bewdy, Maaate!

Silvia Pfeiffer and Conrad Parker
CSIRO Mathematical and Information Sciences
Locked Bag 17, North Ryde NSW 1670, Australia
{Silvia.Pfeiffer,Conrad.Parker}@cmis.csiro.au

ABSTRACT

With the vast amount of multimedia data online, content-based access to multimedia files becomes more and more interesting to users. One type of multimedia files widely used nowadays are MPEG-encoded audio files (MPEG-1 layers 1, 2, 3 (MP3)). *MPEG Maaate* is an audio analysis toolkit that supports the extraction of structure and content of such audio files.

bewdy is a graphical interface to easily play around with the analysis modules of *MPEG Maaate* and create content and structural information for MPEG-encoded audio files. It is thus also an application example for the *MPEG Maaate* libraries.

MPEG Maaate has has been published under the GNU GPL and can be found under
http://www.cmis.csiro.au/dmis/Maaate/.

1. INTRODUCTION

Audio content analysis is a very active research area within the multimedia content analysis community. It is especially interesting with respect to new multimedia applications such as new services in the digital TV area, and searching for sounds in the Internet. The tools we present in this paper support research into algorithms that are required for such applications.

The tools operate on MPEG-encoded audio files for several reasons. Firstly, many audio files are compressed in MPEG format. Currently, MP3 (MPEG-1 encoded audio files in Layer 3) is the de-facto standard format for audio files on the Internet. MPEG-encoded videos also contain MPEG-compressed audio. However the MPEG-compressed format is not only popular on the Internet: it is also used within digital video cameras, within radio stations for digital archiving purposes, and is envisaged as the format for future digital TV broadcasts.

Secondly, most audio content information is only accessible in the frequency domain. Analysis of audio content is usually a time consuming task as the data must first be transformed into this domain. However the MPEG compression algorithms do this during encoding and compress the subband information [5, 6, 3, 2]. It is therefore possible to perform audio analysis on MPEG audio files without decoding, and faster algorithms are possible than for the analysis of raw audio data. The only drawback is that analysis algorithms are restricted to the frequency resolution provided by the encoding algorithm.

A further advantage of performing analysis in the MPEG compressed domain is that some inaudible sounds contained in the raw audio have already been removed during encoding by the application of a psychoacoustic model. Thus mainly sounds within human perception remain, providing a cleaner information source for content analysis than was available using the original audio recording.

The tools that are presented in this paper provide libraries to directly access the encoded fields of MPEG audio files and perform content analysis in the compressed domain (*MPEG Maaate*) and a graphical interface (*bewdy*) to parameterise and visualise the results of analyses.

MPEG Maaate's architecture enables integration of different content analysis algorithms for MPEG-encoded audio files. Our own algorithms for loudness approximation [8], background noise level calculation, generic segmentation of 1D-data [8], and generic histograms are currently available as a plugin library. The segmentation module may e.g. be used to detect silent or noisy segments based on the energy, or to segment sound scenes based on the background energy. The histogram module allows to calculate quantized distributions of features such as the energy. *MPEG Maaate* is designed such that inclusion of further analysis algorithms such as have been published in [1, 4, 7] is simple.

The paper is set up as follows: Section 2 presents the architecture of *MPEG Maaate*, Section 3 describes in detail the plugin module interface, and Section 4 presents the GUI. The paper is concluded in Section 5 with an outlook on future work.

2. *MPEG MAAATE OVERVIEW*

MPEG Maaate is implemented in C++ using the standard template library. It consists of two tiers, both of which are a library with a C++ and a C API. It is designed in tiers in order to separate different functionalities and provide simple application programming interfaces (APIs):

- tier 1 implements the parsing of an MPEG audio stream and provides convenience functions to access the fields encoded in an MPEG audio frame. The relevant class that contains the API for tier 1 is the **MPEGfile** class.

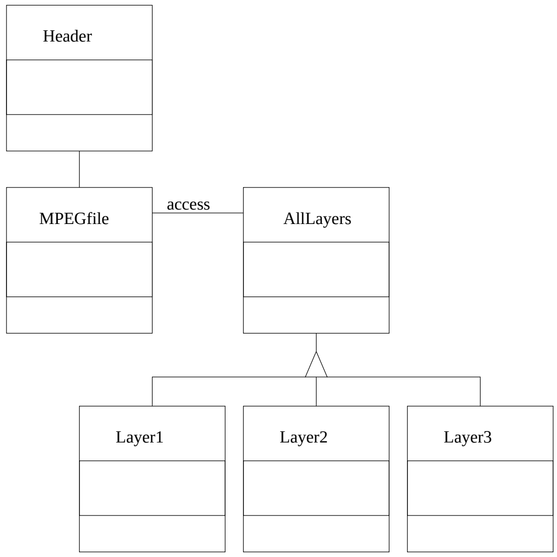

Figure 1: Classes diagram of *MPEG Maaate* parser in OMT

- tier 2 provides two generic data containers as utilities for analysis modules. In addition, it provides a module interface to plugin analysis routines which are stored in dynamically loaded libraries and loaded as requested by an application. The classes that provide the data containers are the **SegmentData** and the **SegmentTable** classes. The plugin module API is provided by the **Plugins** class.

2.1 Design of tier 1

Tier 1, the parsing tier, consists of seven classes altogether:

- MPEGfile: contains the API to open an MPEG audio file and process the audio frames (MPEGfile is derived from Header and thus also contains all header field access functions).
- Header: contains code to parse and access MPEG audio frame headers.
- AllLayers: contains code that all three layers require for parsing one MPEG audio frame. This class is an abstract class as no instances of it may be created but only of its subclasses Layer1-Layer3.
- Layer1-Layer3: are subclasses of AllLayers and contain Layer1/2/3-specific code. They are only supportive classes for the MPEGfile class.
- MDecoder: is a convenience class that provides a simple API to use for playback applications where decoding to PCM into a buffer is required.

All of the API is based on the MPEGfile class. When instantiating an MPEGfile object, it gets tied to an MPEG audio input file. An MPEG audio file consists of a sequence of (audio) frames. Each frame has a header which contains information about the type of data that is encoded in the frame e.g. MPEG version 1 or 2, layer, bitrate, sampling frequency, channels. Based on this information, the length of the data encoded in the frame can be calculated and the data can be parsed. At the API, one frame at a time may be parsed and encoded data requested. The encoded data differs between layers (layers 1 and 2 are similar, layer 3 is very different).

Here are a few examples of access functions to fields encoded in an MPEG audio frame:

```
int    bitallocation    (unsigned int channel,
                          unsigned int subband);
int    scfsi            (unsigned int channel,
                          unsigned int subband);
float  scalefactor      (unsigned int channel,
                          unsigned int subband,
                          unsigned int subsubband=0);
int    sample           (unsigned int channel,
                          unsigned int subband,
                          unsigned int granule,
                          unsigned int subsubband=0);
double restored_sample  (unsigned int channel,
                          unsigned int subband,
                          unsigned int granule,
                          unsigned int subsubband=0);
short  pcm_sample       (unsigned int channel,
                          unsigned int subband,
                          unsigned int granule,
                          unsigned int subsubband=0);
```

The following is sample code for using the API of the *MPEG Maaate* parser:

```
// open MPEG audio file
MPEGfile *infile = new MPEGfile(filename);

// go thru file frame by frame
while (infile->data_available()) {
  infile->parse_frame();
  infile->printheader();

  // get scalefactor of first channel,
  // first subband and first subsubband
  scf = infile->scalefactor(0,0,0);
}
```

Such code may be required within analysis modules.

Although the MPEGfile class provides all the necessary API for tier 1, we have decided to implement another class with an API to enable simple decoding to PCM samples which is required mainly for playback applications: the **MDecoder** class. An example for using the API of the MDecoder is the following:

```
// open MPEG file
MDecoder  *dec = new MDecoder(filename);

// open and setup audio device
```

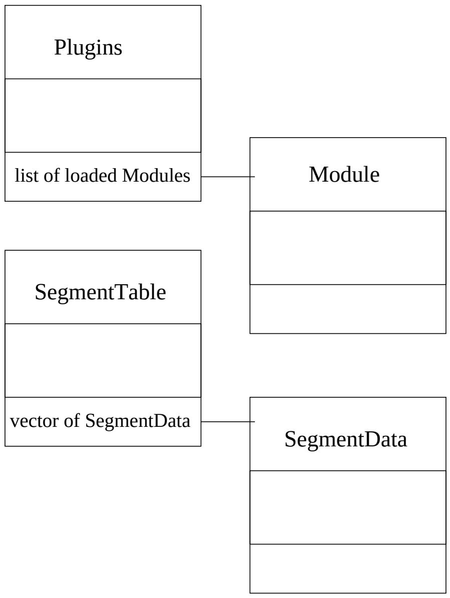

Figure 2: Classes diagram of *MPEG Maaate* analyser in OMT

```
//...

// create output buffer
short buffer[BUFFERSIZE];
long nr_samples = BUFFERSIZE;

// go through file and play decoded frames as long
// as there are samples available
long frame = 0;
while (nr_samples > 0) {
    frame = dec->decode (frame,
                         buffer,
                         &nr_samples,
                         STEREO);

    // play samples
    play (buffer, nr_samples);
}
```

2.2 Design of tier 2

MPEG Maaate tier 2 implements two generic data containers to support the analysis task and a plugin interface to dynamically load analysis modules into an application using *MPEG Maaate*.

2.2.1 Analysis Data Containers

When analysing MPEG audio data, calculated features often come in the form of one- or two-dimensional data that is tied to a specific time range. In addition, the calculation of temporal segmentations is another type of common analysis result. We have implemented two data structures that support the handling of such data:

- SegmentData: contains code to keep content information on a segment of the audio file. A segment is a continuous time range. Example content are a silence period, which contains no data, just time information, or an energy curve, which contains 1D data on a segment of time, namely the energy at specific sampled intervals.
- SegmentTable: contains code to keep content information on a collection of segments of an MPEG audio file, e.g. all silence periods of a file. The segments are ordered by starttime and duration.

An example for the usage of these data containers is the following code:

```
// open an MPEG audio file
MPEGfile    *mf = new MPEGfile(filename);

// create a segment and fill it with subband values
SegmentData *result = new SegmentData(startTime,
                                      endTime,
                                      columns,
                                      rows);
for (int i=0; i<columns; i++) {
   infile->parse_frame();
   for (int j=0; j<rows; j++) {
      result->data[i][j] =
         mf->restored_sample (0, j, 0);
   }
}

// copy the segment to a table of segments
SegmentTable *segments = new SegmentTable();
segments->append (*result);
```

2.2.2 Plugin Module Interface

Modules are routines that provide some analysis function to an application of *MPEG Maaate*. They get compiled separately from the *MPEG Maaate* libraries and linked into their own shared library. They are dynamically loaded into an application, which goes back to *MPEG Maaate* tier 2 and tells it to load those libraries that contain the modules it requires. After loading they are available to the application.

Some advantages of the plugin interface are:

- Modules can be developed and compiled separately from *MPEG Maaate*.
- The *MPEG Maaate* libraries can be extended by analysis modules without ever having to recompile *MPEG Maaate*.
- The boundary between *MPEG Maaate* code and analysis module code is explicit. An author of a plugin module does not have to add any code into *MPEG Maaate* for his module to be used with it.
- The separation of *MPEG Maaate* and the modules simplifies determination of legal ownership of code.

3. *MPEG MAAATE PLUGIN MODULES*

3.1 Module specification

3.1.1 What is a module?

A module is a collection of functions that provide some augmented functionality on top of *MPEG Maaate.* Modules that analyse the content of MPEG audio files usually collect information on several MPEG audio frames and calculate more abstract information from these. Tier 2 was constructed for such types of modules. However, other modules are possible, too.

Examples of modules are

- feature extraction modules such as energy, spectral centroid or spectral bandwidth modules. Such modules usually have to make use of the tier 1 field access functions and store their results in one of the data containers supplied by tier 2. An example of such a module is the sumscf module which can be inspected in the Maaate source archive under src/plugins/sumscf.cc.
- feature analysis modules that use the extracted features for some further (usually statistical) analysis such as clustering, segmentation or histogram modules. These modules usually make use of a filled container and store their results in another convenience container. An example of such a module is the segmentation module which can be inspected in the Maaate source archive under src/plugins/segmentation.cc.
- content analysis modules that calculate higher level information using feature extraction and analysis modules such as silence / music / speech determination. Suchd modules usually call other modules to calculate their results, which again may be stored in convenience containers. An example of such a module is the silence segmentation module which can be inspected in the Maaate source archive under src/plugins/silences.cc.

A module is an instance of the Module class, which also provides convenience functions to get information on the instantiated module, handle input and output parameters, check constraints on parameters and call the module functions.

3.1.2 Module functions

The apply-function of a module contains the implementation of the analysis functionality provided to Maaate. It takes as input a list of parameters and produces as a result of its processing a list of output parameters. There are other functions required to set up the environment under which the apply-function will work. Here is a description of all functions possibly contained within a module and callable at the module interface:

- an **init-function** (required), which sets up the basic information of the module such as its name, author, or description, and the input and output parameter specification.
- a **default-function** (required), which sets default values for input parameters and returns the input parameter list.
- a **suggest-function** (optional, recommended), which takes an input parameter list, suggests parameter values based on information provided by other parameters, and changes constraints of input parameters as required.
- a **reset-function** (optional), which provides the possibility to reset a module, e.g. internal processing values or parameter values.
- an **apply-function** (required), which takes an input parameter list, performs its analysis function and returns the calculated output parameters.
- a **destroy-function** (optional), which cleans up memory allocated within the module and deletes parameter specifications.

In addition, a function to construct the module and add it to the list of available modules must be provided.

3.1.3 Module parameters

The apply-function requires input parameters to work on and produces output parameters that contain its results. A parameter is an instance of the **ModuleParam** class. Module parameters are handled by an application as follows: The init-function sets up the list of parameter specifications for input and output parameters. Thereafter, the application sets up a concrete input parameter list with default values by calling the default-function. The application may then change the input parameter values as it requires. Then it has the possibility to call the suggest-function which will take care of setting further parameter values and parameter constraints based upon internal module knowledge and the provided parameter values. It will also check provided parameter values for sanity and change them appropriately.

Now, the application may call the apply-function. The first step within the apply-function is to check parameter values again for being within constraints (such as acceptable numeric ranges or predefined values). So, if an application has decided not to use the default- and suggest-functions, this still ensures that provided parameter values are sane. Constraints are hard limits, i.e. the apply-function will only be called if the parameters satisfy their constraints. The apply-function will return a list of output parameter values, which contain the results of the module execution.

3.1.4 Module parameter data types

There is a small set of allowed data types for parameters. They are either basic types or complex types and are all enumerated in the type MaaateType.

3.1.4.1 Basic types

The *MPEG Maaate* module interface provides the following basic types for parameters to be passed to a module or resulting from a module:

- a boolean type: `MAAATE_TYPE_BOOL`,
- an integer type: `MAAATE_TYPE_INT`, (this type may also be used to pass file descriptors to or from the module)

- a real type: `MAAATE_TYPE_REAL`, and
- a string type: `MAAATE_TYPE_STRING`.

3.1.4.2 Complex types

The Maaate module interface provides the following complex types for parameters to be passed to a module or resulting from a module:

- a pointer to an opened MPEG audio file (MPEGfile *) constructed using tier 1: `MAAATE_TYPE_MPEGFILE`,
- a pointer to a segment data structure (SegmentData *), which is provided by tier 2 and contains for a certain specified time period a matrix of values: `MAAATE_TYPE_SEGMENTDATA`, and
- a pointer to a segment table (SegmentTable *), which is also provided by tier 2 and contains a collection of SegmentData containers thus covering several time periods: `MAAATE_TYPE_SEGMENTTABLE`.

3.1.5 Parameter constraints

Modules are provided with input parameter values via the input parameter list. The apply-function of a module usually requires parameter values to be within a specific range of allowed values. This is what we call parameter constraints. There are three types of constraints:

- no constraints (`MAAATE_CONSTRAINT_NONE`): parameter values are allowed to take on any value of the parameter's data type,
- a (list of) single value(s) (`MAAATE_CONSTRAINT_VALUE`): parameter values are allowed to take on any value of a list of provided values of the parameter's data type,
- a (list of) value range(s) (`MAAATE_CONSTRAINT_RANGE`): parameter values are allowed to take on any value contained within any of the ranges in a range list.

A single constraint is an instance of the class **ModuleParamConstraint** being either a single value or a single range. For one parameter, there is usually a list of constraints. Such a constraint list is realized by instantiating the **MaaateConstraint** class. This class also provides convenience functions to handle constraints such as adding constraints or checking if values satisfy constraints.

3.1.6 Parameter specifications

A module is a generic interface to an analysis function. Every analysis function however requires different input parameters and produces different output parameters. The generic interface therefore specifies that a list of (generic) parameters be accepted by a module and a list of parameters be output from a module. The exact specification of the parameters can only be produced by the module itself. This is performed in the module's init-function.

A parameter specification is an instance of the **ModuleParamSpec** class, which contains the specification of a single parameter. The specification consists of the parameter's name, a description, its data type (see Section 3.1.4 for possible data types), default value and constraints (see Section 3.1.5 for constraint descriptions). The default value that is provided to a parameter must be of the data type of its parameter specification.

3.1.7 Existing modules

The current version of *MPEG Maaate* contains a plugin library with the following analysis algorithms:

- **sumscf** - loudness approximation: calculates a loudness approximation based on the scalefactor values of the frames. For more details refer to [8].
- **segmentation**: calculates segments on a 1D SegmentData structure primarily based on a threshold and a minimum segment duration. For more details also refer to [8].
- **silence**: calculates segments of (relative) silence using the sumscf and segmentation modules. For more details also refer to [8].
- background noise level: using the loudness approximation, calculates how high the silence threshold has to be set before any silence is detected.
- noise segmentation: calculates segments of high loudness also using a threshold and a minimum noise duration. This is analogous to the silence segmentation.
- histogram: calculates histograms on a 1D SegmentData structure giving an overview of the distribution of the 1D data. This may be used to support the selection of thresholds for segmentation.

3.1.8 Writing a module

The following header file has to be included in files containing module code: <MaaateA.h>. It includes all required *MPEG Maaate* header files including tier 1 and tier 2 header files.

When you want to write a module, you have to provide the functions that specify a module: init-, default-, suggest-, reset-, apply-, and destroy-function. The names you give to these functions are arbitrary, however we propose to call them `init_<modulename>`, `default_<modulename>` etc. as we have done with our modules. They should be static functions in order not to appear in the library's symbol table.

Here are the generic layouts of these functions:

```
// init-function: only takes a Module pointer
typedef
void (*ModuleInitFunc) (Module *);

// default-function: takes a Module pointer and
// returns the input parameter list
typedef
list<ModuleParam> (*ModuleDefaultFunc) (Module * m);

// suggest-function: takes a Module pointer and a
```

```
// pointer to the input parameter list which it
// might change
typedef
void (*ModuleSuggestValues)
        (Module * m,
         list<ModuleParam> * paramsIn);

// apply-function: takes a Module pointer and the
// input parameter list and returns the output
// parameter list
typedef
list<ModuleParam> (*ModuleApplyFunc)
                       (Module * m,
                        list<ModuleParam> * paramsIn);

// destroy-function: only takes a Module pointer
typedef
void (*ModuleDestroyFunc) (Module *);

// reset-function: only takes a Module pointer
typedef
void (*ModuleResetFunc) (Module *);
```

In the init-function, you need to set up the module's specification consisting of its description (modName, modDesc, modAuthor, modCopyright, modUrl) and the input and output parameter specifications (modParamInSpecs and modParamOutSpecs), possibly with constraints.

The modName must be a short string without blanks which is used to identify the module. As we are planning a scripting interface to Maaate that uses this name as command, it must be without blanks. The modDesc should be a somewhat longer ASCII text describing what this module does. The modAuthor field should be a comma separated list of names of authors of this module, each optionally providing an email address in angle brackets. The modCopyright field should be a string such as "(c) 2000 Sue Clancy". The modUrl field can optionally give the URL of a web page that contains more details on the module.

The module input and output parameters also have to be specified. When adding another parameter specification, it is required to give an identifying name for the parameter, a description, the parameter's type, its default value and possibly constraints.

You should also specify a default-function, which creates a parameter list from the parameter specification given in the init-function and sets default values for the parameters. Input and output parameters of modules are stored in a list (list<ModuleParam>). The parameter values have to adhere to the parameter specification defined during the init-function of the module (list<ModuleParamSpec>) and come in the correct order. If this function relies only upon default values given in the init-function, it is not required to implement the default-function as its standard implementation uses the specification given in the init-function to set up the input parameter list.

You don't need to specify a suggest-, reset- or destroy-function, except if you would like to provide their functionality to an application. We recommend to at least specify the suggest-function, which changes constraints and parameter values according to already specified values. It also assures that constraints are met.

The apply-function is the function the provides the functionality of the module to an application. If you would like to share information between the functions of a Module or between different calls of a function, it is best to declare static global variables within the source code file of the module and communicate values through these variables. Thus, it is best to specify all the module functions within one file. Remember to specify the functions and global variables as static to prevent them from appearing in the library's symbol table.

3.2 Plugin libraries

3.2.1 What is a plugin library?

A plugin library is a separately compiled, shared library. It may contain one or many modules. Several such libraries may be dynamically loaded using *MPEG Maaate* tier 2. Single modules or complete libraries may also be unloaded from *Maaate Maaate* as required by the application.

The modules are all administrated within one instance of the **Plugins** class. This class therefore provides functionality to load and unload single modules and whole plugin libraries, and administrates the list of available modules. It also provides functions to access modules by their name.

3.2.2 Building plugin libraries

When building a shared library to contain one or more modules, a function called **loadModules** has to be supplied, which instantiates the modules of the library and returns their list to *MPEG Maaate*. Similarly an **unloadModules** function has to be supplied, which deletes the instantiated modules. These are the only names that should be exposed in the library's symbol table; declare all other functions and global variables as static.

In order to load several modules contained within one plugin library, it is best to specify within each module's file another function that instantiates a Module with the given functions and returns it. Then, there has to be a separate file that contains the loadModules function, which calls all theses functions, pushing the instantiated Modules into a Module list which gets returned by the loadModules function upon opening the shared library with tier 2.

3.2.3 Using plugins in applications

If an application wants to dynamically load modules, it is first required to create an instance of the Plugins class and then load the shared module libraries with any of the loading functions defined within the Plugins class, for example:

```
// open a plugin library
Plugins * plugins = new Plugins();
plugins->AddLibrary(string("libMaaateM.so"));
```

The **AddLibrary** call also creates an instance of the PluginLibrary class which loads all the contained modules of the given plugin library and sets up input and output parameter specifications for them.

The application may then call the following functions in sequence to use a module's functionality:

```
// returns a pointer to the requested module
Module * m = plugins->GetModule("modulename");

// creates the input parameter list for a
// module from its specification
list<ModuleParam> paramsIn = m->defaultValues();

// application makes changes to parameter values
// ...

// possibly call suggestValues:
// suggests further parameter values and changes
// constraints
m->suggestValues(&paramsIn);

// application makes possibly further changes
// to parameter values
// ...

// perform operations using input parameter values
// and store results in output parameter list
list<ModuleParam> paramsOut = m->apply(&paramsIn);

// possibly call reset()
// resets internal module values
m->reset();

// possibly call apply() again
// ...

// deletes the module after calling its destroy
// function and unloads all the modules of the
// library from Maaate
m->~PluginLibrary();
```

4. *BEWDY*

In order to demonstrate the capabilities of *MPEG Maaate*, we have implemented a graphical application that enables users to visualize the results of a few modules. The application is called *bewdy*. It runs in the GNOME application environment and requires an OpenSound Sysetm compatible sound API. As mentioned in Section 3.1.7 the current distribution of *MPEG Maaate* contains a plugin library with modules to calculate a loudness approximation, silence segmentation, noise segmentation, background noise level and loudness histogram. These modules are visualised in *bewdy*. It displays the distribution of silence and noise segments and enables navigation through the file based on these. A user can also interactively change the parameter settings for calculating silence or noise segments and thus experiment with different types of results.

Figure 3 shows the display of *bewdy* for a radio news broadcast. The display is organised into timelines with time increasing from left to right. Time resolution may be chosen as required using the zooming buttons. The top timeline displays the loudness approximation as a curve. To the left of it, the loudness histogram is displayed and helps in adjusting parameters for silence and noise segmentation. At the bottom, the calculated background noise level is displayed which also helps in setting parameters.

The timelines below the loudness are synchronised with the loudness with respect to time. Each row represents the results of one execution of either a silence or a noise segmentation and thus visualizes a SegmentTable. The displayed top two SegmentTables are results of silence segmentations with different parameter settings. The brightness of a segment indicates its confidence, i.e. the brighter it is the more reliable is the calculated silence. As the loaded file is a radio broadcast, we have chosen parameter settings such that the first one gives news stories and the second one phrases. On a live demonstration, this can be validated interactively by selecting a segment or a subpart of the loudness curve and activating the play-segment-button. The third segmentation displays the results of a noise calculation.

5. CONCLUSIONS AND OUTLOOK

We have presented our toolkit for audio content analysis on MPEG-compressed audio: *MPEG Maaate*. The toolkit has been published as open source software under the GNU General Public License (GPL) and is available from http://www.cmis.csiro.au/dmis/Maaate/. We are currently working on further analysis algorithms such as speaker change detection or segmentation of music, speech, and sound effects. Such algorithms are useful to automatically segment and index MPEG-encoded audio files to gain direct access to specific content. An example application may be the automatic extraction of all music parts contained in a movie.

We have also presented *bewdy*, a GUI to the loudness-based segmentation algorithms contained within *MPEG Maaate*. It provides an intuitive human computer interface to demonstrate and investigate the effects of different parameter settings on loudness-based segmentation results. We are planning to adapt *bewdy* to the plugin module interface such that it dynamically creates menus and displays based upon the list of loaded modules.

6. REFERENCES

[1] G. Boccignone, M. D. Santo, and G. Percannella. Joint audio-video processing of mpeg encoded sequences. In *Proc. IEEE Intl. Conf. on Multimedia Computing and Systems (ICMCS)*, volume 2, pages 225–229, 1999.

[2] Haskell, Puri, and Netravali. *Digital Video: An Introduction to MPEG-2*, chapter 4. Chapman & Hall, New York, 1997.

[3] ISO, International Organization for Standardization. International standard 11172-3, Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 MBit/s - Part 3: Audio, 1993.

[4] Y. Nakajima, Y. Lu, M. Sugano, A. Yoneyama, H. Yanagihara, and A. Kurematsu. A fast audio classification from mpeg. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*,

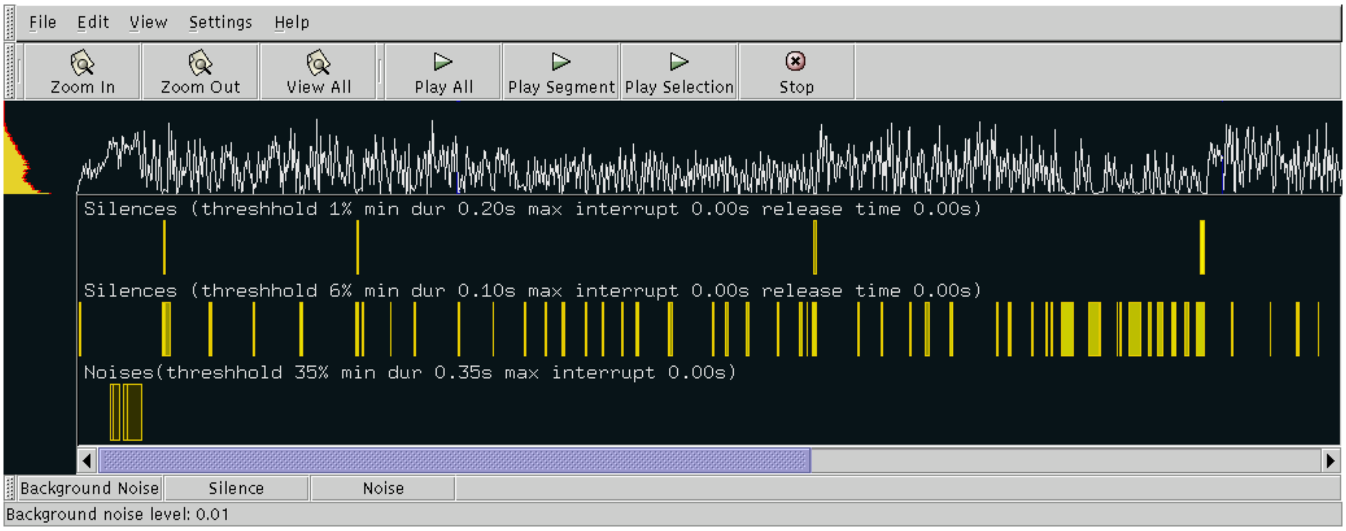

Figure 3: Screen dump of *bewdy*

volume IV, pages 3005–3008, Phoenix, Arizona, USA, May 1999.

[5] P. Noll. MPEG digital audio coding. *IEEE Signal Processing Magazine*, pages 59–81, September 1997.

[6] D. Pan. A tutorial on MPEG/audio compression. *IEEE Multimedia*, 2(2):60–74, Summer 1995.

[7] N. Patel and I. Sethi. Audio characterization for video indexing. In *Proc. SPIE, Storage and Retrieval for Still Image and Video Databases IV*, volume 2670, pages 373–384, San José, CA, USA, February 1996.

[8] S. Pfeiffer, J. Robert-Ribes, and D. Kim. Audio content extraction from MPEG-encoded sequences. In P. Wang, editor, *Proc. Fifth Joint Conference on Information Sciences*, volume II, pages 513–516, Atlantic City, New Jersey, Feb/Mar 1999.