Music Programming in Minim

John Anderson Mills III
Université de Mons, TCTS
31 Bvd Dolez
Mons, Belgium
nodog@konfuzo.net

Damien Di Fede*
Kokoromi
619B W 35th St
Austin, Texas, USA
dcomposer@gmail.com

Nicolas Brix
Université de Mons, TCTS
31 Bvd Dolez
Mons, Belgium
nicolas.brix@gmail.com

ABSTRACT

Our team realized that a need existed for a music programming interface in the Minim audio library of the Processing programming environment. The audience for this new interface would be the novice programmer interested in using music as part of the learning experience, though the interface should also be complex enough to benefit experienced artist-programmers. We collected many ideas from currently available music programming languages and libraries to design and create the new capabilities in Minim. The basic mechanisms include chained unit generators, instruments, and notes. In general, one "patches" unit generators (for example, oscillators, delays, and envelopes) together in order to create synthesis algorithms. These algorithms can then either create continuous sound, or be used in instruments to play notes with specific start time and duration. We have written a base set of unit generators to enable a wide variety of synthesis options, and the capabilities of the unit generators, instruments, and Processing allow for a wide range of composition techniques.

Keywords

Minim, music programming, audio library, Processing, music software

1. INTRODUCTION

Processing[6] is an open-source, textual programming environment which was originally focussed on both allowing visual artists easy access to programming as well as providing quick prototypes for visualizing ideas. It has grown to be a full development environment and has a large community of users. One can find much more information at `http://www.processing.org`, including tutorials, downloads, and forums. Many people have contributed to Processing since it began, creating libraries which extend the capabilities of Processing.

Minim[3] is one such library and is open-source as well. Minim extends the capabilities of Processing to include audio capabilities and is distributed with the standard installation of Processing. The previous version of Minim, 2.0.2, provides many capabilities for adding sound to sketches, or the individual programs, in Processing, but we felt that Minim did not provide adequate capabilities for a programmer with musical goals.

*Damien Di Fede is the original creator of Minim.

We decided to add a music programming interface to Minim. We reviewed several different music programming languages, including Csound[1], SuperCollider[4], ChucK[9], JMusic[7], and Beads[2]. We also reviewed the graphical music prgramming software packages Max/MSP[10], and PureData[5]. We took a mix of our own ideas and the ideas from these languages, libraries, and software to try to make music programming easy for the novice programmer working within the Processing environment.

Processing is basically a simple Java development environment with some extra capabilities, therefore many of the benefits and drawbacks associated with Java are characteristics of Processing as well. Processing permits one to quickly develop programs, standalone applications, and web applets. By adding music programming capabilities to Minim, we would also be creating a simple way to produce music programs for download and web publication.

The purpose of this paper is to present the new capabilities of Minim to a techncally knowledgable audience by describing the design decisions made and some resulting consequences. This paper is not intended as the user documentation for the software.

This paper does assume a basic knowledge of electronic music concepts, Processing, Java, and object-oriented programming. For the rest of the paper, Minim will refer to the newly developed version which includes the music programming interface described here. Programming constructs will be referred to as follows. Class names will be capitalized. Objects instantiated from classes will start with a lowercase letter. Class methods will be followed by parentheses. Finally, in this paper, by music, we mean all sound art.

The examples presented in this paper are simple. This choice was made for brevity and clarity. Both simple and complex examples are provided with the Minim library.

Section 2 describes the design process for Minim's music programming interface. Section 2.1 describes the framework of Minim. Section 2.2 describes the benefits and drawbacks of working within Processing and Java for this task. Section 2.3 describes the unit generators, or UGens, which were created for Minim. Section 2.4 describes the extra capabilities of Minim which make it easy to use for the novice programmer. Section 2.5 describes different compositional techniques facilitated by Minim. Finally section 3 will describe how Minim performed, availability, and the future work we intend.

2. MUSIC PROCESSING IN MINIM

Fortunately, version 2.0.2 of Minim is a stable audio library for the Processing language. This gave us a good basis to begin our new work. Minim version 2.0.2 uses buffered audio and provides audio to the Java Sound API[8], though Minim version 2.0.2 does have facilities to use other low level drivers.

On October 13 and 14, 2009, we conducted a workshop in which we determined the audience and scope of our work. We decided that the main audience would be sound artists who are new to programming and want to use Processing to learn programming. We determined that the music programming interface should keep those artist-programmers interested as their skills grew. We also wanted to provide a complete music programming environment for visual artists using Processing wanting to complement their visual art with sound. Our audience is referred to from this point as the artist-programmer.

In that same design workshop we decided on the basic framework which will be described in section 2.1. A decision was made to calculate audio signals and control signals at the same rate. This decision is similar to a design decision made in the ChucK audio programming language. This decision also indirectly limits the number of simultaneous sounds Minim can provide in real-time applications and is discussed further in section 2.2.

Before going into section 2.1 about the choices we made on the framework for Minim, it is important to describe the basic structure of a non-trivial Processing sketch. These sketches are usually built around two methods, `setup()` and `draw()`. `setup()` is called once at the beginning of the execution of a sketch and contains initialization code. After `setup()` has finished, `draw()` is called repeatedly during the execution of a sketch, at a nominal rate of 60 frames/s. It is the `draw()` method which allows Processing to present animation, but the timing of the calls to `draw()` is not fast or consistent enough to ensure the sample accuracy often required by music.

2.1 Music Programming Framework

The first decisions made were that synthesis would be driven by unit generators, or UGens, which each did a specific small task and could be connected together. Unit generators exist in almost every music programming language in one form or another. Section 2.1.1 describes our design decisions about UGens and `patch()`ing in detail.

We also made the design decision that we wanted to be able to make Instruments and use them to play notes. This mechanism is not inherently a part of every music programming languages (for example, ChucK and Beads do not have a strong idea of instruments) though it is a part of many (for example, Csound and JMusic). Section 2.1.2 describes our design decision about Instruments and notes in detail.

The Java Sound API imposes that AudioOutputs are either stereo or monaural. If one wishes to use more than two channels of audio concurrently, one must create more than one AudioOutput and process them as multiple stereo pairs. With this in mind, we have built our framework with no more than two channels of control. The audio signal is actually handled in Minim by an array whose size is determined by the AudioOutput and is therefore not limited to two channels. The UGens which control audio imaging, however, never provide for more than stereo sound at this point.

2.1.1 UGens and Patching

The decision was made to build synthesis up from UGens. This is a common technique in music programming languages. These unit generators include sound generators, like noise and tone generators; effects, like delays and filters; and other tools, like envelopes. Section 2.3 will describe the

```
lFO1.patch( sineOsc.frequency );
lFO2.patch( sineOsc.amplitude );
sineOsc.patch( gain ).patch( out );
```

Figure 1: Example code showing the `patch()`ing of one Oscil (lFO1) into the frequency input of another Oscil (sineOsc), the `patch()`ing of another Oscil (lFO2) into the amplitude input of the sineOsc Oscil, and finally the `patch()`ing of the sineOsc Oscil through a Gain UGen (gain) to the AudioOutput (out).

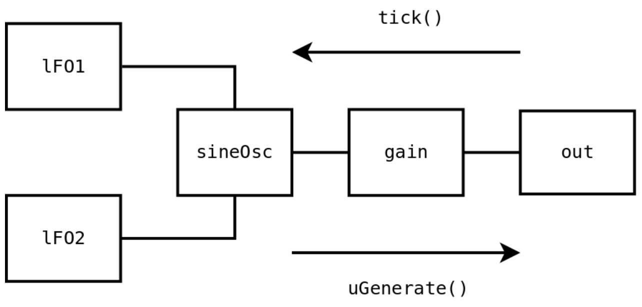

Figure 2: A graphical representation of the `patch()`ing specified in figure 1. The direction of audio processing by the `tick()` method and resulting cascade direction of the results of `uGenerate()` are shown here also.

UGens that have been created thus far.

The method of connecting the UGens together varies considerably from one language to another. For Minim, we decided that we wanted to be able to `patch()` the UGens into one another using a Java method. We quickly realized that this meant that the UGens would need to have some number of "inputs" beyond audio. For example, an oscillator UGen would need to have input control for at least frequency. Figure 1 shows how `patch()`ing occurs in Minim, and figure 2 shows a graphical representation of the `patch()`ed chain created in figure 1.

AudioOutput is the point in where Minim changes how audio is generated. The AudioOutput class in Minim provides buffered audio to the Java Sound API, but the UGens generate their audio sampleframe by sampleframe. A sampleframe is one sample from each of the signal channels (monaural or stereo). AudioOutput is the class which asks the `patch()`ed chains of UGens to provide single sampleframes enough times to fill a buffer. The generation of audio is therefore output (AudioOutput) driven. The mechanism for doing this generation is the `tick()` and `uGenerate()` methods that exist in every UGen.

To create audio, the AudioOutput `tick()`s every UGen connected to it as an input. Every `tick()`ed UGen then `tick()`s all of its inputs up the chain until a UGen is reached which has no inputs. That UGen then uses its `uGenerate()` method to generate a single sampleframe of values, and returns that sampleframe to the UGen which `tick()`ed it. Once the `tick()` method in a particular UGen has collected sampleframes from all of its inputs, it then returns its own sampleframe. The AudioOutput collects the final summed value of sampleframes of all UGens connected to it and places that sampleframe in the buffer. The AudioOutput then moves time forward and starts the process again. Once the buffer is full, AudioOutput gives the buffer to the Java Sound API. Pseudocode for the `tick()` and `uGenerate()`

```
tick()
{
  for all of my inputs
    input.tick()
  uGenerate()
}
```

Figure 3: Pseudocode for the `tick()` and `uGenerate()` method placement.

```
public void noteOn( float dur )
{
   toneEnv.setDampTimeFromDuration( dur );
   toneEnv.activate();
   noiseADSR.noteOn();
}
public void noteOff()
{
   noiseADSR.noteOff();
}
```

Figure 4: Example code for an Instrument's `noteOn()` and `noteOff()` methods. In this example, the damping time of an amplitude envelope is being set using the duration of the note and then that amplitude envelope is being `activate()`ed. The ADSR envelope for noise is having its `noteOn()` and `noteOff()` methods called from the respective Instrument methods.

method placement is shown in figure 3. The flow of the the `tick()` and `uGenerate()` calls is shown in figure 2

Multiple outputs for a particular UGen are also permitted. For any UGen which has N connections to its output, that UGen only `uGenerate()`s a new sampleframe once every N `tick()`s. This mechanism keeps the generation of values for sampleframes at the same rate as the AudioOutput sampling rate. This mechanism also prevents infinite loops between a UGen's input and output.

Using just chains of UGens `patch()`ed together, an artist-programmer can create continuous sound using Minim. This is an acceptable way to use Minim, especially for algorithmic and/or interactive composition. Section 2.1.2 describes the additional mechanism of Instruments which allow the artist-programmer to control the music as notes.

2.1.2 Instruments and Notes

The sound generating mechanisms described in section 2.1.1 can be thought of as the underlying synthesis algorithms. Many musicians think of music in terms of notes played on Instruments and combinations of those notes. The Instrument interface allows the artist-programmer to create Instruments and use them to play notes given a specific start delay and duration.

Certainly the concept of instruments is not new in music programming and is exhibited by other languages such as Csound and JMusic. The particular mechanic that we designed is that the artist-programmer creates their own Java classes which implement our Instrument interface. Basically, this means is that the user class must include a `noteOn()` and `noteOff()` method which specify how the Instrument behaves when it is told to begin and end its note. Figure 4 shows example code for the `noteOn()` and `noteOff()` methods of an Instrument. The appendix shows the Processing code for a complete Instrument.

```
out.playNote( 2.0, 2.9, "C3" );
```

Figure 5: Example code for creating a note using the `playNote()` method. This creates a note using the DefaultInstrument with a start time of 2.0 beats from the time of the method call, a duration of 2.9 beats, and a pitch of C3.

```
out.pauseNotes();
out.setTempo( 90.0 );
out.setNoteOffset( 2.0 );
float vol = 0.33;
out.playNote( 0.00, 0.2,
   new ToneInstrument( "E5", vol, out) );
out.playNote( 0.25, 0.2,
   new ToneInstrument( "E5", vol, out) );
out.playNote( 0.75, 0.2,
   new ToneInstrument( "E5", vol, out) );
out.playNote( 1.25, 0.2,
   new ToneInstrument( "C5", vol, out) );
out.playNote( 1.50, 0.2,
   new ToneInstrument( "E5", vol, out) );
out.playNote( 2.00, 0.2,
   new ToneInstrument( "G5", vol, out) );
out.playNote( 2.75, 0.2,
   new ToneInstrument( "G4", vol, out) );
out.resumeNotes();
```

Figure 6: Example code showing complex `playNote()` calls, composition helper methods, and traditional composition.

`noteOn()` is actually `noteOn( float )`. It is worth remarking that the duration of note is passed to the Instrument. This is useful, for example, as is shown in figure 4 to set envelopes based on the duration of the note.

The actual calling of the `noteOn()` and `noteOff()` methods are handled by an unseen NoteManager object which exists by default in the Minim library. One gives the appropriate information to the NoteManager by using the `playNote()` method of an AudioOutput as is shown in figure 5. NoteManager runs in a separate thread in order to maintain sample accuracy of the sound.

In the most complex case, the appropriate information includes a start delay, a duration, and the Instrument object used to play the note. Notes of this type will be seen in figure 6. The reason we chose a start delay rather than a start time was because it is more general and is appropriate when generating music in a real-time application. Start delay and start time are equivalent if all notes are added at once.

It is also useful at times to establish communication methods between Instruments. This is relatively easy to do by first creating a method, for example `changeBehavior()`, which alters the behavior of a ReceivingInstrument and then giving the a reference to a ReceivingInstrument object to another Instrument. That reference can be used to call the `changeBehavior()` method of the ReceivingInstrument object, and change its behavior.

In writing the examples to demonstrate compositional techniques using Minim, we created several helper methods in order to facilitate composition. `setTempo()` sets the actual length of one beat and is 60.0 beats per minute by default. `setNoteOffset()` adds an offset to the start delay of all notes and is 0.0 beats by default. `setDurationFactor()` gives a multiplication factor to the duration of the notes specified and is 1.0 by default. `pauseNotes()` pauses the updating of time in the NoteManager so that all notes added until the `resumeNotes()` method is called are guaranteed to have accurate relative timing. The `pauseNotes()`–`resumeNotes()` mechanism only guarantees that everything between these method calls will have accurate timing, and that the block of notes may not be timed accurately with any notes outside of it. Examples of these helper methods can be seen in figure 6.

The two modes of generating sound can be intermixed as well. For example, Instruments can make a final `patch()` to a continually running Summer, a UGen which sums all of its inputs, instead of an output. This can be thought of as multiple audio channels running into an effects bus.

2.2 Benefits of Java and Processing

There are several advantages to working with the Java language. The flexibility of object-oriented programming does stand out. Being able to create multiple class constructors for Instruments facilitates easy-of-use programming, such as default values, and easy inclusion of backward compatibility in the case of changing the Instrument. The ability to create loops and new classes which create notes themselves is a huge benefit to algorithmic composition as discussed in section 2.5.2.

Several advantages of using the Processing environment also exist. It is extremely easy to link visual and audio art together. When one wishes a particular note to be associated with a particular visual, mouse, or keyboard event, one simply creates a note (as in figure 5) with a start delay of 0.0 beats in the `draw()`, `mouseMoved()`, or `keyPressed()` Processing method respectively. In this situation, the timing issues are between an event and a sound, unlike the timing situation described in section 2. Video features can be changed according to the audio being generated by passing variable references to Instruments. Processing also facilitates the creation of web applets from a sketch, so publishing a sketch as a web applet using Minim is simple.

It should be remarked that when combining the visual and audio capabilities of Processing and Minim, the latency between what is seen on the screen and what is heard from the speakers can be perceptible. One can choose the buffer size used by the Java Sound API to be small (<1024), although smaller buffers do come at the increased risk of the program not being able to produce sampleframes fast enough to fill the buffers in the AudioOutput. This tradeoff limits the number of UGens which can be active simultaneously. When not performing the music in real time, these issues are not present, for example, when recording to a file.

For our examples on modern computers, the number of simultaneously available UGens has been sufficient with one caveat. Instruments should `patch()` to the AudioOutput in their `noteOn()` method and `unpatch()` from the AudioOutput in the `noteOff()`, as demonstrated by the example Instrument in the appendix. This `patch()`ing and `unpatch()`ing restricts the calculation of the sampleframes from the UGens in the Instrument to the time when the Instrument is actually generating sound. The choice to do this, however, is left to the artist-programmer, though highly suggested for complex compositions.

2.3 UGens

This section describes the UGens which have already been created for Minim as of April 2010. The "controls" listed under many of the UGens are the parameters that can be connected to the output of another UGen.

As stated in section 2, control signals and audio signals are treated equivalently in Minim. Further discussion is warranted here. Control signals are generally expected to be monaural and audio signals are expected to be monaural or stereo. At this point, we do not provide for stereo control signals, but is a possibility for future work.

2.3.1 *Sound Generators*

When creating UGens, one of the most important types is those that actually produce audio signals. The following UGens are these sound generators.

Noise
: generates noise of several different tints: white, pink, and red/brown.

Oscil
: repeats a waveform at a specified frequency and amplitude. Oscil can also be used as a low frequency oscillator for control of other UGens.
controls: amplitude, frequency, phase

LiveInput
: passes sampleframes from the audio input of the computer.

FilePlayer
: pulls sampleframes from a specified file.

2.3.2 *Effects*

Many of the UGens can be seen as effects processors. They receive some sort of incoming signal and change the nature of it.

Delay
: repeats a delayed version of the incoming signal.
controls: delayTime, delayAmplitude

Pan
: takes a mono signal and specifies a stereo position for that signal.
controls: pan

Balance
: attenuates the left or right channel of stereo signal.
controls: balance

Gain
: attenuates or amplifies the incoming signal.
controls: gain

IIRFilter
: filters the spectrum of the incoming signal. This can be implemented as any one of the following filters: BandPass, ChebFilter, HighPassSP (single pole), LowPassSP (single pole), LowPassFS (four stage), or NotchFilter.
controls: cutoff (effectively center frequency for BandPass and NotchFilter)

BitCrush
: reduces the bit resolution of the incoming signal.
controls: bitRes

WaveShaper
: uses the incoming signal as the index to a Wavetable. (Wavetables are discussed in section 2.4.) controls: mapAmplitude, outAmplitude

2.3.3 *Envelopes*

To turn the audio generated by sound generators into notes, it is practical to specify an amplitude envelope which defines the note. These envelopes are not inherently based at 0, so they can also be used as control for other UGens for which 0 is not a usual value, such as the frequency of an Oscil.

Line
: produces a line from one point to another over a specified time.

ADSR
: produces an attack-decay-sustain-release envelope.

Damp
: produces an attack-decay envelope.

Oscil
: can be used as an amplitude envelope if the frequency is set to a value on the order of $\frac{1}{P}$ where P in the duration of the note.

GranulateSteady
: produces steady granular-synthesis grains from the input signal.
controls: grainLen, spaceLen, fadeLen

GranulateRandom
: produces random granular-synthesis grains from the input signal.
controls: grainLenMin, spaceLenMin, fadeLenMin, grainLenMax, spaceLenMax, fadeLenMax

2.3.4 *Mathematics*

Due to our decision to use the `patch()`ing mechanism as described in section 2.1.1, it is not simple to use the arithmetic functions built into Java directly to change the values of signals. We needed to implement math directly as UGens.

Constant
: generates a constant value as a signal.

Summer
: adds (sums) all incoming inputs.

Multiplier
: multiplies an incoming signal by amplitude.
controls: amplitude

Reciprocal
: generates the reciprocal of the incoming signal.

Midi2Hz
: generates the equivalent frequency in Hertz for an incoming signal given as a MIDI note number. The MIDI note number does not need to be an integer.

2.4 Ease of Use

Based on the decisions made in the workshop, we created several classes to facilitate ease of use for music programming in Minim. One of these, the DefaultInstrument, was created specifically with the beginning programmer in mind. The `playNote()` method will use the DefaultInstrument if the artist-programmer has not specified another Instrument. Admittedly, there is a step in the learning curve between using this DefaultInstrument and moving into Instrument design, but the goal was to make the initial steps easy, and then instruct with good documentation.

The Frequency class was also made in order to make converting between different representations of a note's frequency. Using the Frequency class, the artist-programmer can specify the frequency in Hertz, MIDI note number, or pitch name (as a String) and receive the frequency in Hertz or MIDI note number. The DefaultInstrument, along with Frequency and a few helper methods in AudioOutput, allow the `playNote()` commands of figure 7 to be valid and, in fact, equivalent. The Frequency class is shown in the example Instrument in the Appendix.

The final class which was created specifically for ease of use was Waves. In order to describe Waves it is first necessary to state that in Minim, oscillators currently repeat Waveforms. A Waveform is an interface that simply requires that the value over the Waveform can be found by

```
out.playNote( 0.0, 1.0, 261.63 );
out.playNote( 0.0, 1.0, "C4" );
out.playNote( 0.0, 261.63 );
out.playNote( 0.0, "C4" );
out.playNote( "C4" );
out.playNote( 261.63 );
out.playNote( "" );
out.playNote();
```

Figure 7: Example `playNote()` calls which use the DefaultInstrument.

an `at()` method which has a float argument between 0 and 1. The Wavetable class is one implementation of the Waveform interface. A Wavetable is effectively an array which returns an interpolated value based on the index of the `at()` method. Waves is a class which includes many ways to build Waveforms using Wavetables.

With the Waves class, the artist-programmer has access to a perfect sine, triangle, saw, or square wave in a Wavetable. The artist-programmer can also create a Wavetable based on

- a perfect triangle, saw, or square wave with one of the zero-crossings shifted in time,
- the first N harmonics of a triangle, saw, or square wave,
- the first N harmonics or N odd harmonics with random amplitudes,
- random noise,
- N impulses placed randomly in silence,
- or a weighted combination of these.

Using the WavetableGenerator class, the artist-programmer can also build Wavetables using function generators similar to several of those found in Csound.

2.5 Compositional Techniques

With music programming languages, the mechanisms built into the language affect the artist-programmer's ability to create compositions. We wanted to allow as many different styles of composition as possible, so as to not limit the artist-programmer. The ways in which we have thought to compose are listed here. We are certain that other artist-programmers will discover new ways as they use Minim.

Minim permits traditional composition techniques (every note is known beforehand), algorithmic composition techniques (notes are determined by algorithms and can be aleatoric in nature), and interactive composition. All of the following compositional techniques can be either generated in real-time by running the Processing sketch, or recorded to a file, by running the sketch and using Minim's AudioRecorder class.

2.5.1 *Traditional Composition*

In what we label as traditional composition techniques, all notes to be played are known before any sound is generated. Thus far we have created this type of composition by specifying many calls to the `playNote()` method in the `setup()` method of a Processing sketch. These `playNote()` calls are effectively the score for the music. An example of this can be seen in figure 6. The ToneInstrument shown in the appendix would be used to play the notes created by the `playNote()` calls in figure 6.

2.5.2 *Algorithmic Composition*

Minim has several different mechanism for facilitating algorithmic composition whether the Instrument interface is used or not. Several of these techniques for altering the compositional material of the music include, but are not limited to

- adding random variables, and conditional and loop statements to the traditional composition technique mentioned in section 2.5.1,
- writing new classes which make calls to `playNote()`, or UGen or custom Instrument methods,
- allowing events in Processing's `draw()` or other methods to call `playNote()`, or UGen or custom Instrument methods,
- using Instrument communication techniques mentioned in section 2.1.2, and
- using the envelope UGens, specifically GranulateSteady and GranulateRandom, with grain lengths near note lengths.

2.5.3 Interactive Composition

Interactive composition can be achieved directly by the use of the many mechanisms in Processing for interaction, such as the mouseMoved() and keyPressed() methods, to control aspects of UGen behavior or Instrument behavior. If aspects of visual art were being controlled interactively, those interactions can also create interactive compositions more indirectly, by calling `playNote()` or controlling the behavior of Instruments or UGens. Any composition which uses the LiveInput UGen will inherently be interactive.

3. CONCLUSION

By combining aspects of many different music programming languages and libraries already in existence and identifying a need in Minim, our team created a useful tool for making music using the Processing environment. We hope that our work facilitates the process for those musicians interested in learning to program, and we look forward to feedback.

We have worked diligently to provide extensive documentation and examples of the Minim music programming framework and UGens. The techniques described in this paper can be found in the examples available with Minim. Minim is available for download from the internet at `http://code.compartmental.net/tools/minim/`.

We intend to maintain and expand the Minim library. Some of the upcoming plans include, but will not be limited to a score file; exponential lines; and compression, reverb, equalizer, sample-hold, pitch-shift, and stereo-control UGens. Minim is a work in progress and under current development. Aspects of the library may change, but we will endeavor to keep the functionality the same until it becomes infeasible to do so due to advances in the interface or technology.

4. ACKNOWLEDGMENTS

John Anderson Mills III is supported by numediart (`http://www.numediart.org`) a long-term research program centered on Digital Media Arts, funded by Région Wallonne, Belgium (grant N°716631). We would like to thank numediart for supporting the workshop in October of 2009 which allowed this project to commence.

5. REFERENCES

[1] R. Boulanger. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming.* MIT Press, Cambridge, MA, USA, 2000.

[2] O. Brown. About beads. `http://www.beadsproject.net/`, Feb. 2010.

[3] D. Di Fede. Minim. `http://code.compartmental.net/tools/minim/`, Apr. 2010.

[4] J. McCartney. Supercollider: A new real time synthesis language. In *Proc. International Computer Music Conference (ICMC'96)*, pages 257–258, 1996.

[5] M. Puckette. Pure data: another integrated computer music environment. In *Proc. the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[6] C. Reas and B. Fry. Processing: a learning environment for creating interactive web graphics. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Web Graphics*, pages 1–1, New York, NY, USA, 2003. ACM.

[7] A. Sorensen and A. R. Brown. Introducing jmusic. In *InterFACES: Proceedings of The Australasian Computer Music Conference. Brisbane: ACMA*, pages 68–76, 2000.

[8] Sun Microsystems, Inc. Java sound api. `http://java.sun.com/products/java-media/sound/`, Feb. 2010.

[9] G. Wang and P. R. Cook. Chuck: a concurrent, on-the-fly audio programming language. In *Proc. ICMC*, pages 219–226, 2003.

[10] D. Zicarelli. An extensible real-time signal processing environment for max. In *Proceedings of the 1998 International Computer Music Conference*, pages San Francisco: ICMA—466, Ann Arbor, Michigan, USA, 1998.

APPENDIX

An example of a complete Instrument is presented here.

```
class ToneInstrument implements Instrument
{
  Oscil sineOsc;
  ADSR adsr;
  AudioOutput out;

  ToneInstrument( String note, float amplitude,
      AudioOutput output )
  {
    out = output;
    float frequency =
        Frequency.ofPitch( note ).asHz();

    sineOsc = new Oscil( frequency, amplitude,
        Waves.TRIANGLE );
    adsr = new ADSR( 1.0, 0.01, 0.01, 1.0, 0.02 );

    sineOsc.patch( adsr );
  }
  void noteOn( float duration )
  {
    adsr.patch( out );
    adsr.noteOn();
  }
  void noteOff()
  {
    adsr.noteOff();
    adsr.unpatchAfterRelease( out );
  }
}
```