New Audio Features 0.1

Synthesis, Playback and Storage

By
Ismael Mosquera Rivera

To my friend Xavier Amatriain, with my best wishes.

Abstract

In this article, some new audio features added to the Java IMR-LIB library are described:

– Envelope: An ADSR envelope.
– Oscillator: An oscillator suitable to generate sinusoidal signals oscillating at any frequency.
– RawDataPlayer: A player to reproduce raw audio data, for instance, the one generated by an oscillator.
– RawDataStorage: A class useful to save raw audio data in an audio file.
- Wave: An abstract class from derive waves of any type.
- SawWave: Derived from Wave.
- SineWave: Derived from Wave.
- TriangularWave: Derived from Wave.
- MusicalNote: A 4-harmonic musical note derived from wave.
- Synthesizer: An abstract class which has an envelope and a Wave object.
- Carrier: Derived from Synthesizer; has a SineWave object and its envelope.
- MusicalInstrument: Derived from Synthesizer; has a MusicalNote object.
- Modulator: Modulates a signal using another wave.
- FMInstrument: Has a Carrier and a Modulator object to do the task.
- WaveType: Class with public fields useful to set the desired kind of wave.
- FrameFactory: Class useful to produce data frames.

We are going to explain in some detail these new audio features, and we also announce that we ar already thinking in more functionallity to add to our library.

Contents

1 Description

Since we added some features to our Java IMR-LIB, we will explain here how they were done. It means that each time we add new functionallity, an article like this one will be publishd in the ‘imrlib’ repository.

Following, we are going to describe the capabilities offered by the new classes integrated in our library.

- **Envelope.java:** The class Envelope implements an ADSR envelope. An envelope can be useful to set a concrete timbre, for instance, to a signal. In our case, an ADSR envelope also known as a 4 segment envelope since it has 4 stages: attack, decay, sustain and release; each stage is determined by a segment.

- *Attack:* From 0 up to the amplitude of the enveloped signal.
- *Decay:* From the end of attack down to the sustain level value.
- *Sustain:* from the end of decay, and keep constant until the begin of release.
- *Release:* From the end of sustain, down to 0.

Several parameters are needed to build such an envelope: amplitude, ADSR values, sample rate, sustain level and duration; according to these values, we can build the wanted envelope.

- *Amplitude:* A value in the range [0..1].
- *ADSR:* Values for each stage expressed in percentage respect to the duration.
- *Sample rate:* Number of samples taken by second.
- *Sustain level:* Value to keep constant during the sustain stage.
- *Duration:* Value expressed in ms. (Milliseconds).

Once all of these values are set, we build the required ADSR envelope ready to use. We apply it to a signal using the ‘apply’ method. We are able to set each value independently through the use of its concrete method and our envelope will be updated.

- **Oscillator.java:** The oscillator class generates sinusoids oscillating at any frequency. Its main parameters are:

- Amplitude: Value in the range [0 .. 1].
- Frequency: Oscillating frequency for the sinusoidal wave.
- Sample rate: Number of samples taken by second.

Each of the values for these parameters can be get and set using methods of this class. When some parameter is changed, the current sinusoid is updated.

To get the signal data, we pass a byte[] frame of any length to be filled with the correct information according to the parameters already explained.

We can, for example, apply an envelope to a signal generated by an oscillator so, if we combine an oscillator with an envelope we can get different timbres for our synthesized souns.

- Wave.java: This is an abstract class from we can derive any kind of wave. In fact, it has all the functionallity already implemented in order to synthesize a wave so, we must only implement its 'buildWave' protected method to get the wanted result. To implement such a method, you have to code just a single oscillation of your desired wave; the method to fill the synthesized data just replicates that single oscillation n-times, since we are talking about periodic signals.

So, the Wave abstract class has all of its public methods already implemented; getters and setters for its main parameters (amplitude, frequency and sample rate). And a method to get the synthesized data:

```
public int get(byte[] frame)
```

Which also returns the number of bytes read from its parameter.

- **SawWave.java:** This is a subclass of Wave implementing a saw wave; as we already said before, we only needed to make a suitable implementation for the 'protected abstract void buildWave()' method in order to get our desired kind of periodic wave.

- **SineWave.java:** This class is also a subclass of Wave; the way to get the wanted result is the same that in the case of SawWave. In this case, we have just to implement its 'buildWave' protected method for a single sinusoid oscillation.

- **TriangularWave.java:** Also a subclass of Wave, in this case implementing a periodic triangular wave.

- **MusicalNote.java:** Another class derived from Wave which produces a 4-harmonic musical note.

We will see later, that this set of classes is useful in the implementation of other kind of components due to its polymorphic behaviour.

- **Synthesizer.java:** This is a convenient abstract class which encapsulates an Envelope object and also a Wave one. Like as in the case of Wave, this abstract class implements all the functionallity needed to manage some kind of wave; in addition, it has an envelope which can be applied to its wave.

The only thing that you must do to subclass this abstract class is to set the kind of wave what you want for your synthesis.

You do not need to worry about updating the encapsulated envelope because it is automatically performed inside the class; nevertheless, you can set some parameters of the envelope: (amplitude, ADSR and sustain level).

As we mentioned before, the following method is all what you have to implement to build a Synthesizer object encapsulating your desired kind of wave with its associated envelope:

```
protected abstract void setWave(float amp, float freq, float sr);
```

A possible implementation for that method can be:

```
protected void setWave(float amp, float freq, float sr)
{
_wave = new SineWave(amp, freq, sr);
}
```

It was so easy, wasn't it?

- **Carrier.java:** This class was implemented for semantic reasons which we clarify soon. It has a sinusoid as wave type and also an envelope encapsulated in its superclass; you can decide to apply or not the envelope, which method 'applyEnvelope' is already implemented in its parent class.

- **MusicalInstrument.java:** This is also, as in the case of Carrier, a subclass of Synthesizer. The kind of wave that encapsulates is a MusicalNote object; remember that such a class implements a 4-harmonic musical note so, in combination which its associated envelope can be used as a musical instrument.

- **Modulator.java:** This is, as in the case of Carrier, a class implemented, in part, for semantic reasons; we say that because it has also an encapsulated envelope object, but this is not an abstract class; in addition to the mentioned envelope, it has also a Wave object; the difference, is that in this case, the wave can be one of the folloeing types:

- SawWave.
- SineWave.
- TriangularWave.

by default, it is a SineWave, but you can change the wave type by its 'setWaveType' method.

That class, as you have already guess, modulates a signal with the kind of wave set to that purpose.

```
public void modulate(byte[] data)
```

like in the case of the Synthesizer class, you can get and set the amplitude, frequency and sample rate parameters for the Wave object, and the amplitude, ADSR and sustain level for its envelope.

Now, we are in time to explain the semantic reasons mentioned before; that is because with a Carrier and a Modulator object, we are able to perform frequency modulation.

- **FMInstrument.java:** This class has a Carrier and a Modulator object; the carrier has an envelope and a sine wave object and the modulator can be configured to have a saw, sine or triangular wave in addition of its envelope to modulate the carrier. We can say that it is a two operators frequency modulator. You can get both the Carrier and Modulator objects independently and set its parameters according to your needs.

To get your synthesized signal there is the following method:

```
public void synth(byte[] data)
```

which fills the byte array passed as parameter with the information for your frequency modulated signal.

- **FrameFactory.java:** This is a convenient class to produce empty frames with a concrete duration. You can use this class to build your needed data frames according to your desired duration expressed in ms. (milliseconds). You can just passed the sample rate value and the number of milliseconds for your frame time duration to its static method:

```
public static byte[] getFrame(int sr, int ms)
```

and you will get an empty data frame with all its values set to zero so, you can produce also silence data frames at a concrete duration in milliseconds.

Maybe you can be interested in make clones for some data frames; the FrameFactory class has also a method to do that:

```
public static byte[] cloneFrame(byte[] frame)
```

as you can see, the implementation for this class is so simple, but it can be so useful too.

.- **WaveType.java:** This class has only 3 constants in order to set the wave type, for example, in a Modulator object; they are the following ones:

```
public static final int wSaw = 0;
public static final int wSine = 1;
public static final int wTriangular = 2;
```

which purpose is self explanatory.

- **RawDataPlayer.java:** This class is so useful to play data produced by any of the presented wave generators added to our library. You just need to pass the sample rate as parameter to its constructor (the default is (44100).

Once you instantiate a RawDataPlayer object, you have to call its ‘start’ method first, then you must generate some kind of audio signal and pass such a data to its ‘play’ method; finally, when all the data is played you can call its ‘stop’ method.

You can follow that steps again to playback more data frames; do not worry about closing the audio device, since this class uses a

javax.sound.sampled.SourceDataLine for playback and that class implements the Autocloseable interface.

An example of use can be as follows:

```
RawDataPlayer player = new RawDataPlayer();
Player.start();
// produce an audio frame
player.play(frame);
player.stop();
```

this is a so simple example, remember that you must wait until all the data is completely played; normally, you will play several frames, maybe inside a loop, when the last frame is passed to the player for playbacdk, you cannot call the 'stop' method immediately, but wait for finish playing that last frame; you can do that in different ways, a possible one is to call the Thread.sleep() method.

This class has also a method to retrieve its javax.soun.sampled.AudioFormat object:

```
public AudioFormat getFormat()
```

which can be very useful in some cases.

- **RawDataStorage.java:** This class allows you to store raw audio data produced, for instance, by any of the synthesis components already presented. You are able to store your audio data in the following file formats:

- WAVE (*.wav).
- AIFF (*.aif).
- AU (*.au).

It use is so simple; you have just to instantiate an object of this class passing a sample rate value to its constructor (the default value is 44100). Once the RawDataStorage object is instantiated, you can produce audio data frames by some way and then, add those frames to be accumulated in its internal buffer; after adding all of the wanted data, you have to call its 'store' method in order to save your data in an audio file in any of the supported audio formats; if you atempt to store your data in an unsupported audio file format, that method will thrown a javax.soune.sampled.UnsupportedAudioFileException. When you wanted to start adding new data for storing another file, you must call its 'reset' method first to discard the old data and set the internal buffer to its first position.

An example of use can be as follows:

```
RawDataStorage saver = new RawDataStorage();
// produce some raw audio data frame
// and add it to the saver
saver.add(frame);
// continue producing and adding more frames
// save your data in an audio file
saver.store("myaudiofile.wav");
// maybe you can continue storing the same data in another audio format ...
```

```
saver.store(“myaudiofile.aif”);
// call the reset method to begin adding data to a new different audio file
saver.reset();
// ...
// ...
```

as you can see, its usage is so simple and its functionallity can be very useful.

We already presented the new added components to our library; we did it briefly, but just to make an overview of their capabilities. In the ‘imrlib’ folder there is an archive called ‘doc.zip’; if you unzip that file, you will see a ‘doc’ directory with all the API documentation to consult more in depth the functionallity offered by all the classes in our library. In addition, you can run the examples and pick up inside the source code to get a more comprehensive knowledge about each subject.

2 Conclusions

We added a number of new components to our library, mainly, to perform audio synthesis. All of these classes were tested, and some examples to illustrate the capabilities offered by this new set of classes were added to the examples folder.

This is the third paper documenting the 'JAVA IMR-LIB' capabilities. As we already said, each time that we add some new features, an article explaining how they work will be added to the 'imrlib' repository.

We hope that with all the available stuff about the subjects covered by our library, an interested reader can take profit from this work; we also are thinking to adding more useful features soon; meanwhile, take in account that, in addition to this paper, you have all the API documentation for the entire library updated and placed in the 'doc.zip' archive; you also can run the examples and pick inside the source code. We will be so glad if you enjoy this effort made for free.

3 Appendix

In this brief appendix, we just are going to talk about the way to run the new examples added to the folder were they are. It is so easy, since all the examples are already compiled; anyway, we encourage you to look at the readme.txt files to know about them.

We added three more examples in order to demonstrate the functionallity offered by the new integrated code:

- test_oscillator: This example shows the functionallity offered by the class Oscillator applying also an envelope to the generated data. The example plays the C2 scale and, finally, stores the data in an audio file.

To test the example, you just have to execute the 'run.bat' batch file if you are under a MSWindows command line, or the 'run.sh' basic script in case that you are using a Linux like shell.

- test_musical_insttrument: This example, mainly, demonstrates the capability of the MusicalInstrument class by reproducing a fragment of the musical piece called 'Fur Elise' which we think is known by everybody. As in the case of the oscillator example, when that fragment is synthesized and playback, an audio file is stored in the bin folder. To run this example, just follow the steps explained for testing the oscillator.

- test_fm_instrument: This example shows how works a simple FMInstrument class using two operators: a carrier and a modulator; you can improve that implementing such an instrument but with more modulator signals modulating the carrier.

When you run the example, in the same fashion that in the already presented ones, a little bit fragment of 'Star Wars' will be played and, afterwards, an audio file storing the result will be saved inside the 'bin' folder.

About The Author

Ismael Mosquera Rivera has a degree in Computer Science by Pompeu Fabra University http://www.upf.edu, Barcelona – Spain, 2004. He also was involved in the Clam project, http://www.clam-project.org (C++ Library for Audio and Music); unfortunately, he is affected by a severe eye disease (Retinitis Pigmenthosa), which caused him to get totally blind a number of years ago. Anyway, he finally decided to get back to the computer world, so, he installed a screen reader in his old computer. The one he installed is only available for Windows OS, but he got the MinGW compiler which has also a command line shell from where you can have a little Linux in your Windows system. So, all the textual information in a computer can be read, and he began to do research. Among the things he has done until that, the development of this new added features encouraged him to write a paper explaining how it was done, and this is the result. He apologizes the reader to the lack of graphics, diagrams and identation in the source code. Anyway, his expectation is that an interested reader can take benefit from this article.