MOZZI: INTERACTIVE SOUND SYNTHESIS ON THE
OPEN SOURCE ARDUINO MICROPROCESSOR

Tim Barrass
Independent Artist
Melbourne

ABSTRACT

Mozzi is an open source sound synthesis library for the Arduino microprocessor. Mozzi can be used to generate algorithmic music for an installation, wearable sounds for a performance, or interactive sonifications of sensors, on a small, modular and cheap Arduino, without the need for additional shields, message passing or external synthesiser hardware. This opens the way to increased creative uses of the Arduino and other compatible platforms. This paper describes software architecture that enables the synthesis of sounds from sensors in real-time on such a limited microprocessor, the programming API, and a variety of artistic and scientific applications that have been created by the Mozzi community.

1. INTRODUCTION

Sound art installations and performances typically involve a combination of laptops, MIDI synths, software packages and sensor systems connected by cables, wireless and MIDI or OSC protocols. These systems are technically complex with many points of potential failure, involve lots of expensive equipment, are not very mobile or robust, and have large power requirements. The open source Arduino microprocessor has provided a platform for artists and designers to create interactive installations and mobile interactive interfaces that are cheap, robust and can be programmed in a java based API specifically targeted to interactive sensor based systems. However, the Arduino API lacks any audio capability beyond the tone() command which produces a square wave beeping sound with no dynamics, and blocks the input from sensors and other processing while it is playing. This paper begins by locating Mozzi in relation to other software approaches to audio for the Arduino. It then highlights the main features of Mozzi which make it a viable synthesis toolkit of a kind which has not been available before. Then a broad outline of how Mozzi works is given in terms of its general framework. The outline is extended with an overview of Mozzi’s programming interface. Then two simple programming examples are described in detail to illustrate basic usage of the library. This is followed by a section which presents existing applications of Mozzi in art, music and education. Finally, options are considered for the future development, utilisation and community support for the open source Mozzi synthesiser.

2. BACKGROUND

The original Arduino was based on an embedded 8 bit microprocessor that can only do integer operations at rates of up to 16 MHz. This capability is very limited compared to computer music systems with floating point operations running at hundreds or thousands of times this speed. However, the uptake of the open source Arduino microprocessor by artists and designers has highlighted the need for sound as part of interactive digital media installations and performances. The recognition of this shortcoming has led to a number of efforts to develop interactive synthesis of sounds on the Arduino. However the limitations of the platform have lead some people to express the feeling that real-time interactive synthesis on the Arduino was not technically possible[1]. Nevertheless, Joe Marshall’s Octosynth[2] which was able to play sixteen oscillators at once provided evidence of what could be done with clever programming and a good understanding of the microprocessor architecture. Martin Nawrath demonstrated the capability to do digital signal processing (DSP) in real-time to create delays and buffering[3]. Bruce Land demonstrated DSP through fixed point arithmetic using integer operations for filters and synthesis[4]. These examples and code were illustrative but not designed for reuse. A previous attempt by Sofian Audry to create a general purpose synthesis library called Glade[5] has critical performance bottlenecks due to overuse of floating point maths, virtual functions and other issues and is no longer maintained.

3. WHAT IS MOZZI?

Mozzi is an open source software library which enables the Arduino microprocessor to generate complex and interesting sounds using familiar synthesis units including oscillators, samples, delays, filters and envelopes. Mozzi is modular and can be used to construct many different sounds and instruments. The library is designed to be flexible and easy to use, while also aiming to use the processor efficiently, which is one of the hurdles preventing this kind of project from succeeding until now. To give an idea of Mozzi’s ability, one of the example sketches which come with the library demonstrates fourteen audio oscillators playing simultaneously while also receiving real-time control data from light and temperature sensors without blocking.

Mozzi developed out of research into mobile sonification[6] when it became apparent that a naive approach to programming audio on Arduino would not provide satisfactory real-time performance. Many optimisation problems existed which had to be teased out one by one. This project is novel in that it has found solutions to the problems of affordable and easy embedded audio synthesis and has broken out of the sample-playback, single wave beeping paradigm widely accepted as embedded audio to date. This opens the way to increased creative uses of the Arduino and other compatible platforms.

Mozzi has the following features:

- 16384 Hz audio sample rate with almost-9-bit STANDARD and 14 bit HiFi output modes.
- Variable control rate from 64 Hz upwards.
- Familiar audio and control units including oscillators, samples, filters, envelopes, delays and interpolation.
- Modules providing fast asynchronous analog to digital conversion, fixed point arithmetic and other CPU-efficient utilities to help keep audio running smoothly.
- Readymade wave tables and scripts to convert sound files or generate custom tables for Mozzi.
- More than 30 example sketches demonstrating basic use.
- Comprehensive API documentation.
- Mozzi is open source software and easy to extend or adapt for specific applications.

4. ARCHITECTURE

4.1. User space

Mozzi inherits the concepts of separate audio and control rate processes directly from Csound[7] and Pure Data[8]. The interface between Mozzi and the Arduino environment consists of four main functions. These are startMozzi(), updateAudio(), updateControl() and audioHook(), visible in the “User space” section of Figure 1. All four are required for a Mozzi sketch to compile. Mozzi developed out of research into mobile sonification[6] when it became apparent that a naive approach to programming audio on Arduino would not provide satisfactory real-time performance. Many optimisation problems existed which had to be teased out one by one. This project is novel in that it has found solutions to the problems of affordable and easy embedded audio synthesis and has broken out of the sample-playback, single wave beeping paradigm widely accepted as embedded audio to date. This opens the way to increased creative uses of the Arduino and other compatible platforms.

Mozzi has the following features:

- 16384 Hz audio sample rate with almost-9-bit STANDARD and 14 bit HiFi output modes.
- Variable control rate from 64 Hz upwards.
- Familiar audio and control units including oscillators, samples, filters, envelopes, delays and interpolation.
- Modules providing fast asynchronous analog to digital conversion, fixed point arithmetic and other CPU-efficient utilities to help keep audio running smoothly.
- Readymade wave tables and scripts to convert sound files or generate custom tables for Mozzi.
- More than 30 example sketches demonstrating basic use.
- Comprehensive API documentation.
- Mozzi is open source software and easy to extend or adapt for specific applications.

4.1. User space

Mozzi inherits the concepts of separate audio and control rate processes directly from Csound[7] and Pure Data[8]. The interface between Mozzi and the Arduino environment consists of four main functions. These are startMozzi(), updateAudio(), updateControl() and audioHook(), visible in the “User space” section of Figure 1. All four are required for a Mozzi sketch to compile. Mozzi developed out of research into mobile sonification[6] when it became apparent that a naive approach to programming audio on Arduino would not provide satisfactory real-time performance. Many optimisation problems existed which had to be teased out one by one. This project is novel in that it has found solutions to the problems of affordable and easy embedded audio synthesis and has broken out of the sample-playback, single wave beeping paradigm widely accepted as embedded audio to date. This opens the way to increased creative uses of the Arduino and other compatible platforms.

Mozzi has the following features:

- 16384 Hz audio sample rate with almost-9-bit STANDARD and 14 bit HiFi output modes.
- Variable control rate from 64 Hz upwards.
- Familiar audio and control units including oscillators, samples, filters, envelopes, delays and interpolation.
- Modules providing fast asynchronous analog to digital conversion, fixed point arithmetic and other CPU-efficient utilities to help keep audio running smoothly.
- Readymade wave tables and scripts to convert sound files or generate custom tables for Mozzi.
- More than 30 example sketches demonstrating basic use.
- Comprehensive API documentation.
- Mozzi is open source software and easy to extend or adapt for specific applications.
MOZZI : INTERACTIVE SOUND SYNTHESIS ON THE OPEN SOURCE ARDUINO MICROPROCESSOR

Tim Barrass
Independent Artist
Melbourne

ABSTRACT

Mozzi is an open source sound synthesis library for the Arduino microprocessor. Mozzi can be used to generate algorithmic music for an installation, wearable sounds for a performance, or interactive sonifications of sensors, on a small, modular and cheap Arduino, without the need for additional shields, message passing or external synthesiser hardware. This opens the way to increased creative uses of the Arduino and other compatible platforms. This paper describes software architecture that enables the synthesis of sounds from sensors in real-time on such a limited microprocessor, the programming API, and a variety of artistic and scientific applications that have been created by the Mozzi community.

1. INTRODUCTION

Sound art installations and performances typically involve a combination of laptops, MIDI synths, software packages and sensor systems connected by cables, wireless and MIDI or OSC protocols. These systems are technically complex with many points of potential failure, involve lots of expensive equipment, are not very mobile or robust, and have large power requirements. The open source Arduino microprocessor has provided a platform for artists and designers to create interactive installations and mobile interactive systems that are cheap, robust and can be programmed in a java based API specifically targeted to interactive sensor based systems. However, the Arduino API lacks any audio capability beyond the tone() command which produces a square wave beeping sound with no dynamics, and blocks the input from sensors and other processing while it is playing.

This paper begins by locating Mozzi in relation to other software approaches to audio for the Arduino. It then highlights the main features of Mozzi which make it a viable synthesis toolkit of a kind which has not been available before. Then a broad outline of how Mozzi works is given in terms of its general framework. The outline is extended with an overview of Mozzi's programming interface. Then two simple programming sketches which comes with the library demonstrates environment consists of four main functions. These are startMozzi(), updateAudio(), updateControl() and audioHook(), visible in the “User space” section of Figure 1. All four are required for a Mozzi sketch to compile. startMozzi(control_rate) goes in Arduino's setup(). It starts the control and audio output timers, given the requested control rate in Hz as a parameter. updateControl() is where any analog input sensing code should be placed and relatively slow changes such as...
4.2. Under the Hood

Under the Hood, audioHook() is where audio synthesis code should be placed. This runs on average 16384 times per second, so updateAudio() runs as fast as possible. The buffer has 256 cells which equates to a maximum delay of about 15 milliseconds, to give leeway for control operations without interrupting audio output. The buffer is emptied behind the scenes by the regular 16384 Hz audio interrupt.

Mozzi employs pulse wave modulation (PWM) for audio output. This allows a single Arduino pin to be allocated for output, requiring minimal external components. Depending on the application, the output signal may be adequate as is. Passive filter designs to reduce aliasing and PWM carrier frequency noise are available on the Mozzi wiki[9] if required.

5. PROGRAMMING INTERFACE

Mozzi has a growing collection of classes for synthesis, modules containing useful functions, commonly used wave tables, and sampled sound tables. Mozzi includes Python scripts to convert raw audio files and templates which can be used to generate other custom tables.

Descriptions of the classes currently available are shown in Table 1. Modules are described in Table 2. Comprehensive documentation of the library is available online[10] and in the Mozzi download[11].

### 6. WRITING A MOZZI SKETCH

#### 6.1. Bare bones example: playing a sine wave

This section explains a minimal Mozzi sketch step by step. The sketch plays a sine wave at a specified frequency. Although there are abundant instances online of Arduino sketches performing this task, this example illustrates the structure and gist of a bare-bones Mozzi sketch. It does not assume much previous experience with Arduino programming.

First include MozziGuts.h. This is always required, as are headers for any other Mozzi classes, modules or tables used in the sketch. In this case an oscillator will be used, and a wavetable for the oscillator to play:

```cpp
#include <Oscil.h>
#include <tables/sin2048.intr.h>
```

The oscillator needs to be instantiated using literal numeric values as template parameters (inside the < > brackets). This allows the compiler to do some of the oscillator’s internal calculations at compile time instead of slowing down execution by repeating the same operations over and over while the program runs. An oscillator is declared as follows:

```cpp
Oscil <2048, AUDIO_RATE> aSin(SIN_DATA);
```

#### 6.2. Signal generation

The audio sine tone oscillator is created like this:

```cpp
Oscil <2048, AUDIO_RATE> aSin(SIN_DATA);
```

The control rate, like the audio rate, must be a literal number and power of two to enable fast internal calculations. It is not necessary to define it as follows, but it helps to keep programs legible and simple to modify.

```cpp
#define CONTROL_RATE 128
```

Now to the program functions. In Arduino’s setup() routine goes:

```cpp
void setup(){
  startMozzi(CONTROL_RATE);
  aSin.setFreq(440u);
}
```

This sets up one timer to call updateControl() at the rate chosen and another timer which works behind the scenes to send audio samples to the output pin at the fixed rate of 16384 Hz.

The oscillator frequency can be set in a range of ways, but in this case it will be with an unsigned integer as follows:

```cpp
aSin.setFreq(440u);
```

Now Arduino’s setup() function looks like this:

```cpp
void setup(){
  startMozzi(CONTROL_RATE);
  aSin.setFreq(440u);
}
```

The next parts of the sketch are updateControl() and updateAudio(), which are both required. In this example the frequency has already been set and the oscillator just needs to be run in updateAudio(), using the Oscil::next() method which returns a signed 8 bit value from the oscillator’s wavetable.

```cpp
void updateControl(){
  // no controls being changed
  int updateAudio(){
    return aSin.next();
  }
}
```

Finally, audioHook() goes in Arduino’s loop().

```cpp
void loop(){
  audioHook();
}
```

This is where the sound actually gets synthesised, running as fast as possible to fill the output buffer which

### Table 1. The current collection of Mozzi classes, with descriptions and the update rates of each.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Audio rate</th>
<th>Control rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oscil</td>
<td>Sample</td>
<td>play a wavetable, cycling</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>Phasor</td>
<td>play a wavetable, with extra controls</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>generates a high resolution ramp</td>
<td>yes</td>
</tr>
<tr>
<td>Filters</td>
<td>StateVariable</td>
<td>12db resonant lp, bp and notch</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>LowPassFilter</td>
<td>resonant low pass filter</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>Smooth</td>
<td>IIR low pass filter</td>
<td>yes</td>
</tr>
<tr>
<td>Envelope generators</td>
<td>ADSR</td>
<td>simple ADSR envelope generator</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Ead</td>
<td>exponential attack decay envelope</td>
<td>-</td>
</tr>
<tr>
<td>Delays</td>
<td>AudioDelay</td>
<td>delay for comb filter, flange, chorus and slabback</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>AudioDelay/FeedBack</td>
<td>audio delay with feedback</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>EventDelay</td>
<td>non-blocking replacement for Arduino's delay()</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>ReverbTank</td>
<td>simple recirculating reverb</td>
<td>yes</td>
</tr>
<tr>
<td>Synthesis/Distortion</td>
<td>WavePacket</td>
<td>overlapping streams of windowed sine wave grains</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>WaveShaper</td>
<td>maps input values through a table to output</td>
<td>yes</td>
</tr>
<tr>
<td>Interpolation</td>
<td>Line</td>
<td>efficient linear interpolator</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>Portamento</td>
<td>simple portamento for note-based applications</td>
<td>yes</td>
</tr>
</tbody>
</table>

1. The Mozzi download contains an example sketch comparing the audible results of different table sizes being played over a sweep of frequencies.
LFO's or frequency changes can be performed. An example of this is shown in section 5.2.

updateAudio() is where audio synthesis code should be placed. This runs on average 16384 times per second, so code here needs to be lean. There is also The only other strict requirement is that it returns an integer between -244 and 243 inclusive in STANDARD mode or -8192 to 8191 in HIFI mode.

audioHook() goes in Arduino’s loop(). It wraps updateAudio() and takes care of filling the output buffer, hiding the details of this from user space.

### 4.2. Under the Hood

Mozi uses hardware interrupts on the processor which automatically call interrupt service routines (ISR) at regular intervals.

startMozzi() sets up two interrupts, one for audio output at a sample rate of 16384 Hz and a control interrupt which can be set by the user at 64 Hz or more, in powers of two.

In STANDARD mode, the internal timers used by Mozi on the ATmega processors are the 16 bit Timer 1 for audio and 8 bit Timer 0 for control. HIFI mode additionally employs Timer 2 with Timer 1 for audio. Using Timer 0 disables Arduino time functions millis(), micros(), delay() and delayMicroseconds(). This saves processor time which would be spent on the interrupts and the blocking action of the delay() functions. Mozi provides an alternative method for scheduling (see EventDelay() in the API).

Audio data is generated in updateAudio() and placed in the output buffer by audioHook(), in Arduino’s loop(), running as fast as possible. The buffer has 256 cells, which equates to a maximum delay of about 15 milliseconds, to give leeway for control operations without interrupting audio output. The buffer is emptied behind the scenes by the regular 16384 Hz audio interrupt.

Mozi employs pulse wave modulation (PWM) for audio output. This allows a single Arduino pin to be allocated for output, requiring minimal external components. Depending on the application, the output signal may be adequate as it is. Passive filter designs to reduce aliasing and PWM carrier frequency noise are available on the Mozi wiki[9] if required.

### 5. PROGRAMMING INTERFACE

Mozi has a growing collection of classes for synthesis, modules containing useful functions, commonly used wave tables, and sampled sound tables. Mozi includes Python scripts to convert raw audio files and templates which can be used to generate other custom tables. Descriptions of the classes currently available are shown in Table 1. Modules are described in Table 2. Comprehensive documentation of the library is available online[10] and in the Mozi download[11].

### 6. WRITING A MOZZI SKETCH

#### 6.1. Bare bones example: playing a sine wave

This section explains a minimal Mozzi sketch step by step. The sketch plays a sine wave at a specified frequency. Although there are abundant instances online of Arduino sketches performing this task, this example illustrates the structure and gist of a bare-bones Mozi sketch. It does not assume much previous experience with Arduino programming.

First include MozziGuts.h. This is always required, as are headers for any other Mozi classes, modules or tables used in the sketch. In this case an oscillator will be used, and a wavetable for the oscillator to play:

```cpp
#include <Oscil.h>
#include <MozziGuts.h>
```

The oscillator needs to be instantiated using literal numeric values as template parameters (inside the <> brackets). This allows the compiler to do some of the Oscil's internal calculations at compile time instead of slowing down execution by repeating the same operations over and over while the program runs. An oscillator is declared as follows:

```cpp
Oscil<2048, AUDIO_RATE> aSin(SIN_DATA);
```

The audio sine tone oscillator is created like this:

```cpp
Oscil<2048, AUDIO_RATE> aSin(<SIN_DATA>);
```

The control rate, like the audio rate, must be a literal number and power of two to enable fast internal calculations. It is not necessary to define it as follows, but it helps to keep programs legible and simple to modify.

```cpp
#define CONTROL_RATE 128
```

Now to the program flow. In Arduino’s setup() routine goes:

```cpp
void setup(){
  startMozzi(CONTROL_RATE);
  aSin.setFreq(440u);
}
```

This sets up one timer to call updateAudio() at the rate chosen and another timer which works behind the scenes to send audio samples to the output pin at the fixed rate of 16384 Hz.

The oscillator frequency can be set in a range of ways, but in this case it will be with an unsigned integer as follows:

```cpp
aSin.setFreq(440u);
```

Now Arduino’s setup() function looks like this:

```cpp
void setup(){
  startMozzi(CONTROL_RATE);
  aSin.setFreq(440u);
}
```

The next parts of the sketch are updateAudio() and updateControl(), which are both required. In this example the frequency has already been set and the oscillator just needs to be run in updateAudio(), using the Oscil::next() method which returns a signed 8 bit value from the oscillator's wavetable. The int return value of updateAudio() must be in the range -244 to 243.

```cpp
void updateAudio(){
  // no controls being changed
  int updateAudio(){
    return aSin.next();
  }
```

Finally, audioHook() goes in Arduino’s loop().

```cpp
void loop(){
  audioHook();
}
```

This is where the sound actually gets synthesised, running as fast as possible to fill the output buffer which
gets steadily emptied at Mozzi’s audio rate. For this reason, it’s best to avoid placing any other code in loop(). It’s important to design a sketch with efficiency in mind in terms of what can be processed in updateAudio(). Keep updateAudio() lean, put slow changing values in updateControl(), and pre-calculate as much as possible in setup(). Control values which directly modify audio synthesis can be efficiently interpolated with a Lut() object in updateAudio() if necessary. The whole sketch is shown in Program 1.

```cpp
#include <MozziGuts.h>
#include <Oscil.h>
#include <tables/sin2048.Int.h>

#define CONTROL_RATE 128
Oscil <2048, AUDIO_RATE> aSin(SIN_DATA);
void setup(){
aSin.setFreq(440.0f);
float depth = 0.25;
void setup1(){
    kVib.setFreq(6.5f);
    aSin.setFreq(centre_freq + vibrato);
}
void updateControl(){
    float vibrato = depth * kVib.next();
    aSin.setFreq(centre_freq + vibrato);
}

Program 1. Playing a sine wave at 440 Hz.
```

6.2. Adding a control signal: vibrato

Vibrato can be added to the sketch by periodically changing the frequency of the audio wave with a low frequency oscillator. The new oscillator can use the same wave table but this time it is instantiated to update at control rate. The naming convention of using a prefix of k for control and a for audio rate units is a personal mnemonic, influenced by Csound.

```cpp
Oscil <2048, CONTROL_RATE> kVib(SIN_DATA);

This time the frequency can be set with a floating point value:
kVib.setFreq(6.5f);

Now, using variables for depth and centre frequency, the vibrato oscillator can modulate the frequency of the audio oscillator in updateControl(). kVib.next() returns a signed byte between -128 to 127 from the wave table, so depth has to be set proportionately.
```

```cpp
void updateControl(){
    float vibrato = depth * kVib.next();
    aSin.setFreq(centre_freq + vibrato);
}

The modified sketch complete with vibrato is listed in Program 2.
```

Program 2. Playing a sine wave with vibrato.

While this example uses floating point numbers, it is best to avoid their use for intensive audio code which needs to run fast, especially in updateAudio(). When the speed of integer maths is required along with fractional precision, it is better to use fixed point fractional arithmetical. The mozzi_fixmath module has number types and conversion functions which assist in keeping track of precision through complex calculations.

7. APPLICATIONS

Mozzi has been available on GitHub since June 2012. The range of potential applications has yet to be explored, however some examples which have appeared so far include:

- A musical fruit fly experiment for a science fair at The Edge in the State Library of Queensland. Kinetic and electronic artists Clinton Freeman, Michael Candy, Daniel Flood and Mick Byrne worked with Dr Caroline Hauwelk from the Queensland University of Technology to produce an interactive installation where people could play chords which represented the different resistances of a group of pieces of fruit infested with fruit flies. According to the project documentation, resistance is sometimes used as a measure of fruit quality[12].

- Teaching interface design at the Queen Mary University. One of the vehicles for learning is a musical instrument based on an electric guitar, exploring ergonomics with digital and analog inputs to a Mozi synthesizer[13].

- B.O.M.B. Beat Of Magic Box, a palm-sized interactive musical device by Yoshihito Nakashishi, designed for cooperative performance between novice participants. The devices communicate wirelessly and produce related evolving harmonic and rhythmic sequences depending on how they are handled[14].

- MIDI-based synthesizers using Mozi as a synthesis engine. One example is “[org] syn” by Václav Pelousk, founder of the Standuino hand-made electronic music project, with six voice polyphony, velocity sensitivity, envelopes, selectable wavetables, modulation and bit-logic distortions[15]. Others include Arduino Mozi synthesizer vX.0 by e-licktronic, a mono synth with selectable wavetables, LFO and resonant filtering[16], and the ironically humorous FM-based CheepSynth constructed and played by Dave Green and Dave Pape in a band called Faketen Polytechnic[17].

My own sound-sculptural object exhibited at PICA during ICMC 2013 also uses Mozi. This is a small solar powered sound-generating object which responds to changes in solar energy with 6 voice polyphonic audio synthesised in real time.

8. THE FUTURE OF MOZZI

Mozi is relatively young yet ripe for a community of open source development and practice to emerge around it. It’s easy to write new classes and to construct complete instrument sketches. Feedback from educators has shown that the library is able to be used by children. There is potential for porting the library to new Arduino platforms as they become available. As it is, there is already a long to-do list, including a variety of half-finished sound generators and instruments, and the ever-receding lure of creative work beyond the making of the tool.

Mozzi expands the possibilities for sonification and synthesis in new contexts away from expensive hardware, cables and power requirements. The low cost and accessibility of Mozi synthesis on open source Arduino and compatible microprocessors offers people a way to create applications and compositions adapted to a wide range of localised conditions.

9. REFERENCES


[14] Hauxwell from the Queensland University of Technology electronic artists Clinton Freeman, Michael Candy, Edge in the State Library of Queensland. Kinetic and interactive musical device by Yoshihito Nakashishi, designed for cooperative performance between novice participants. The devices communicate wirelessly and produce related evolving harmonic and rhythmic sequences depending on how they are handled[14].


[20] Hauxwell from the Queensland University of Technology electronic artists Clinton Freeman, Michael Candy, Edge in the State Library of Queensland. Kinetic and interactive musical device by Yoshihito Nakashishi, designed for cooperative performance between novice participants. The devices communicate wirelessly and produce related evolving harmonic and rhythmic sequences depending on how they are handled[14].


6.2. Adding a control signal: vibrato

Vibrato can be added to the sketch by periodically changing the frequency of the audio wave with a low frequency oscillator. The new oscillator can use the same control rate. The naming convention of using a prefix of kVib setFreq(6.5f);

Oscil <2048, AUDIO_RATE> kVib(SIN_DATA);

float centre_freq = 440.0;

void updateControl(){
  float vibrato = depth * kVib.next();
  aSin.setFreq(centre_freq+vibrato);
}

The modified sketch complete with vibrato is listed in Program 2.

Program 2. Playing a sine wave with vibrato.

While this example uses floating point numbers, it is best to avoid their use for intensive audio code which needs to run fast, especially in updateAudio(). When the speed of integer maths is required along with fractional precision, it is better to use fixed point fractional arithmetic. The mozzi.fixedmath module has number types and conversion functions which assist in keeping track of precision through complex calculations.

7. APPLICATIONS

Mozzi has been available on Github since June 2012. The range of potential applications has yet to be explored, however some examples which have appeared so far include:

- A musical fruit fly experiment for a science fair at The Edge in the State Library of Queensland. Kinetic and electronic artists Clinton Freeman, Michael Candy, Daniel Flood and Mick Byrne worked with Dr Caroline Haukwell from the Queensland University of Technology to produce an interactive installation where people could play chords which represented the different resistances of a group of pieces of fruit infested with fruit flies. According to the project documentation, resistance is sometimes used as a measure of fruit quality[12].

- Teaching interface design at the Queen Mary University. One of the vehicles for learning is a musical instrument based on an electric guitar, exploring ergonomics with digital and analog inputs to a Mozi synthesizer[13].

- B.O.M.B.-Beat Of Magic Box-, a palm-sized interactive musical device by Yoshihito Nakanishi, designed for cooperative performance between novice participants. The devices communicate wirelessly and produce related evolving harmonic and rhythmic sequences depending on how they are handled[14].

- MIDI-based synthesisers using Mozi as a synthesis engine. One example is “[or] syn” by Václav Peloušek, founder of the Standuino hand-made electronic music project, with six voice polyphony, velocity sensitivity, envelopes, selectable wavetables, modulation and bi-logic distortions[15]. Others include Arduino Mozi synthesizer vX.0 by e-lctronic, a mono synth with selectable wavetables, LFO and resonant filtering[16], and the ironically humorous FM-based CheapSynth constructed and played by Dave Green and Dave Pape in a band called Fakebit Polytechnic[17].

My own sound-sculptural object exhibited at PICA during ICMM 2013 also uses Mozi. This is a small solar powered sound-generating object which responds to changes in solar energy with 6 voice polyphonic audio synthesised in real time.

8. THE FUTURE OF MOZZI

Mozzi is relatively young yet ripe for a community of open source development and practice to emerge around it. It’s easy to write new classes and to construct complete instrument sketches. Feedback from educators has shown that the library is able to be used by children. There is potential for porting the library to new Arduino platforms as they become available. As it is, there is already a long to-do list, including a variety of half-finished sound generators and instruments, and the ever-receding lure of creative work beyond the making of the tool.

Mozzi expands the possibilities for sonification and synthesis in new contexts away from expensive hardware, cables and power requirements. The low cost and accessibility of Mozi synthesis on open source Arduino and compatible microprocessors offers people a way to create applications and compositions adapted to a wide range of localised conditions.

9. REFERENCES


A JAVA-BASED REMOTE LIVE CODING SYSTEM FOR CONTROLLING MULTIPLE RASPBERRY PI UNITS

Oliver Bown
Design Lab
University of Sydney
oliver.bown@sydney.edu.au

Miriama Young
Music Department
University of New South Wales
miriama.young@gmail.com

Samuel Johnson
Design Lab
University of Sydney
sjoh7452@uni.sydney.edu.au

ABSTRACT

Cheap embedded devices create new opportunities for networked, distributed, generative or remote-controlled music. In this paper we present a simple audio programming environment designed to run realtime, remote live-coded audio on a low-cost completely wireless hardware setup consisting of a Raspberry PI, a WiFi dongle, a speaker and a battery pack. Audio is processed in realtime using the Beads library for realtime audio in Java, running on Oracle’s distribution of Java for embedded devices. Code is remotely injected in realtime by sending Java class files over a socket connection to a dynamic class loader, which instantiates and runs the classes. We describe the system and its capabilities, and give an example of a performance that utilises this system. This paper is accompanied by a musical performance at ICMC 2013.

1. INTRODUCTION

The range and availability of cheap embedded devices has been accelerating in recent years, exciting audio developers with the growing plausibility of use-cases involving multiple low-cost audio computers. The release of the Raspberry PI in 2012 marked a significant development in this area. The Raspberry PI is a $35 computer capable of running a full Linux distribution, loaded from an SD card. Importantly it has USB ports and an analogue audio output. For a total cost of around $100 the PI can be expanded to include speaker, battery pack and WiFi dongle, turning it into a fully portable networked music generator (Figure 1). Alternatively, for roughly the same price it can be expanded with an audio dongle, switches and knobs to become a programmable effects pedal.

In this paper we consider the first configuration. In this configuration the PI has no audio input, but can be interacted with remotely over a network connection to control the realtime generation of sound. Although not necessary for most distributed music applications, the fact that the system is completely wireless extends its range of possible uses, given that it can be incorporated into miscellaneous portable objects. The ability to hide the device in objects without telltale hidden wires is also conducive to a sense of enchantment in interaction design. Finally the PI can be easily extended to further include other kinds of inputs and outputs, such as sensors and lights. This range of uses and interactive experiences is already covered by programmable smartphones but not at the same cost, programmability and configurability of the PI-based system.

The PI is easily configured so that on power-up, it automatically logs in, connects to a given WiFi network with a known network address, and loads an audio application. Thus, given a collection of PI modules, each module is ready to respond to dedicated control messages once activated.

Figure 1. Portable wireless configuration consisting of Raspberry PI (centre), Moshi Bassburger self-powered rechargeable speaker (top), WiFi dongle (left of PI) and battery pack (right). Data is stored on an SD card (right of PI).