J-DAFX - DIGITAL AUDIO EFFECTS IN JAVA

Mijail Guillemard, Christian Ruwwe and Udo Zölzer

Department for Signal Processing and Communications
Helmut-Schmidt-University
University of the Federal Armed Forces Hamburg, Germany
{mijail|christian.ruwwe|udo.zoelzer}@hsu-hh.de
http://ant.hsu-hh.de

ABSTRACT

This paper describes an attempt to provide an online learning platform for digital audio effects. After a comprehensive study of different technologies presenting multimedia content dynamically reacting to user input, we decided to use Java Applets. Further investigations regard the implementation issues - especially the processing and visualization of audio data - and present a general framework used in our department. Recent and future digital effects implemented in this framework can be found on our web site.

1. INTRODUCTION

The purpose of our work is to provide a learning platform for students and everyone interested in digital audio signal processing [1] and digital audio effects [2]. This guidance should come in a very handy and *easy-to-use* format so that everyone can handle it without being a computer expert. In contrast to the general purpose investigations of [3] and [4], we will concentrate on this e-learning platform only.

Several approaches for usage in this application have been investigated: a stand-alone application, plugins for well-known audio applications and web-enabled multimedia tools including Macromedia's Flash and Java Applets. Chapter 2 will give a short summary of these tools and their pros and cons. We decided to use Java Applets for our purpose. In Chapter 3 you will find some details of the implementation done in Java.

2. MULTIMEDIA TOOLS

2.1. Key Features

There are several possibilities to provide students with guidance and helping tools for the lectures like [1] and [2]. To select the one fitting to our needs, four key issues have been proposed:

1. the algorithmic functionality
2. the user interface (GUI)
3. sound I/O
4. and the local computer environment

The first question is, what functionality is needed? In the end there should be at least some kind of sound produced by the computer, demonstrating the effect. This could be done by a simple playback of audio files, but there would be no dynamics enclosed - reacting to user input. In this simple approach parameter changes have to provided as different audio files.

Another requirement is to provide some adjustable parameters of the effect algorithm to the user. Of course, this should be in a handy and easy-to-use format like a graphical environment (GUI). The user can twist the knobs or move sliders up and down. The algorithm has to adapt these changes (almost) immediately.

After including some parameters, the next question is, what kind of audio source the algorithm will process. A generated sinusoid, a more complex but fixed melody or a complete audio file? The first two choices are easy to handle: build and pack everything what is needed into the code. But the latter one is more appropriate for demonstrating different audio effects. Every effect has its own *good* sounds where it behaves well and produces clearly its effect, and vice versa. In some effects there isn't any hearable difference when applied to single sinusoids, for example. So we have to provide some entry points for audio files in our application. The benefit of this effort is that the user can even use his own private and favored audio files.

So far we only handled the application side. Now we will come to the environmental side. Every application needs an operating system (OS) around it to provide some basic input/output functions. Most of this world is divided into two parts: the Microsoft one and the rest, called Unix/Linux (nowadays the Mac users belong to them). For use on the campus or even the World Wide Web, there is no *right* choice to be made. To be precise, for a wider range of users and user acceptance, one should at least serve these two OSs.

2.2. Implementational Issues

In the following we will compare different approaches to use in presenting digital audio effects. In addition to the four key issues above, there are more to keep in mind: as always, an algorithm or an application will not be static over time, so there is need for some maintenance. But the more complex the application, more time will be wasted for bug fixing and maintenance. Besides the easy-to-use issue we have to regard it to be *easy-to-program*.

Since there are several audio effects to implement in the future, all of them bare a common base: in its core every effect is a black box working as a simple input/output machine with some additional parameters. So there is a need to represent this common basic functionality in a programmable framework used for all forthcoming audio effects. We will investigate the possibilities of every approach in this sense, too.

2.3. Stand-alone application

The very first and (not mandatory) easiest way to do this is to write a single stand-alone application. You are not fixed to any programming language, but instead you can use anyone you like. Using this language for all audio effects, you can surely build up your own framework to use. You can even switch between different programming languages - if you want, but this will make it even harder to maintain the bunch of code in the future.

There are at least two drawbacks in writing your own application: there is no software or library already written for use in audio applications, for example, well-known applications like Steinberg Cubase or Adobe Audition. So you really have to do everything on your own, including handling driver problems with the audio hardware and building your own graphical environment from scratch. The second - and even bigger - drawback is the platform dependency: you have to maintain two (totally different) applications, one for each of the OSs. To circumvent this problem, one can use platform-independent programming tools like Sun's Java (as we will see later) or Microsoft's .NET.

2.4. Plugins

Why don't we use the well-known audio applications as the basis for input/output of audio files and only provide the audio effect as an optional plugin (or AddIn) inside this application? Well, instead of arguing with platform dependencies, now you have to handle different plugin-architectures. Favored architectures are, for example: Steinberg VST (Virtual Studio Technology) [5] or Microsoft DirectX.

Another possible drawback is the installation of a host application on every users PC, which provides this VST- or DirectX-interface. In our campus-orientation this is not applicable because in general there is no student who owns one of this general-purpose audio applications, and even there are few ones who are willing to do a test installation with a demo version.

All in all, we don't need this bunch of functionality these applications provide. Going back to a simple input/output system, we will find tools almost everyone will already have on his computer: audio players like Windows MediaPlayer, WinAmp or the Linux-correspondent XMMS. These tools provide all we need, and even use a plugin architecture to extend their functionality. Microsoft's MediaPlayer uses DirectX again, of course. WinAmp and XMMS share a different but common plugin framework. So again, we have to provide and maintain at least to different versions of the audio effect: one for the Microsoft Windows world, and one for the rest.

All of the plugin architectures provide a more or less sophisticated abstraction layer for the graphical environment. This way it is easier to build your own GUI, without programming the details and platform specific things. For example in Steinberg's VST a complete general purpose GUI is included (as far as the host application provide the whole interface!), and you simply have to announce the number of your parameters.

2.5. Macromedia Flash

Macromedia Flash is a browser plugin used in the World Wide Web to present multimedia content and to handle user interaction. This tool seems to fit perfectly our needs: platform-independent, small amount of user effort for installation of the flash-plugin (if not already done), easy handling of multimedia content (audio and video) and basic GUI elements for user interaction. But coming to the point: there is no possibility to interact directly with the multimedia content! You can arrange and playback different parts of audio/video files but you cannot process them through an effect algorithm.

So, besides all the positive properties of this technology, it violates our very first key issue. Therefore it is not usable for our purpose.

2.6. The *Processing* Language

Processing [6] is an open source and cross platform programming language for multimedia content. It is based on the Java programming language, but with new graphics and utility API, and it is tailored to the specific needs of processing and presenting multimedia content. Even being a rapidly growing environment, it currently doesn't support processing of audio content. You can, of course, playback audio files (i.e. wav-files), but there is no entry point for your algorithm to process the audio samples. Maybe this will be an interesting alternative in the future.

2.7. *MATLAB* implementation

The well-known engineering environment *MATLAB* is always a good idea when designing an algorithm. Programming is done in a script-like manner, and debugging is very easy due to its interpretative nature.

Despite these obvious advantages this is not an interactive tool and there is no way to use it over the internet or some other remote protocols in an easy-to-use manner. The tutorials are intended to give a very brief overview of what can be done with digital audio effects without having to know anything about audio samples, programming and data representation. So we will spend no further time for a deeper investigation on this field. (For audio effects using *MATLAB* see for example [7], [8], [9] and [10].)

2.8. Java Applets

Finally, driven by the platform-dependencies of all the other technologies, we come to the Java programming language. Its purpose is to provide a platform-independent layer, a so called runtime-environment, on which you can build up your specific application. Again you can choose to build stand-alone applications written in

Java. Alternatively - coming back to the World Wide Web - you can develop small applications, called applets, and include them directly on a web page.

We choose the latter one because there is no need for the user to download or install anything on the computer. Regarding our campus-driven development, this is a big issue, since the administrative access to university-PCs is prohibited for normal users.

Keeping in mind the other 3 key features:

- Adding effects to an audio file:
  we have to grab some source of sound, process the audio sample with an effect algorithm and play them back to the soundcard.
- The graphical environment:
  we can build using Java libraries like Swing for GUI elements.
- Using arbitrary audio files:
  opening and closing of files either from the web server or from the local file system.

When trying to access the local file system, Java will prevent this at first hand due to the strict security settings of applets. To circumvent this problem, one can provide some kind of certification to the applet. The author certifies his code and the user can trust or deny this certificate. Applying the certificate the access to the file system will be granted, denying it the applet can only access the files stored at the remote web server. Albeit it is still usable, only restricted to some predefined sound files. The full functionality can only be retrieved by downloading the applet and running it on a local web page or by accepting the provided certificate.

In the following section we will describe our basic framework implemented in Java. All recent and future audio effects built for our web page are based on this framework as a common base.

3. JAVA IMPLEMENTATION

So far we have implemented the following audio effects, that can be found on on our website
`http://ant.hsu-hh.de/jdafx`

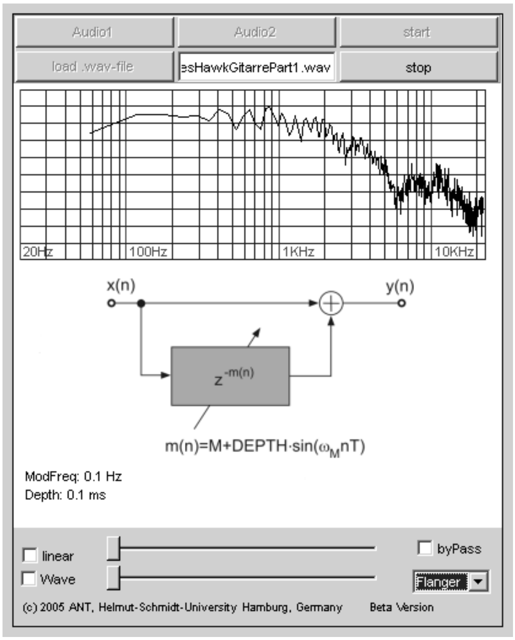
- **Quantization**
  Demonstrates audio effects resulting from Quantization. It is designed for a first insight into the perceptual effects of quantizing an audio signal. It is a Quantizer with optional Dither and Noise Shaping.
- **Audio Filters**
  This applet demonstrates audio effects resulting from Audio Filters. The following filter functions can be selected: Low-Pass/High-Pass, Shelving Filter and Peak Filter
- **Nonlinear Signal Processing**
  Audio effects resulting from signal distortion with a nonlinear transfer function.
- **Psychoacoustics**
  It is designed for a first insight into the perceptual experience of masking a sinusoidal signal with band-limited noise.
- **Delays**
  This applet demonstrates audio effects resulting from delay-based algorithms. One amplitude-modulation and three delay-based audio effects can be selected: tremolo, vibrato, chorus, and flanger.
- **Fast Convolution**
  An applet calculating a room impulse response using the fast convolution technique. The user can adjust the shape of the impulse response by the amplitude and the exponential decay of a random signal.

For a more specific view, we will describe some details of our *Delay Applet* (Figure 1).

3.1. The Delay Applet

The code in an audio effect applet is divided into two main parts: a common base class (controlling the audio data flow and managing the GUI elements) and the algorithm class (depending on the effect itself). The interface between these classes descibe all the dependencies like parameters used in the GUI, or processed sample values by the effect algorithm.

Depending on the context of the algorithm, there may be several helper classes, i.e. for calculations with complex values or computing the FFT [11].



ModFreq: 0.1 Hz
Depth: 0.1 ms

(c) 2005 ANT, Helmut-Schmidt-University Hamburg, Germany Beta Version

Figure 1: *Screenshot of the Delay applet.*

3.2. The GUI

The current GUI used in the DAFx applets has three components: buttons for loading and playing audio files, graphical display of algorithm information, and controls for algorithm settings. The objective of this structure is to provide an easy to use interface that can process an audio file while varying algorithm parameters. The graphical display of algorithm information is a central component of the GUI: it can represent block diagrams, frequency response curves, or any graphical view representing parameters modifications.

This GUI structure is an answer to some key features originally discussed in section 2: as a target we want to demonstrate an audio effect by playing and processing an audio file, and offer the user the possibility to adjust algorithm parameters. The option of playing a predefined audio sample, or processing a local audio file are also considered.

3.2.1. Playing audio files

The first GUI component is then composed of several buttons for loading and playing audio. There are two predefined files that can be downloaded from the main server. The idea of this feature is to prepare for the user special audio examples that can be used for each specific algorithm. For instance, the vibrato and other delay effects can be conveniently experienced with clean guitar music sounds. A plain sine wave is more adequate for understanding a quantization and dithering procedure. Psychoacoustic masking phenomena can be tested with band limited noise. The playback of each of these predefined audio files can be activated or stopped with buttons in the upper side of the applet graphic interface. An additional button for loading local audio files allows the user to experiment with its own audio material.

3.2.2. Visual Presentation

The second GUI component displays algorithm information. Each algorithm has different characteristics that are useful to represent graphically. In addition to the listening experience we have then a diagrammatic explanation of the audio effect. For instance, in the equalization applet the frequency response of each filter is the main graphical interface output. For the case of the quantization or delays applets, the block diagram of the algorithms are used as visual representations. For some audio effects, plotting the spectrum or waveform makes the understanding of the algorithm functionality easier.

3.2.3. Control elements

The third GUI component are sliders, knobs and other controls used for the interaction with the algorithm. Modifying the parameters of the algorithm has an auditive and graphical interface effect. In the case of the equalization applet, sliders can be used to modify the filter behavior. This modification is perceived almost immediately in the processed sound, and represented graphically in the plot of filter frequency response.

Notice that in this case different slider behaviors are available. On the one hand, a linear scale allows controlling linearly the filter frequency position. On the other hand, a logarithmic behavior is also available, as it is more convenient for handling low frequency limits.

In the case of the quantization applet, we have option knobs that can be used to select different algorithm configurations: among others we have quantization with or without dithering or quantization with or without noise shaping. When the user selects a special algorithm configuration, the graphical interface is updated by showing the corresponding block diagram.

```
jSlider1.addChangeListener( this );
jSlider1.setMaximum( 200 );
jSlider1.setValue( 0 );
```

All the handling of GUI elements is event based. The corresponding event handler for the sliders is included in the applet class as follows:

```
public void stateChanged( ChangeEvent e )
{
  JSlider source = (JSlider)e.getSource();

  if ( source.equals(jSlider1) ) {
    // automatic filter correction
    double scale = (double)
            jSlider1.getValue();
    mAlgorithm.setParameter( scale );
  }
  ...

  // destroy the drawing
  // (paint() will make a new one)
  repaint();
}
```

3.3. Java Sound

Sound routing and processing is included in the Java libraries since the very first Java version 1.0. But this first implementation only supports 8 kHz sampling rate. It is not usable for presenting audio effects.

Few things have been added in version 1.1, but since version 1.3 several enhancements have been made to the core libraries. Now different (and higher) sampling rates are supported, and all the classes managing the different I/O interfaces - described in the next section - have been introduced.

A new technology - the Java Media Framework (JMF) - gives rise to new possibilities in processing and presenting multimedia content. Since the JMF is an enhancement to Java, it is not included into the basic runtime environment. We don't consider it here, because the user has to make some additional installations of the JMF.

The audio administration routines in the DAFx applets are performed with three main structures

- `AudioSystem`
  The `AudioSystem` is a class that acts as an entry point for the sampled-audio system resources. In particular this class allows loading a specific audio file requested by the user.
- `AudioInputStream`
  The `AudioInputStream` is the class responsible for reading the requested audio data.

- `DataLine`
  The `DataLine` is an interface that prepares, together with the `AudioSystem` class, a line target for the audio (a soundcard or a loudspeaker).

Additionally the algorithm class is a fourth structure playing a main role in a DAFx applet. This class encapsulates the main processing function, and all required mechanisms for the actual algorithm. For example, in the equalization applet a filter class supports the algorithm by administrating different filter structures (low pass filters, high pass filters, Shelving filters and peak filters).

```
// get the file
inputStream = AudioSystem.
        getAudioInputStream( inputFile );

// open AudioInputStream
AudioFormat format =
        inputStream.getFormat();

// open a dataLine as target
// (soundcard/speaker)
DataLine.Info info = new DataLine.Info
        ( SourceDataLine.class, format );

SourceDataLine dataLine = null;
dataLine = (SourceDataLine)
        AudioSystem.getLine( info );
dataLine.open( audioFormat );
dataLine.start();
```

As soon as the audio input is loaded with the `AudioSystem`, a `DataLine` is prepared to be used with the audio output. A loop is then activated, and successive chunks of audio data are read by the `AudioInputStream`.

```
// allocate memory for the audio data
ByteBuffer data =
        ByteBuffer.allocate( BLOCKSIZE );
data.order( ByteOrder.LITTLE_ENDIAN );

ShortBuffer shortData =
        data.asShortBuffer();
byte[] byteData = audioData.array();

// read the samples blockwise
nBytesRead = inputStream.
        read( byteData, 0, BLOCKSIZE );
```

The applet algorithms are sample-based, although the supporting Java classes are block-based [12]. In reducing the block-size, the overhead in processing time is increased, but the introduced block delay gets lower. So there is a tradeoff between available CPU and processing resources and the hearable delay. For the sake if simplicity we are working with a constant blocksize of 128 samples.

All the internal calculations are done in floating point arithmetic. So the integer values - 2 bytes in little-endian order - have to be transformed in a normalized floating point value, and vice versa afterwards.

```
// the main process algorithm block
// is now done for this data block
for ( int n=0; n<nBytesRead/2; n++ ) {
  short sampleIn = shortData.get( n );

  double outValue = mAlgorithm.process
          ( sampleIn/32768 ) * 32768;

  shortData.put( n, (short)outValue );
}
```

The processed audio is then sent to the `DataLine`, and the final result is an audio output flow played by the loudspeaker.

```
// write the processed block
if ( nBytesRead >= 0 ) {
  int nBytesWritten = dataLine.
        write( byteData, 0, nBytesRead );
}
```

3.4. Design patterns: Strategy pattern

The object oriented features of the java language allows to design flexible code for the dafx applets. For instance the delays applet has several algorithms that can be selected at run time. The corresponding graphical interface, as slider positions, need to be updated accordingly.

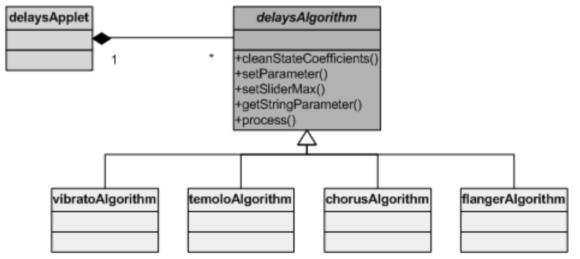


Figure 2: *The Strategy Pattern.*

For this case, a *strategy* design pattern (Figure 2) allows a simple and clean implementation of these features [13]. An abstract class representing the algorithm is handled by the graphical interface classes. Each specific algorithm is implemented as an instance of this abstract class. Due to the abstraction, and the common interface, it is then possible to conveniently switch between several algorithms dynamically.

```
abstract class delaysAlgorithm {
  public abstract void
           cleanStateCoefficients();
  public abstract void
           setParameter( double input );
  public abstract String
           getStringParameter();
  public abstract double
           process( double entry );
}
```

```
class tremoloAlgorithm
extends delaysAlgorithm {
  public void cleanStateCoefficients()
  { ... }
  public void setParameter( double input )
  { ... }
  public String getStringParameter()
  { ... }
  public double process( double input )
  { ... }
}
```

3.5. Design patterns: Mediator pattern

The communication between several classes is an important issue when handling multiple algorithms in the same applet. It is necessary to avoid a tangled class structure, and instead it is better to build a clean design where the code is easy to maintain.

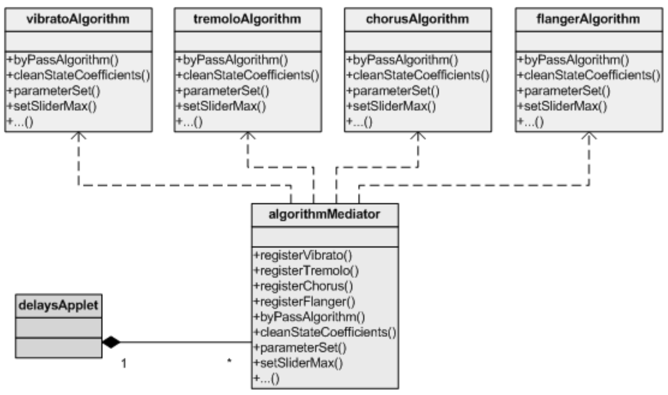


Figure 3: *The Mediator Pattern.*

In the previous example of the delays applet, several algorithms are interacting with the graphical interface. For this situation a *mediator* design pattern (Figure 3) can be used as a communicator between the graphical interface classes and the algorithms [13]. When the user selects an option in the applet interface (e.g., the user wants to bypass the algorithm), the mediator is updated. All algorithm instances are then notified via the mediator. In this way the communication complexity is concentrated in the mediator class. We can then maintain the code with a minimum amount of changes, keeping the same design and class communication flow.

4. CONCLUSIONS AND OUTLOOK

We have presented a basic but extendable framework for implementing digital audio effects. Our effects can be used online - directly through the internet - and they are interactive. No additional hard- or software except an internet browser and a soundcard is needed.

So far we have implemented several applets for use in research and teaching. And still there is much more to do. The applets and their source code are freely available for everyone interested in participating the development. Licence details and contact addresses can be found on our website
`http://ant.hsu-hh.de/jdafx`.

The next steps include converting the common Java base classes into a general purpose effect framework as in [14]. It should be possible to derive a new effect class directly from the framework, without knowing all the internal processes.

5. REFERENCES

[1] Udo Zoelzer, *Digital Audio Signal Processing*, John Wiley & Sons, New York, 1997, ISBN 0-471-97226-6.

[2] Udo Zoelzer et al., *DAFX - Digital Audio Effects*, John Wiley & Sons, New York, 2002, ISBN 0-471-49078-4.

[3] Nicola Bernardini and Davide Rocchesso, "Making Sounds with Numbers: A Tutorial on Music Software dedicated to Digital Audio," in *Proc. of the COST G-6 Conference on Digital Audio Effects (DAFX-98)*, Barcelona, Spain, Nov. 1998.

[4] Nicola Bernardini, Damien Cirotteau, Free Ekanayaka, and Andrea Glorioso, "Making Sound with Numbers, six years later," in *Proc. of the COST G-6 Conference on Digital Audio Effects (DAFX-04)*, Naples, Italy, Oct. 2004.

[5] Steinberg, *VST 3rd Party Developer Support*, http://ygrabit.steinberg.de/.

[6] Ben Fry and Casey Reas, *Processing*, http://processing.org/.

[7] Fernando A. Beltrán, Jos R. Beltrán, Nicolas Holzem, and Adrian Gogu, "Matlab Implementation of Reverberation Algorithms," in *First COST-G6 Workshop on Digital Audio Effects (DAFX98)*, Barcelona, Spain, Nov. 1998.

[8] Yusuf Jafry, "A modular realtime PC-based Audio Processing Tool for Efect Developers, Engineers, Musicians and Educators," in *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*, Verona, Italy, Dec. 2000.

[9] Joseph Timoney, Thomas Lysaght, Marc Schoenwiesner, and Lorcn Mac Manus, "Implementing Loudness Models in MATLAB," in *Proc. of the COST G-6 Conference on Digital Audio Effects (DAFX-04)*, Naples, Italy, Oct. 2004.

[10] Bob L. Sturm, "MATConcat: An Application for exploring concatenative Sound Synthesis using MATLAB," in *Proc. of the COST G-6 Conference on Digital Audio Effects (DAFX-04)*, Naples, Italy, Oct. 2004.

[11] Robert Sedgewick AND Kevin Wayne, *Introduction to Computer Science*, http://www.cs.princeton.edu/introcs/home/.

[12] Daniel Arfib, "Different Ways to write Digital Audio Effects Programs," in *Proc. of the COST G-6 Conference on Digital Audio Effects (DAFX-98)*, Barcelona, Spain, Nov. 1998.

[13] James W. Cooper, *Java Design Patterns: A Tutorial*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] Pablo Fernandez-Cid, Javier Casajus, and Lino Garcia, "A Java Framework for FX Development," in *Proc. of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*, Verona, Italy, Dec. 2000.