CMLS HW3 - Design and implementation of a Computer Music System

Xenharmonics Group:
Baroli Gabriele
Ferrando Alessandro
Mauri Noemi
Passoni Riccardo

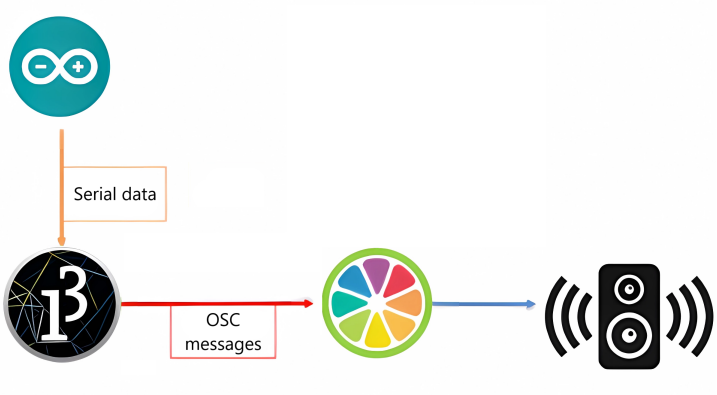
Kinetic Harmonizer

Abstract

Kinetic Harmonizer is an effect plugin developed with the JUCE framework and partially controlled by a triple-axis accelerometer.
It implements an harmonizer through a variable delay line, the cent shift in pitch is controlled by the accelerometer's movements in real time, employing interaction design techniques in its development.
The interactive section of the GUI has been implemented in Processing.

0.1 Introduction

The plugin that's been developed is an audio effect, more specifically it's an harmonizer, implemented through a delay line. It allows for the precise control of each parameter that influences the pitch and placement in time of the delayed signal in relation to the input audio.

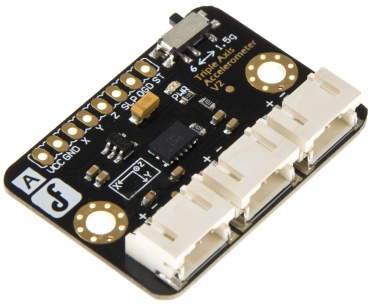


Interaction is an essential part of this plugin, the main parameters of the effect (the pitch shift in semitones/cents and the dry-wet mix) are controlled by an accelerometer. The data collected by the accelerometer is processed by the Arduino microcontroller, which then is passed to Processing via Serial communication. The Processing sketch uses this data to update the visualizer in real time, then it routs the signals to the plugin via OSC, natively supported in JUCE. We'll now explain how the data is collected in this next section of the report.

1 Interaction System Unit - Board and Accelerometer

1.1 MMA7361 Triple-Axis Accelerometer

The MMA7361 is a triple-axis accelerometer, meaning it gathers acceleration data for each one of the three dimensions of space: x, y and z. It's manifactured by Freescale, a company specialized in microcontrollers and analog sensors. Indeed, the MMA7361 outputs a 3.3V analog voltage for each of the three outputs, corresponding to the three available coordinates.



This output voltage is proportional to the acceleration measured by the sensors and to the power supply voltage. Its sensitivity can be selected using a dip switch, allowing to choose from either 1.5g or 6g. The sensitivity of an accelerometer refers to the greatest amount of acceleration the sensor can measure and accurately represent as an output.

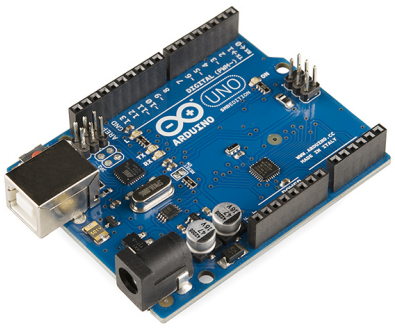
This will be the component used to control the plugin's parameters in real time.

Specification	Value
Voltage	3.3V or 8V
Sensitivity	±1.5g or ±6g
Power Consumption	500μA
Output	Analog
Size	37x26mm

Table 1: MMA7361 spec table.

Which parameters in particular are controlled by the component's outputs will be explained in detail in the second section of this report.
In order to work with the data gathered by its sensors, it's necessary to convert their analog signals to usable digital ones. The Arduino UNO is a microcontroller that can be used to achieve this necessary conversion.
The sensor's information is transmitted to the microcontroller as raw data. Arduino will then convert it to physical values.

1.2 Arduino UNO



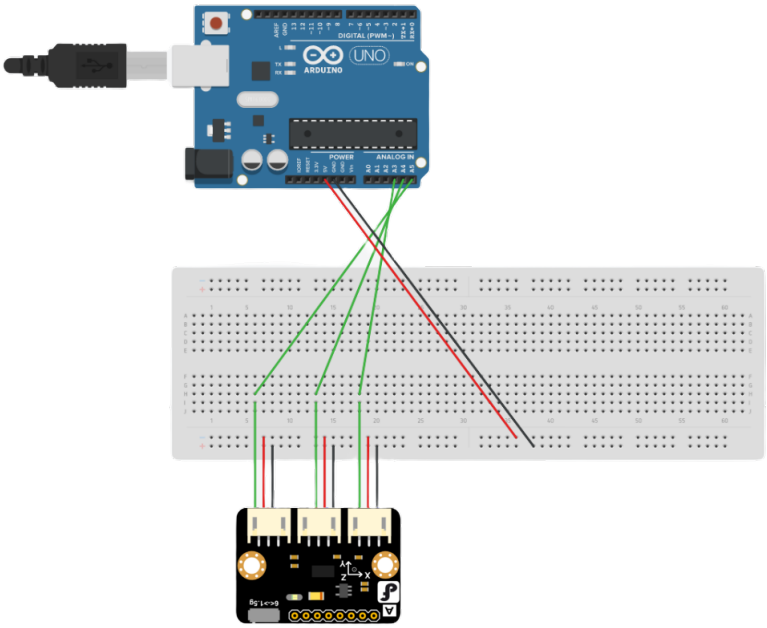
The Arduino UNO is a microcontroller board based on the ATmega328P chip. It provides a number of input sockets: 6 analog inputs, 14 digital input/output pins (6 of which may be used as PWM outputs) and a USB port. It can be connected with a variety of different sensors, such as the accelerometer that's being used in this particular project, and other electrical components that can be programmed using the Arduino programming language.

In order to use the data from the accelerometer, the sensor must be connected to the microcontroller, which will then transmit the signals to the Processing sketch, which will then transmit data to the plugin's host.
This is done through the OSC protocol (Open Sound Control), a content format intended for sharing music performance data.

1.3 Breadboard setup and connections

The MMA7361 used to collect the data, as it can be seen in the following illustration, is connected to the microcontroller using a breadboard.

Three connections are required for each of the three sensors, for a total of nine: one for communication to the Arduino, one to provide power to the accelerometer and another for ground connection.



In the Arduino code, the three analog values from the x, y, and z axes are received and converted to digital. To ensure a more refined signal, the readMMA7361() function employs an averaging technique, calculating the mean value of 50 measurements for each raw value across the three axes. In the getGValues() function, the values are further processed to obtain acceleration data (expressed in g units) using the relevant information outlined in the MMA7361 datasheet. Subsequently, the rotation around the x and y axes, known as Pitch and Roll respectively, is computed utilizing the following formulas:

$$roll = \arctan(\frac{y}{\sqrt{x^2 + z^2}}) \cdot \frac{180}{\pi} \tag{1}$$

$$pitch = \arctan(-1 \cdot \frac{x}{\sqrt{y^2 + z^2}}) \cdot \frac{180}{\pi} \tag{2}$$

Lastly, the calculated Roll and Pitch values are transmitted via the Serial output, ready to be received by Processing.

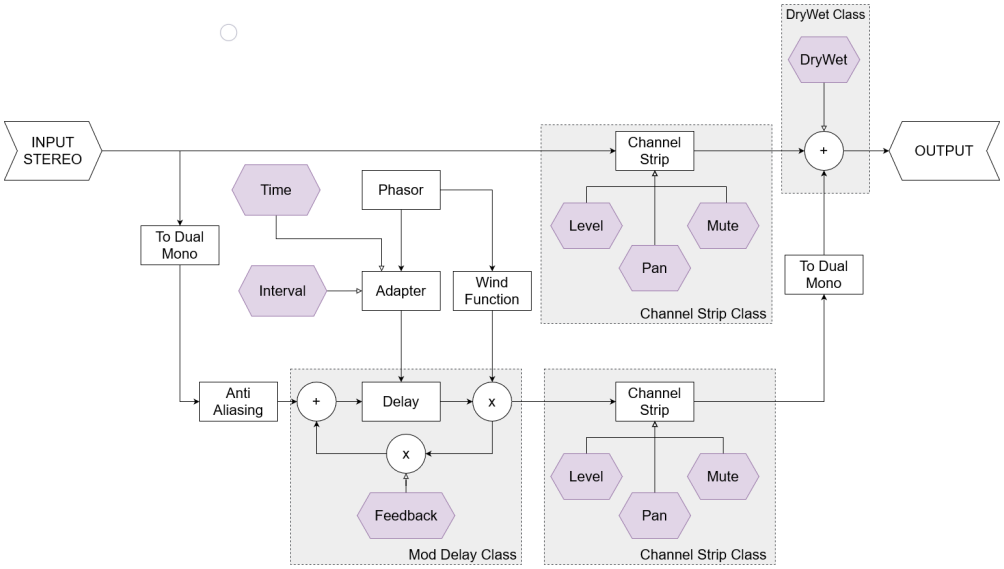
2 Graphical Feedback Unit - GUI and Processing

In the Processing code, a straightforward visualization of the sensor movements is implemented. Additionally, the Roll and Pitch data are transmitted via OSC messages to the JUCE plugin.

Within the setup() function, the same serial port utilized by Arduino for data reception is configured. An OscP5 object is created to handle OSC message communication, with the remote address and port specified for sending the OSC messages.
Subsequently, in the draw() function, a 3D box is generated to visualize the movements. The rotation of the box is determined by the Pitch and Roll data received through the Serial port. Furthermore, these data values are encapsulated within an OSC message and sent to the predefined remote location established during the initialization phase.

3 Computer Music Unit - Kinetic Harmonizer

3.1 JUCE implementation



This block diagram shows the signal flow of the plug-in implemented in JUCE.

To Dual Mono

First of all, the signal is processed by the dual-mono block, which function is to copy the signal in the dual-mono buffer and then process it. The dual mono buffer has two channels so if the signal is mono it is copied in both channels, otherwise, if the signal is stereo, the sum of the two original channels is memorized in each channel of the dual mono buffer.

```
if (totalNumInputChannels == 1) {
  dualmono.copyFrom(0, 0, buffer, 0, 0, numSamples);
  dualmono.copyFrom(1, 0, buffer, 0, 0, numSamples);
  } else {
  dualmono.copyFrom(0, 0, buffer, 0, 0, numSamples);
  dualmono.copyFrom(1, 0, buffer, 1, 0, numSamples);
  dualmono.addFrom(0, 0, dualmono, 1, 0, numSamples);
  dualmono.copyFrom(1, 0, dualmono, 0, 0, numSamples);
  dualmono.applyGain(0.5f);
}
```

The dual mono processing ensures that each channel is being processed with the same precision, making the result more accurate, this is performed in the **PluginProcessor.cpp** file.

Anti Aliasing

In order to avoid the aliasing phenomenon the signal is filtered following the *Nyquist Theorem*, which is processed in the **Filter.h** file. This is a simple filter class that implements a low-pass IIR filter.

```
void processBlock(AudioBuffer<float>& buffer, const int numSamples)
   {
      for (int ch = buffer.getNumChannels(); --ch >= 0;)
         iirFilters.getUnchecked(ch)->processSamples(buffer.
             getWritePointer(ch), numSamples);
   }
```

Mod Delay Class

The following block implements the pitch shifting by the means of delay lines. The delay time will determine the amount of pitch shifting that will occur. Increasing the delay time will lower the pitch of the delayed signal, while decreasing the delay time will raise the pitch of the delayed signal.

Adapter

The delay time is determined by the adapter block, taking into consideration the interval which represents the number of semitones and cents by which the user wants to pitch shift the original signal and a phasor.

```
1    void merge()
2    {
3        auto sum = semitones;
4        n.setTargetValue(sum);
5        shift.setTargetValue(pow(2, sum/12));
6    }
```

The phasor consists of a saw up or saw down LFO depending if we want to increase or decrease the pitch (**Merger.h**, **Oscillator.h**).
The LFO is implemented by the **NaiveOscillator** class, which is a simple oscillator that generates a sawtooth wave.

The adapter first scales the input signal between 0 and 1.

```
1 for (int ch = 0; ch < numCh; ++ch)
2       {
3           FloatVectorOperations::add(bufferData[ch], 1.0, numSamples)
                ;
4           FloatVectorOperations::multiply(bufferData[ch], 0.5f,
                numSamples);
5       }
```

It then applies the modulation amount to the signal using the applyGain() method of the SmoothedValue object.

```
1 modAmount.applyGain(buffer, numSamples);
```

The delay time from the parameter value is then added to the modulated signal. Finally, the signal is scaled to the maximum delay time and the output buffer is filled with the resulting signal.

```
1 for (int ch = 0; ch < numCh; ++ch)
2           FloatVectorOperations::min(bufferData[ch], bufferData[ch],
                MAX_DELAY_TIME, numSamples);
```

Delay

The delay function, implemented in the **Delay.h** file, iterates over the output buffer, processing one sample at a time for each channel. For each sample, it retrieves the current feedback value and computes the delay time for the current modulation channel. It then computes the read and write indices for the delay memory buffer based on the current write index, delay time, and sample rate.
The function then computes the fractional part of the read index using the "integerPart" variable, which is dependent on the *alpha* value.
Finally, the function computes the output sample value by performing linear interpolation between two adjacent samples in the delay memory buffer using the *alpha* value. It then writes the output sample to the output buffer and updates the delay memory buffer with the feedback value.
The *oldSample* array is used to store the previous output sample value for each channel, which is needed for the interpolation computation.

```
1          for (int ch = 0; ch < numChannels; ++ch)
2          {
3              auto dt = delayTimeArray[jmin(ch, numModChannels - 1)][
                   smp];
4
5              auto actualWriteIndex = (writeIndex + smp) % memorySize;
6              auto actualReadIndex = memorySize + actualWriteIndex - (
                   dt * sr);
7
8              auto integerPart = (int)actualReadIndex;
9              auto fractionalPart = actualReadIndex - integerPart;
10
11             auto A = (integerPart + memorySize) % memorySize;
12             auto B = (A + 1) % memorySize;
13
14             const auto alpha = fractionalPart / (2.0f -
                   fractionalPart);
15
16             auto sampleValue = alpha * (delayData[ch][B] - oldSample
                   [ch]) + delayData[ch][A];
17             oldSample[ch] = sampleValue;
18
19             outputData[ch][smp] = sampleValue;
20
21             delayData[ch][actualWriteIndex] += sampleValue * fb;
```

Channel Strip

The signal is then processed by the Channel Strip block which is able to set the level, panning and mute on/off.

DryWet

The signal is finally mixed with the original one, processed by the channel strip class, through a dry/wet that allows users to choose how much of the processed signal is heard in the output.

3.2 Connection with Processing

To enable the reception of Open Sound Control (OSC) messages from an external application, the plugin incorporates the JUCE OSCReceiver class. This class facilitates the establishment of a connection to a specified port, in our case, port 8000, which aligns with the port used by the external application. Subsequently, a listener is implemented in order to wait for incoming OSC messages from the external application.
Once an OSC message is received, the oscMessageReceived() method is triggered, allowing for the extraction and parsing of the message contents. Specifically, the method retrieves the float values representing the *Roll* and *Pitch* parameters, these values are subsequently utilized to dynamically adjust the parameters of the *Dry/Wet* and *Cent Semitones* parameters, within the plugin, respectively.