for audio clips. Figure 3 shows the API functions and linkage between MUVIS applications and a sample *AFeX* module. Each audio feature extraction algorithm should be implemented as a Dynamically Linked Library (*DLL)* with respect to *AFeX* API. *AFeX* API provides the necessary handshaking and information flow between a MUVIS application and an *AFeX* module.

AFeX interface is defined in **AFex_API.h** file. As mentioned before, any *AFeX* algorithm should be implemented as a DLL using this API header file. Mainly **AFex_API.h** defines about five different API functions required to manage all feature extraction operations in a dynamic way. It also specifies a certain data structure necessary for feature extraction and communication between the module and application. Figure 1 summarizes the API functions and linkage between MUVIS applications and a sample *FeX* module. There is a naming convention for any *AFeX* module as follows:

AFex_[fourCC code].dll (i.e AFex_MFCC.dll)

All the *AFeX* modules should be stored in the same directory with the application (*DBSEditor* and *MBrowser*) or simply in **C:\MUVIS\** directory. If the database contains audio clips or video clips with audio track, for each clip, again a separate feature file for each *AFeX* module is created and stored along with the video clip. In this case this feature file contains the feature vectors for all the key-frames in audio track. The naming convention in this is as follows:

[indexed video file name]_AFex.[FourCC code] (i.e. "MTV_CLIP_12_AFex.MFCC")

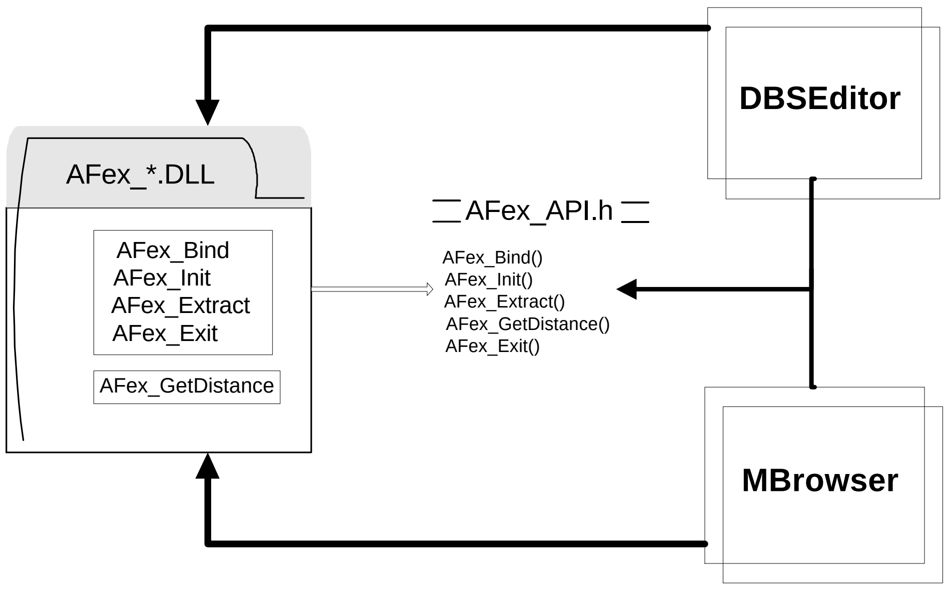

Figure 3: *AFeX* Module interaction with MUVIS applications.

1) *Data Structures in AFeX Modules*

Two enumeration types, two structures and five API function properties (name and types) are declared in **AFex_API.h**. The owner of a *AFeX* module should implement all these API functions. There also exists a macro in order to convert a character array to *FourCC* code.

<u>*c_categ* Enumeration*:*</u>
Defines the audio class types of each audio frame to be used for feature extraction.

<u>*AudioType* Enumeration*:*</u>
This enumeration is not currently used.

AFexParam Structure*:*
Created and filled by an *AFeX* module for handshaking operation. One of the MUVIS applications calls **AFex_Bind** function once with a pointer to this structure in run-time. Therefore, *AFeX* module fills the following members of this structure to introduce itself to the application:

char feat_name[255]	: Description of the feature (i.e. "The feature: MFCC"). Used by applications as title.
long feat_fourcc	: Feature fourcc code (i.e. _FourCC('MFCC')). Unique identification code for the feature extraction algorithm. Used by applications to identify each *AFeX* module and associated files.
unsigned int feat_param_no	: Number of parameters (i.e. 6 for MFCC).
long feat_param_fourcc*	: Array of parameter fourcc codes (i.e. [_FourCC('NoFl'), _FourCC('Hfre'), _FourCC('Ford'), …]). Used for display purposes.
double feat_param_default*	: Array of parameter default values (i.e. [32,22050,24, …] for MFCC).

AFrameParam Structure*:*
A MUVIS application calls the **AFex_Extract** function to extract the features of the frame stored inside the given *FrameParam* structure. The format of the frame is the format that is specified by the *AFeX* module in the *AFexParam* structure.

```
short*    buffer;      // buffer of audio PCM samples..
int       buf_len;     // lenght of buffer..
AudioType a_type;      // frame audio type..
c_categ   f_class;     // frame classification type..
```

```
#define _FourCC(x) ((((x)>>24)&0xff) | (((x)>>8)&0xff00) | (((x)<<8)&0xff0000) |
(((x)<<24)&0xff000000))

enum c_categ {
_uncertain=-1000, // classification is uncertain..
_env_noise=-2,
_music=-1,
_silence,
_speech,
_nonsilent,
_notclassified=1000, // No classification is applied yet..
};

enum AudioType {

_unknown=-1,
_voiced,
_unvoiced
};

typedef struct {

short*   buffer; // buffer of audio PCM samples..
int    buf_len; // lenght of buffer..
AudioType a_type;  // frame audio type..
c_categ   f_class; // frame classification type..
} AFrameParam;

typedef struct{ // AFeX Parameters..

char    feat_name[255]; // Description of the feature (i.e. "Aural feature: MFCC").
long    feat_fourcc; // feature fourcc code (i.e. _FourCC('MFCC') )
unsigned int feat_param_no; // No of parameters (i.e. 6 for MFCC)
long*   feat_param_fourcc; // each param. fourcc code (i.e. _FourCC('NoFl'), ...)
double*    feat_param_default; // param. default value (i.e. [32,22050,24, …] for MFCC)

} AFexParam;

typedef int      (*AFEX_BIND)     (AFexParam*);
typedef int      (*AFEX_INIT)     (double*, int, int, int);
typedef double*  (*AFEX_EXTRACT)  (AFrameParam, int&);
typedef double   (*AFEX_DISTANCE) (double*, double*, int);
typedef int      (*AFEX_EXIT)     (AFexParam*);

#define  TEXT_AFEX_INIT      "AFex_Init"
#define  TEXT_AFEX_BIND      "AFex_Bind"
#define  TEXT_AFEX_EXTRACT   "AFex_Extract"
#define  TEXT_AFEX_EXIT      "AFex_Exit"
#define  TEXT_AFEX_DISTANCE  "AFex_GetDistance"
#define  TEXT_FEX_DISTANCE   "Fex_GetDistance"
```

AFex_API.h

2) *API Functions in FeX Modules*

Five API function properties (name and types) are declared in *AFex_API.h*. The creator of an *AFeX* module should implement all specified API functions, which are described as follows:

- ***AFex_Bind:*** Used for handshaking operation between a MUVIS application and an *AFeX* module. *AFeX* module fills the specific structure to introduce itself to the application. This function is called only once at the beginning, just after the application links the *AFeX* module in run-time.

- ***AFex_Init:*** The feature extraction parameters are given to initialize the *AFeX* module. The *AFeX* module performs necessary initialization operations, i.e. memory allocation, table creation etc. This function is called for the initialization of a unique sub-feature extraction operation. A new sub-feature can be created by using different set of feature parameters.
- ***AFex_Extract:*** It is used to extract the features of an audio frame (buffer). It returns the feature vectors, which should be normalized in such a way that the total length of the vector should be in between 0.0 and 1.0. This normalization is required for merging multiple (sub-) features while querying in *MBrowser*.
- ***AFex_Exit:*** It is for resetting and terminating the *AFeX* module operation. It frees the entire memory space allocated in *AFex_Bind* function. Additionally, if *AFex_Init* has been called already, this function resets the *AFeX* module to perform further feature extraction operations. This function is called at least once while the MUVIS application is terminated, but it might be called at the end of each *AFeX* operation per sub-feature extraction.
- ***AFex_GetDistance:*** This function is used to obtain the similarity measure via calculating the distance between two feature vectors and therefore, the appropriate distance measurement algorithm should be implemented in this function. The resulting distance is returned as a double precision number..

3) *Step-by-Step Aural Feature Extraction Processes in MUVIS Applications*

Same steps are carried out as in *FeX* framework.

4) *Audio Feature Files*

Each audio feature file is created per audio/video clip and stored in the same directory with the clip. A feature file stores all the sub-features' parameters and data (binary) in it. The sub-features are created (extracted) via changing the feature parameters of that particular feature. The following presents a sample (MFCC) audio feature file:

v 1.8	Version of the feature file
32000 1 184520	<sampling freq (hz.)> <no_channels> <total duration (ms.)>\n
MFCC 2 6	<Feat. FourCC> <no of sub-features=S> <no of parameters=N>\n
2 2	<no. of classes in sub_feat 1> <nc2> ... < no. of classes in sub_feat S>\n
20.0 22050.0 16.0 0.0 0.0 1.0 10.0	<feat_param1> <fp2> ... <fpN> <frame duration in msec>\n
-1 12344 16	<1st class=music> <no. of frames> <vector size>
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
ccccccccccccccccccccccccccccccccccc	
1 24545 16	<2nd class=speech> <no. of frames> <vector size>
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
ccccccccccccccccccccccccccccccccccc	
24.0 22050.0 20.0 0.0 0.0 1.0 20.0	<feat_param1> <fp2> ... <fpN> <frame duration in msec>\n
-1 14004 20	<1st class=music> <no. of frames> <vector size>
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
ccccccccccccccccccccccccccccccccccc\n	
1 31004 20	<2nd class=speech> <no. of frames> <vector size>
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
ccccccccccccccccccccccccccccccccccc\n	

The binary data contains the feature vectors of each and every audio key-frame in sequential order.

IV. SBD Framework in MUVIS

Similar to *FeX* framework, Shot Boundary Detection (*SBD*) framework is designed to dynamically integrate any *SBD* algorithm into MUVIS using a dedicated *SBD* API, which is described in this section. As shown in Figure 4, any implemented *SBD* module can therefore be used to extract the following entities from a video clip in a MUVIS database:

- The list of shot boundaries: The start and end frame numbers (indexes) of each shot in a video stream.
- The list of key-frames: One or more key-frames can be chosen in a shot.

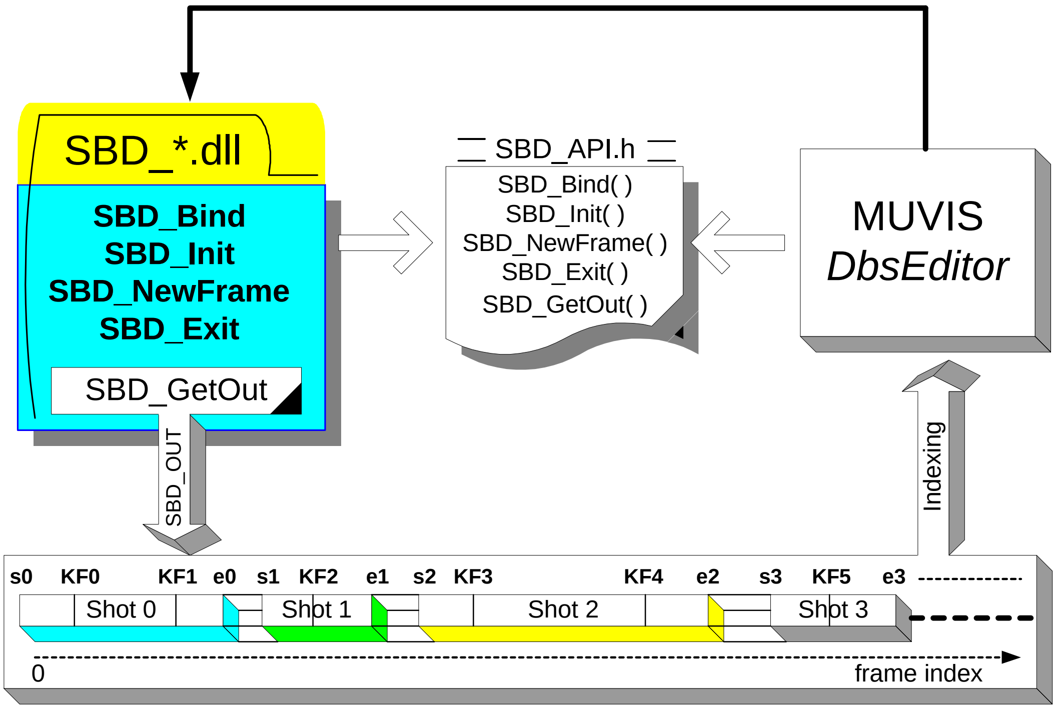

Figure 4: SBD module interaction with MUVIS-DbsEditor

SBD modules are mainly developed for and used by *DbsEditor* application during the indexing of the video clips in a MUVIS database and the shot boundaries with key-frames extracted are stored along with the associated video clip in the same directory. All the *SBD* algorithms should be implemented as a *Dynamic Link Library* (DLL) with respect to a specific *SBD* API, and stored in an appropriate folder (in the same directory with the applications or in "C:\MUVIS\" directory). Also the name of the *SBD* DLL should be in the format as follows:

SBD_[fourCC code].dll (i.e. "SBD_YUVD.dll")

SBD API provides the necessary handshaking and information flow between a MUVIS application and a *SBD* module. If the database contains video clips, for each video clip, again a separate *SBD* file per *SBD* module is created and stored along with the video clip. The naming convention for *SBD* files is as follows:

[indexed video file name]_SBD.[SBD module FourCC] (i.e. "MTV_Clip_12_SBD.YUVD")

1) Data Structures in SBD Modules

Four structures and five API function properties (name and types) are declared in **SBD_API.h**. The owner of a *SBD* module should implement all these API functions. There also exists a macro in order to convert a character array to *FourCC* code.

SBDParam Structure*:*

Created and filled by the *SBD* module for handshaking operation. A MUVIS application (i.e. *DbsEditor*) calls **SBD_Bind** function once with a pointer to this structure in run-time. Therefore, *SBD* module fills the following members of this structure to introduce itself to the application:

char sbd_name[512]	: Description of the *SBD* module (i.e. "Shot Detection by YUV Histogram Difference").
long sbd_fourcc	: *SBD* module fourcc code (i.e. _FourCC('YUVD')). Unique identification code for the SBD algorithm. Used by applications to identify each *SBD* module and associated files.
unsigned int sbd_param_no	: Number of parameters (i.e.4 for YUVD).
long sbd_param_fourcc*	: Array of parameter fourcc codes (i.e. [_FourCC('Ybin'), _FourCC('Ubin'), _FourCC('Vbin'), _FourCC('Thr')]). Used for display purposes.
double sbd_param_default*	: Array of parameter default values (i.e. [8,4,4, 0.5] for YUVD).

FrameParam Structure*:*

Each time a new frame is fed into SBD module via calling *SBD_NewFrame* API function with the frame stored inside the given *FrameParam* structure. The format of the frame must be YUV 4:2:0.

*unsigned char *buffer* : Raw frame data in a single dimension array. Frame pixels are located in left-to-right top-to-down raster scan order. The frame is in YUV4:2:0 format, so first part of the buffer keeps the Y values for each pixel, then a quarter size of it array keeps the U values each per 4 pixels, and the last part keeps the V values each per 4 pixel.

unsigned long fr_index: The index number of the frame. Note that these numbers may not be sequential and they are used to determine shot boundaries and key-frame indexes.

int Xs, int Ys : Width and height of the frame.

ShotBoundary Structure*:*

A simple structure to indicate the start and end frame indexes of a shot.

unsigned long start, end: The start and end frame indexes of a shot

SBD_OUT Structure*:*

The primary output structure of a *SBD* module. The module create and fill this structure and once all the frames are fed into *SBD* module, then it must return this structure filled by the extracted shot boundaries and key-frames to the application via *SBD_GetOut* API function.

ShotBoundary shots:*	The array of shots
unsigned int no_shots:	Number of shots
*unsigned long *KFs:*	The array of Key-Frames
unsigned int no_KFs:	Number of Key-Frames

```
#define _FourCC(x) ((((x)>>24)&0xff) | (((x)>>8)&0xff00) | (((x)<<8)&0xff0000) |
(((x)<<24)&0xff000000))


typedef struct {

unsigned char *buffer;
unsigned long fr_index;
int Xs, Ys;
} FrameParam;

typedef struct{

char sbd_name[255];        // description of the SBD mod (i.e. "YUV Histogram Diff")
long sbd_fourcc;             // SBD fourcc code (i.e. _FourCC('YUVD') )
unsigned int sbd_param_no; // No of parameters (i.e. =4 for YUVD)
long* sbd_param_fourcc;     // each param. fourcc code (i.e. _FourCC('Ybin'))
double* sbd_param_default; // each param. default value (i.e. 8 for Ybin);
} SBDParam;

typedef struct {
unsigned long start, end;   // The start and end frame indexes of a shot..
} ShotBoundary;

typedef struct {

ShotBoundary* shots; // The array of shots..
int no_shots;      // Number of shots..
unsigned long *KFs; // The array of Key-Frames..
int no_KFs;        // Number of Key-Frames..
}SBD_OUT;

typedef int      (*SBD_BIND)    (SBDParam*);
typedef int      (*SBD_INIT)    (double*);
typedef int      (*SBD_NEWFRAME)  (FrameParam, int);
typedef SBD_OUT*   (*SBD_GETOUT)  (void);
typedef int      (*SBD_EXIT)    (SBDParam*);

#define  TEXT_SBD_INIT      "SBD_Init"
#define  TEXT_SBD_BIND      "SBD_Bind"
#define  TEXT_SBD_NEWFRAME  "SBD_NewFrame"
#define  TEXT_SBD_EXIT      "SBD_Exit"
#define  TEXT_SBD_OUTPUT    "SBD_GetOut"
```

SBD_API.h

2) API Functions in SBD Modules

int **SBD_Bind***(SBDParam*)*
: Used for handshaking operation between the MUVIS application (*DbsEditor*) and the *SBD* module (DLL). A pointer to *SBDParam* structure is given as a parameter. The *SBD* module fills the structure to introduce itself to the application. This function should be called only once at the beginning, just after the application links the *SBD* module in run-time. This function should return 0 if a problem occurs, and a nonzero value if successfully completed.

int **SBD_Init***(double*)*: Used to initialize the *SBD* module. The Shot Boundary Detection parameters are given in a double array, which has the size defined in the *SBDParam* structure. The *SBD* module performs necessary initialization operations, i.e. memory allocation, table creation etc. This function should be called once at the beginning after the *SBD* operation parameters are set by the user via GUI of *DbsEditor* application. This function should return 0 if a problem occurs, and a nonzero value if successfully completed.

int **SBD_NewFrame***(FrameParam frame, int eos)*

: Used to feed a new *frame* into the *SBD* module. The new *frame* in *FrameParam* structure is used in the shot boundary detection algorithm and the second parameter *eos* is a nonzero value if this is the last frame in video stream –when the shot boundary detection is due to terminate, otherwise 0.

int **SBD_Exit***(SBDParam*)*

: Used for both resetting and terminating the *SBD* module operation. It deallocates the memory spaces for *SBDParam* structure allocated in **SBD_Bind** function. Also, if **SBD_Init** has been called already, calling this function with no *SBDParam* pointer (set it to NULL) resets the *SBD* module to prepare it for further *SBD* operations. This function should be called at least once while the MUVIS application is closed, but usually it can be called at the end of each *SBD* operation with different *SBD* parameters.

`SBD_OUT*` **SBD_GetOut***()*

: This is the function to get the output of *SBD* module as a pointer of *SBD_OUT* structure. The *SBD_OUT* object should be created and allocated by the *SBD* module owner and once **SBD_Exit** is called, the object can then be de-allocated again by the *SBD* module.

3) *Step-by-Step Shot Boundary Detection Processes in DbsEditor*

The following steps are carried in **DbsEditor** application for shot boundary detection and key-frame extraction using a *SBD* module implemented as a DLL according to *SBD* API.

1. Whenever the *DbsEditor* is started, it looks for the *SBD* modules (DLLs) in proper directories, loads them and links all their functions. It then proceeds by calling their **SBD_Bind** functions to establish handshaking operation. Associated *SBDParam* structures are filled by the modules, so that *DbsEditor* has all the necessary information for proper SBD operation.
2. The adaptive **DbsEditor** user interface lets the user supply the *SBD* parameters and once the user enters the parameters to perform SBD operation over the video clips of the active database, it first calls the **SBD_Init** function with the given parameters to initialize the *SBD* module.
3. For each *frame* in the next video clip of the database, the **SBD_NewFrame** function is called to feed the *frame* into the module.
4. Once all the frames are fed (the last frame is fed with setting *eos*=1 to signal <end of stream> to module), then the extracted shots (with their boundaries) and key-frames can be retrieved by calling **SBD_GetOut** function.
5. Now in order to perform the same *SBD* operation for the next video clip, steps 3 and 4 are simply repeated.
6. In order to perform a new *SBD* operation with different parameters, first **SBD_Exit** function is called to reset the module (with passing NULL pointer) and then steps 2 to 4 are simply repeated.
7. When *DbsEditor* application is about to terminate, **SBD_Exit** function is called for final clean-up. This is necessary to de-allocate the members inside the *SBDParam* structure even in case steps 2 to 4 have never been executed during the lifetime of the application.

4) SBD Files for Video Clips

Each *SBD* file is created per video clip in the database and stored in the same directory with the video file. A *SBD* file stores all the *SBD* parameters and data (binary) in it. The following presents a sample (YUVD) *SBD* file:

v 1.8	Version of the SBD file \n
YUVD 2300 2 4	< SBD FOURCC> < Total frame no> < sub-SBD no> <N= param. no>\n
`8.0 8.0 8.0 0.5 25 25`	<SBDparam_1> <SBDp_2> ... <SBDp_N> <K=KF no> <S=Shot no>\n
kf_1 kf_2 ……………………………kf_K	Binary data
ShotBoundary_1 SB_2 SB_3……………….	
………………………………………SB_S	
`32.0 1.0 1.0 0.5 20 20`	<SBDparam_1> <SBDp_2> ... <SBDp_N> <K=KF no> <S=Shot no>\n
kf_1 kf_2 ……………………………kf_K	Binary data
ShotBoundary_1 SB_2 SB_3……………….	
……………………………………….. SB_S	

The first part of binary data contains key-frame indexes in sequential order. Each key-frame index is stored in 4 bytes (unsigned long). The second part contains the shot boundaries, sequentially dumped in **`ShotBoundary`** structure**.**

V. SEG Framework in MUVIS

Similar to *FeX* framework, a SEGmentation (*SEG*) framework is designed to dynamically integrate any spatial segmentation algorithm into MUVIS using a dedicated *SEG* API, which is described in this section. *SEG* modules are used to create Segmentation Masks (SMs) from an image or key-frame of a video clip. Each SM contains two or more segmented regions (segments) indicated by a distinct 8 bits gray-scale value (between 0-255). Therefore, as a practical limit there can be maximum 256 segments in a SM image. Any SEG module is responsible to assign different (and unique) gray-scale values to the pixels within each segment and the segment assigned with pixel value “0” can be assumed for the background segment.

SEG API provides the necessary handshaking and information flow between a MUVIS application and the *SEG* DLL. ”). *SEG* modules are mainly developed for and used by *DbsEditor* application during the indexing of the video clips and images in a MUVIS database All the *SEG* algorithms should be implemented as a *Dynamic Link Library* (DLL) with respect to the specific *SEG* API, and stored in an appropriate folder (in the same directory with the applications or in “C:\MUVIS\” directory). The naming convention of a *SEG* module must be as follows:

SEG_[fourCC code].dll (i.e. “SEG_RSST.dll”)

1) *Data Structures in SEG Modules*

Two structures, one enumeration and four API function properties (name and types) are declared in **SEG_API.h**. The owner of a *SEG* module should implement all these API functions. There also exists a macro in order to convert a character array to *FourCC* code.

<u>*FrameType* Enumeration*:*</u>
Enumerates two frame formats to be used between the application and a *SEG* module. Currently RGB 24 bit and YUV 4:2:0 frame buffers are supported.

<u>*SegParam* Structure*:*</u>
Created and filled by the *SEG* module for handshaking operation. One of the MUVIS applications (*DbsEditor*) calls **Seg_Bind** function once with a pointer to this structure in run-time. Therefore, *SEG* module fills the following members of this structure to introduce itself to the application:

char seg_name[512]	: Description of the *SEG* module (i.e. "Segmentation by RSST").
long seg_fourcc	: *SEG* module fourcc code (i.e. _FourCC('RSST')). Unique identification code for the SEG algorithm. Used by applications to identify each *SEG* module and associated files.
unsigned int seg_param_no	: Number of parameters (i.e.4 for RSST).
long seg_param_fourcc*	: Array of parameter fourcc codes (i.e. [_FourCC('NoS'), _FourCC('ThrS'), …]). Used for display purposes.
double seg_param_default*	: Array of parameter default values (i.e. [8, 0.5, 4, 0.5] for RSST).
FrameType ftype	: input *frame* that should be given to the *SEG* module (i.e. _RGB24)

<u>*SegParam* Structure*:*</u>
Each time a new frame is fed into SEG module via calling **Seg_Extract** API function with the frame stored inside the given *SegParam* structure. The format of the frame is the format that is specified by the *SEG* module in the *SegParam* structure.

*unsigned char *buffer* : Raw frame data in a single dimension array. Frame pixels are located in left-to-right top-to-down raster scan order. If the frame is in YUV4:2:0 format, then first part of the buffer keeps the Y values for each pixel, then the part with quarter size of it keeps the U values each per 4 pixels, and the last part keeps the V values each per 4 pixel. If the frame is in RGB 24bit format, then R, G and B values are all in raster-scan order for each pixel (i.e. RGBRGBRGB…)

int Xs, int Ys : Width and height of the frame.

```
#define _FourCC(x) ((((x)>>24)&0xff) | (((x)>>8)&0xff00) | (((x)<<8)&0xff0000) |
(((x)<<24)&0xff000000))

enum FrameType {
  _YUV420, // I420 or YV12 format ..
  _RGB24, // 3byte frame buffer (For RGBRGBRGB... order)
};

typedef struct{

unsigned char *buffer;
int Xs, Ys;
} FrameParam;

typedef struct
{
char seg_name[512];          // description of the SEG mod (i.e. "Segmentation RSST")
long seg_fourcc;             // SEG fourcc code (i.e. _FourCC('RSST') )
unsigned int seg_param_no;   // No of parameters (i.e. =4 for RSST)
long*  seg_param_fourcc;     // each param. fourcc code (i.e. _FourCC('NoS'))
double* seg_param_default;   // each param. default value (i.e. 2 for NoS)
FrameType ftype;             // Required frame type..
} SegParam;


typedef int         (*SEG_BIND)    (SegParam*);
typedef int         (*SEG_INIT)    (double*);
typedef int         (*SEG_EXTRACT) (FrameParam, FrameParam);
typedef int         (*SEG_EXIT)    (SegParam*);

#define  TEXT_SEG_INIT      "Seg_Init"
#define  TEXT_SEG_BIND      "Seg_Bind"
#define  TEXT_SEG_EXTRACT   "Seg_Extract"
#define  TEXT_SEG_EXIT      "Seg_Exit"
```

SEG_API.h

2) *API Functions in SEG Modules*

All four of the API functions are listed as below and they should basically return 0 if a problem occurs, and a nonzero value if successfully completed.

int **Seg_Bind**(*SegParam**)

: Used for handshaking operation between the MUVIS application (*DbsEditor*) and the *SEG* module (DLL). A pointer to *SegParam* structure is given as a parameter. The *SEG* module fills the structure to introduce itself to the application. This function should be called only once at the beginning, just after the application links the *SEG* module in run-time.

int **Seg_Init***(double*)* : Used to initialize the *SEG* module. The segmentation parameters are given in a double array, which has the size defined in the *SegParam* structure. The *SEG* module performs necessary initialization operations, i.e. memory allocation, table creation etc. This function should be called once at the beginning after the *SEG* operation parameters are set in the MUVIS application.

int **Seg_Extract***(FrameParam frame, FrameParam SM)*
: Used to extract the SM of a *frame* (buffer) pointed inside the *FrameParam* structure. The segmentation parameters are already given before when **Seg_Init** is called. This function is called to perform segmentation of each and every frame (of an image or video key-frame). Upon completion it fills SM indexes into the buffer of *FrameParam* structure, which is already created and allocated by the application.

int **Seg_Exit***(SegParam*)*
: Used for both resetting and terminating the *SEG* module operation. It deallocates the memory spaces for *SegParam* structure allocated in **Seg_Bind** function. Also, if **Seg_Init** has been called already, calling this function with no *SegParam* pointer (set it to NULL) resets the *SEG* module to prepare it for further *SEG* operations. This function should be called at least once while the MUVIS application is closed, but usually it can be called at the end of each *SEG* operation with different *SEG* parameters.

3) Step-by-Step Segmentation Processes in DbsEditor

The following steps are carried in *DbsEditor* application for spatial segmentation of any media item (a video key-frame or an image):

1. Whenever the *DbsEditor* is started, it looks for the *SEG* modules (DLLs) in proper directories, loads them and links all their functions. It then proceeds by calling their **Seg_Bind** functions to establish handshaking operation. Associated *SegParam* structures are filled by the modules, so that *DbsEditor* has all the necessary information for the segmentation operation.
2. The adaptive user interface of *DbsEditor* lets the user supply the *SEG* parameters and once the user enters the parameters to perform segmentation over the database, it first calls the **Seg_Init** function with the given parameters to initialize the *SEG* module.
3. For each frame in the database (either all the key-frames or images), *DbsEditor* calls the **Seg_Extract** function passing the input *frame* within the *FrameParam* structure, and the fills the buffer of the output *FrameParam* structure with SM indexes.
4. Once the segmentation operation with certain parameters are completed for all the media items in database, **Seg_Exit** function is called to reset the *SEG* module (with passing NULL pointer).
5. Now in order to perform the same segmentation (use the same *SEG* module) with different parameters (if required), steps 3 and 4 can be repeated if the application is not terminated meanwhile.
6. In order to perform a new *SEG* operation with different parameters, first **Seg_Exit** function is called to reset the module (with passing NULL pointer) and then steps 2 to 4 are simply repeated.
7. When *DbsEditor* application is through termination, **Seg_Exit** function is called the *SegParam* structure for final clean-up. This is necessary to de-allocate the members inside the *SegParam* structure in case steps 2 to 4 have never been executed during the lifetime of the application.

4) SEG Files for Images and Video Clips

One or more SM files are created per key-frame of every video clip in the database and stored in “.\SMs\” directory created over the root directory of the video clip. Similarly for images, one or more SM files are created image in the database and stored in “.\SMs\” directory created over the root directory of the database file. The naming convention is as follows:

[indexed video filename]_[KF index]_[SEG module FourCC]_[Sub-Segment Index].pgm - for a Key-Frame

[indexed image filename]_[SEG module FourCC]_[Sub-Segment Index].pgm - for an image

The sub-segment index is used for enumeration of multiple sub-segmentation masks. For instance if 3 SMs are generated using a specific SEG module (say RSST) for an image; “Image_12”, then the filenames will be:

- Image_12_RSST_0.pgm
- Image_12_RSST_1. pgm
- Image_12_RSST_2. pgm

Similarly the following three SM filenames are valid for 8th KF of a video clip called “MTV_Clip_12”:

- MTV_Clip_12_KF_8_RSST_0. pgm
- MTV_Clip_12_KF_8_RSST_1. pgm
- MTV_Clip_12_KF_8_RSST_2. pgm

Note that sub-segments are created by varying the segmentation parameters of a specific *SEG* module (segmentation algorithm). Binary raw PGM ((Portable Gray Map) file format is used for compatibility and efficiency to store SM files. A PGM image represents a grayscale graphic image. A raw PGM file stores all the *SEG* parameters and data (binary) in it. The following presents a sample (RSST) *SEG* file in PGM file format:

P5	The header for binary PGM file.
# v 1.8	# Version of the SBD file \n
# RSST 4	# <SEG FOURCC> <N= param. no>\n
# 8.0 0.5 5 25	# <SEGparam_1> <SEGp_2> ... <SEGp_N> \n
352 288 8	Width Height <segment no - 1> \n
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Binary data
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb	
ccccccccccccccccccccccccccccccccccc	

The binary data contains 1b (8-bits) pixel values of the segments, each of which sequentially indexed from 0 to number of segments in SM. By convention the pixels of the background segment is assumed to be indexed by 0.

VI. CONCLUSION

MUVIS framework is to be reformed to support dynamic integration of visual and aural feature extraction (*FeX* & *AFeX*), Shot Boundary Detection (*SBD*) and Segmentation (*SEG*) modules. Once the modules are implemented according to C-type template defined in the particular interface (API) then they become completely exclusive to MUVIS and the module owners can therefore develop and test them without knowing the implementation details of MUVIS applications.

The new SBD framework allows developers to build and test one or several shot boundary detection algorithms over video collections, using *DbsEditor* application, which stores the resultant shot boundary and key-frame information into the database for further analysis and indexing purposes. Similarly the proposed SEG framework forms a basis to implement spatial segmentation algorithms dynamically within MUVIS for images and key-frames. The resultant SMs are stored as PGM images in the database again for testing (examining and performance evaluation of each algorithm) and indexing purposes. By developing efficient *FeX*, *SBD* and *SEG* modules over the extended framework, performing multi-modal analysis over a MUVIS database will be feasible.

The following *FeX*/*AFeX* modules have been already implemented and integrated into MUVIS framework in order to provide sample implementations of *FeX* /*AFeX* modules to third parties:

- HSV Color Histogram (Fex_HSV.dll).
- YUV Color Histogram (Fex_YUV.dll).
- Dominant Color (Fex_DOMC.dll).
- Canny Edge Direction histogram (Fex_CANN.dll).
- Angular moment histogram over closed-loop segmentation (Fex_MSBF.dll).
- Texture feature via Gray Level Co-occurrence Matrix (Fex_GLCM.dll) [2].
- Texture feature using Ordinal Co-occurrence Matrices (Fex_ORDC.dll) [3].
- Texture feature using Gabor Wavelet Transform (Fex_TEXT.dll) [4].
- MFCC audio features (AFeX_MFCC.dll).

Additionally available MPEG-7 descriptors can be converted as *FeX* modules and thus easily integrated into MUVIS. Upon mutual approval over the proposed (extended) framework and due completion of MUVIS v1.8, several SBD and SEG modules can thus be developed and integrated accordingly.

REFERENCES

[1] MUVIS. http://muvis.cs.tut.fi

[2] M. Partio, B. Cramariuc, M. Gabbouj, A. Visa, “Rock Texture Retrieval Using Gray Level Co-occurrence Matrix”, Proc. of 5th Nordic Signal Processing Symposium, Oct. 2002.

[3] M. Partio, B. Cramariuc, M. Gabbouj, “BLOCK-BASED ORDINAL CO-OCCURRENCE MATRICES FOR TEXTURE SIMILARITY EVALUATION”, Proc. of ICIP 2005, Genoa, Italy, 2005.

[4] W. Y. Ma, B. Manjunath, ”Texture Features for Browsing and Retrieval of Image Data”, IEEE Trans. PAMI, vol. 18, pp 837-842, Aug. 1996.