A Motion-based Controller for Real-time Computer Music with applications for Dance Choreography and Music Composition

The Design, Construction and Programming of a
Wireless accelerometer-based interface system

May 2018

Christopher R. Morgan, Ph.D., cmorgan@collin.edu
Collin College

Objective

To create a low-cost wireless accelerometer-based interface system for Collin College composers, musicians and dancers, that is easy to build, set-up and use with a computer-based software synthesis environment. The system would facilitate collaboration between multiple performing arts departments by providing a technological "bridge" between disciplines.

This paper describes the design, construction and programming of this system. The current version utilizes a three-axis accelerometer connected to a wi-fi capable microcontroller board transmitting wireless Open Sound Control (OSC) messages to a computer running the software synthesis environment Max-MSP. This environment is capable of creating sound in realtime as well as controlling lights, video and electro-mechanical devices. The performer can wear multiple interfaces attached to the hands, feet and torso and the XYZ-axis accelerometer data from each interface can be mapped to synthesis and digital signal processing effects. This paper will also describe a representative synthesis technique used for the initial prototype since the parameter mapping proved especially effective in terms of variety of timbres, response sensitivity as well as expressiveness. Since this controller system generates all of the sounds and effects in real-time, it establishes a unique environment for dance choreography wherein the dancer essentially performs on a hybrid musical instrument in what would traditionally be the role of the composer. As a result, the system is equally effective for choreographed as well as improvised performances.

Context and History

There are three approaches to interactive dance music which have co-existed for over two decades:

- Touch sensitive floors
- Video-based motion tracking systems
- Wearable technologies (both wired and wireless)

Each approach has advantages and disadvantages. For instance, touch-sensitive floors are technologically very simple since they are switches or variable resistance, however they can only respond to being stepped on and therefore don't respond to the dancer's arm, head and torso movements. Video tracking systems can excel at tracking the full body and even recently some hand and finger movements (Kinect v2) but this tracking requires the dancer to stay within a pre-defined area of the camera's field of view. Wearable technologies do not have this limitation because the sensors are attached to different parts of the dancer's body and travel with the performer.

In recent years, the popularity of so-called "maker" (aka DIY) electronics have increased the availability of inexpensive sensors including accelerometers. However, there were several technological obstacles that slowed down widespread development and use. Among the initial obstacles was the fact the wireless protocols available to the microcontrollers were not easily compatible with real-time synthesis languages. For example, there are several wireless communication protocols available for use with microcontrollers and sensors, and getting the microcontrollers to communicate sensor information to each other is relatively easily achieved. However, routing the data into a real-time synthesis environment was difficult and required low-level programming which is typically beyond the reach of performing artists who are typically not programmers.

With the introduction of Open Sound Control (OSC) in 2001, a communication protocol standard began to emerge for wired and later wireless systems. In 2015 OSC software libraries were released for both wired and wireless Arduino-based boards. However, as will be shown, there are technical difficulties that continue to make using OSC challenging for some users.

The second major obstacle for the use of wearable sensors was the sheer size of the wireless transceivers themselves which were often bulky even before factoring in battery packs and antennae, making their use for dancers problematic.

Initial Prototype

The recent availability of smart phones with accelerometers lead to the development of the author's initial prototype of a wearable interface for dancers and computer musicians to control sound synthesis parameters. This initial prototype utilized two smart phones each attached to the back of the dancer's hands so that their movement and choreography created and shaped realtime computer music by transmitting the accelerometer data over wi-fi using the program TouchOSC (hexlar.net). This initial version was premiered in May, 2016 in a work entitled "Fractures". In March 2017 the interface was also used to control DMX-based

Figure 1: Initial Prototype with smartphone

lighting in addition to the sound synthesis and spatialization.

As a "proof of concept" prototype, the smart phones worn on the backs of the wrists worked very well but there are two shortcomings with this approach. First, the smart phones are bulky and while they can be worn on the backs of the hands, they are two big to attach to feet. Secondly, it is cost prohibitive to use anything other than "retired" smart phones since a smart phone/device typically costs several hundred dollars each. In addition, this cost is unnecessary when all that is needed is the accelerometer data. Lastly, setting up TouchOSC to communicate to a computer over a wireless network can be difficult and often it was necessary to create an "ad hoc" network to accomplish this. This increases the technical difficulty of using this form of wearable technology and prevents its widespread use among dancers or even many computer music composers.

System Overview

The interface system is made up of multiple interfaces worn by a musician or dancer on the hands, feet and torso all communicating wirelessly through an access point to a computer running a synthesis environment. The user wears from one to five interfaces but two interfaces worn on the backs of the hands is most common. As mentioned above, the interfaces use Open Sound Control to transmit accelerometer data to the computer. One of the primary objectives of this system is to simplify the process of composing and performing with the interface by automatically establishing OSC connections. For example, at a minimum, wireless OSC messages require access to a network (SSID and Password), a destination IP address and port number. These four settings can pose challenges for the user, especially when they need to be updated due to location/WAN changes. In the case of Arduino-based firmware programming, this requires accessing the interfaces from the IDE, editing code and then uploading the updated firmware containing the above-mentioned values.

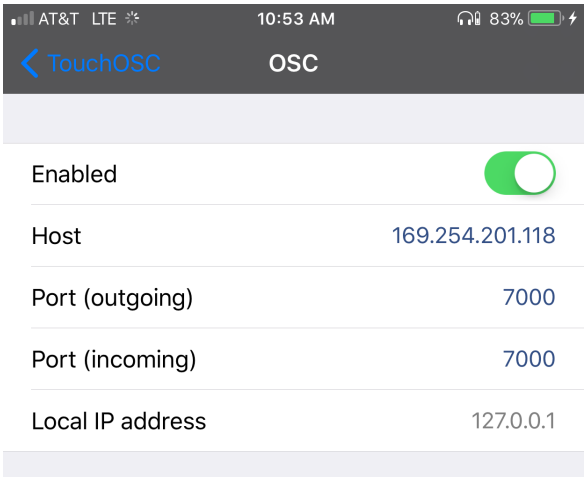

Figure 2: TouchOSC settings page. Note the necessity to enter IP and Port values for each network.

This system eliminates those technical obstacles by using an ESP8266-based wireless access point along with the ESP8266-based wireless accelerometer interfaces. The interfaces and computer all connect to the same access point to exchange the OSC messages. The composer/sound-designer can focus on attaching Max abstractions (such as the one created for this system) to sound synthesis

parameters they wish to control while the performer can focus⇔ on the resulting sounds without having to change settings for multiple interfaces.

Interface ESP8266 and LIS3DH	**OSC** ⇔	**Access Point** ESP8266	**OSC** ⇔	**Computer** Running audio synthesis software

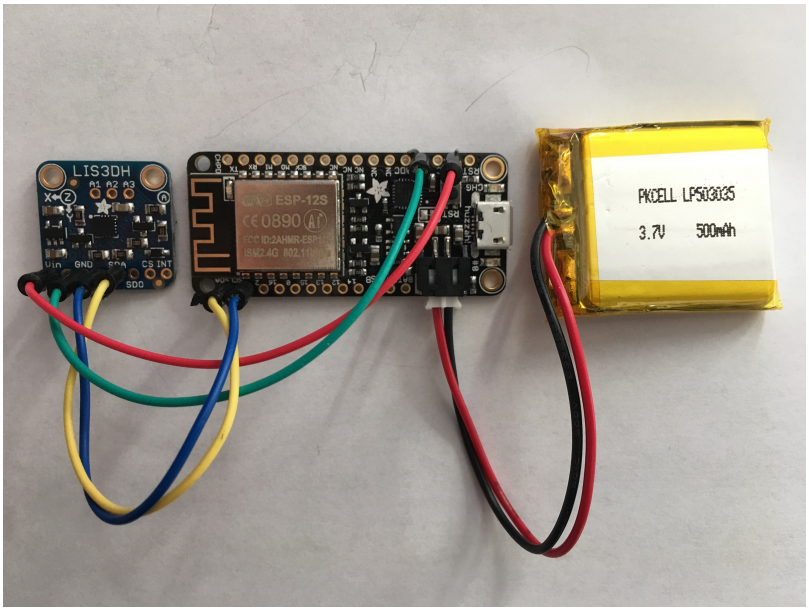

Figure 3: Wiring diagram for LIS3DH, 8266 and battery.

Hardware Components

The following hardware components are used in this interface system:

- ESP8266 Adafruit Huzzah Feather Board
- LIS3DH 3-axis Accelerometer on Adafruit breakout board
- 3.7V 500 mAh Lithium Battery
- Two SPST momentary and one SPST toggle switch(es)
- 3D-printed Enclosure

The **ESP8266** is a low-cost microcontroller with built-in Wi-Fi capability made by Espressif Systems and programmable in the Arduino IDE.

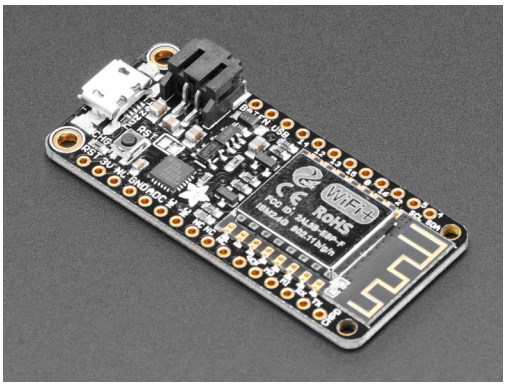

Figure 5: Huzzah 8266 top view. Source: Adafruit.com

Adafruit's Feather Huzzah with ESP8266 includes several add-ons such as battery and micro-USB ports, voltage

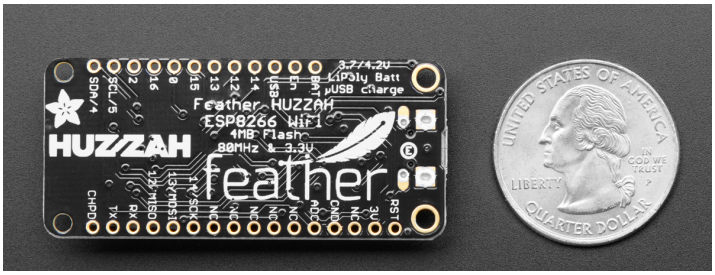

Figure 4: Huzzah 8266 back view. Source: Adafruit.com

regulator and three on-board LEDs. A "pinout" is displayed below showing the available GPIO including the I2C bus connections on pins four and five which are connected to the LIS3DH accelerometer.

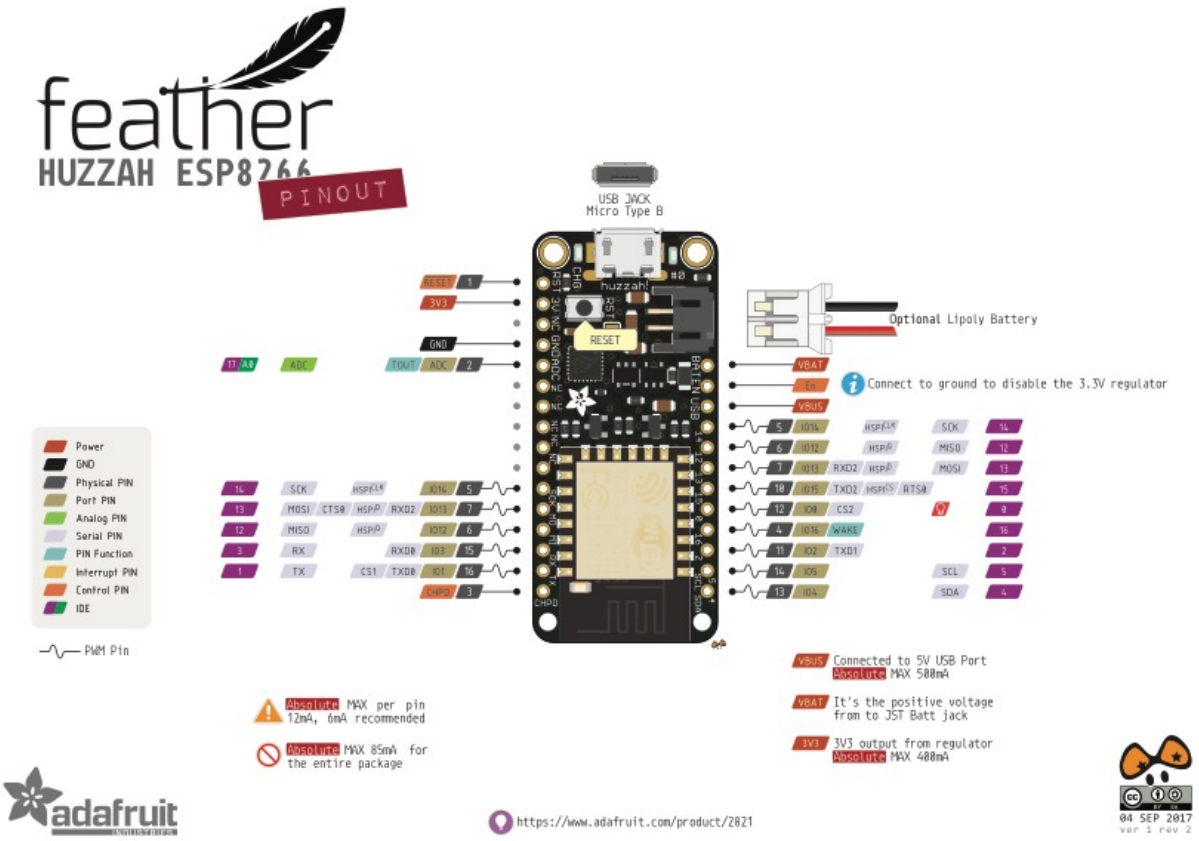

Figure 6: Pinout from Adafruit showing I2C bus on pins four and five (lower right), LEDs and digital IO used for this system.

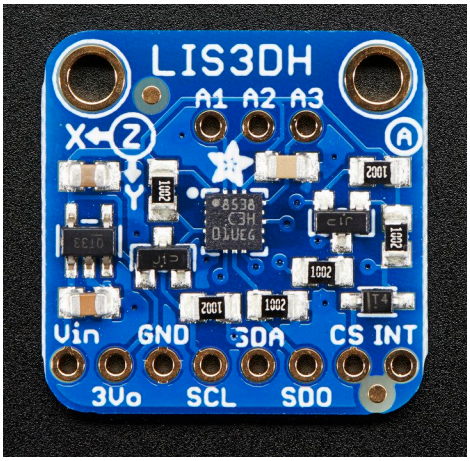

Figure 7: LIS3DH breakout board from Adafruit Industries.

The **LIS3DH** is a low-cost three-axis accelerometer which supports the I2C bus protocol. Adafruit provides the breakout board version used for the project. Both the LIS3DH as well as the 8266 boards run on 3.3 VDC which can be supplied by a battery such as the lithium battery shown above in Figure 3. Note in the figure that power comes to the board via "Vin" (volage in) and "GND" (ground). Accelerometer information is sent via I2C bus on the SCL (clock) and SDA (data) pins.

Wiring the Components

The LIS3DH is wired to the Huzzah ESP8266 by soldering four wires: power, ground, I2C Clock and Data. As shown in Figure 3 above, the main components require little soldering and the additional soldering comes in with added buttons and a power switch.

Two tactile SPST buttons are connected to pins 2 and 12 respectively. Pin 2 is a reset which can be used to re-establish the network connection if needed and also "wake up" from the battery-saving "Deep Sleep" mode. Pin 12 is a digital general-purpose IO and corresponds to the "Aux Button" in the Max *ec05* abstraction. Other digital IO pins are available but only one analog pin. The LIS3DH has three analog pins available. Due to battery life and size considerations, no additional digital or analog pins are used at this time but could be incorporated for future use.

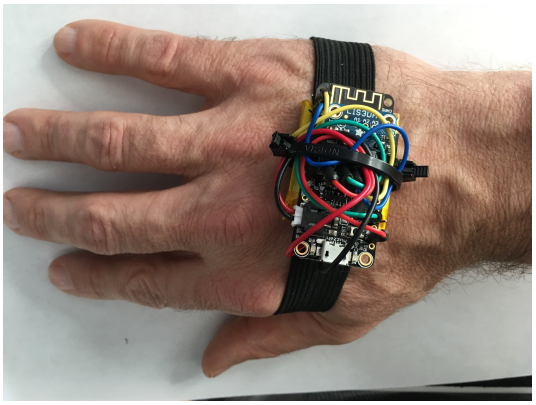

Figure 8: Example of interface without enclosure.

Adafruit's Feather boards include a battery input jack but in order to be able to power down the interface, a SPST switch is placed inline with the ground wire.

Enclosure

An enclosure designed in TinkerCad holds the components and an elastic strap. This environment is an entry-level 3D modeling environment which would make custom user interfaces or new experimental designs accessible for a performing artist or composer. The current size and dimensions are based on the components. Future components could consolidate parts and allow for small dimensions.

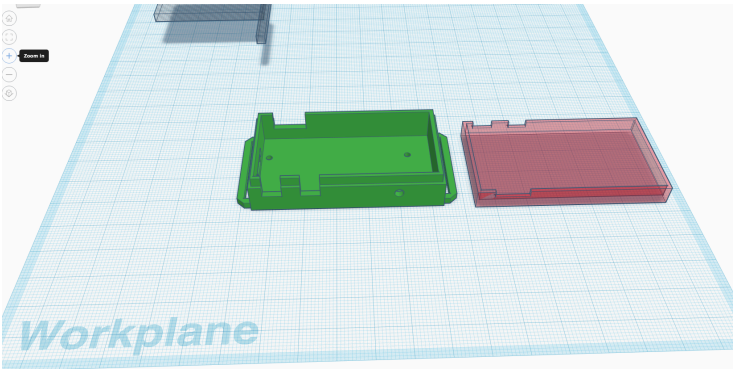

Figure 9: Sample view from TinkerCad showing 3D design for enclosures.

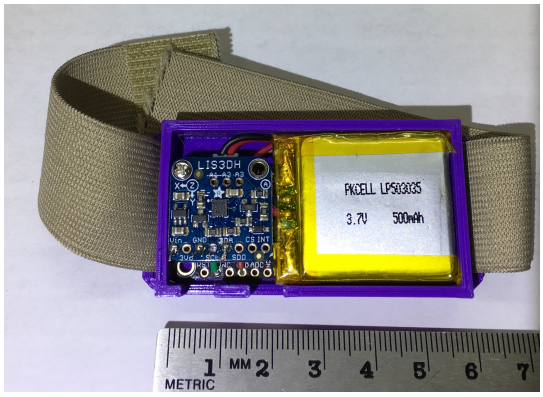

Figure 10: Partially assembled interface showing scale.

At present smaller enclosures are possible using ESP8266 variants but these come with tradeoffs to be added back into the enclosure such as power regulators and a programming connection.

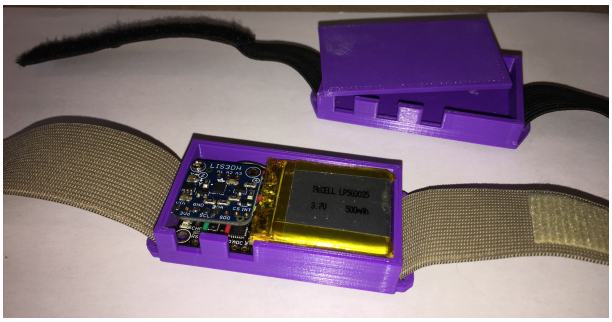

Figure 11: Assembled interface showing scale. The 8266Wi-Fi card is positioned underneath the LIS3dH and battery**.**

Accelerometers as Controllers

An accelerometer-based system is not dependent on the location of the interface but rather on the rate of location change. In contrast, many electronic music controllers are based on changes in location. Examples include sliders and knobs (excluding rotary step encoders) as well as video-tracking systems in which the user navigates a sound space.

Though there are many variations on these location-based navigation systems, at a fundamental level, they share a common design which is giving the user the ability to change parameters along a continuum of possible values or morph between preset values. The accelerometer-based system can be ideal for dance since the dancer can create similar results from different body positions.

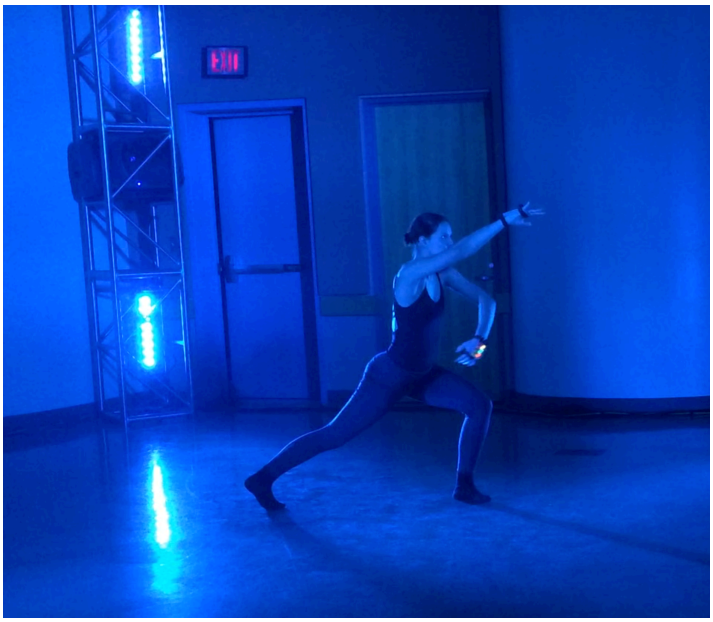

Figure 12: Dancer with prototype interfaces.

In Figure 11, a dancer is wearing two interfaces from an early prototype. The OSC messages are controlling the lights through DMX, as well as creating and spatializing the sound through four loudspeakers.

While the accelerometer's primary data is the rate of change along an axis, there is also a positional data based on orientation such as when the hand is rotated. Each stationary hand position is a unique combination of XYZ values and while these values are small relative to fast movements, they do provide an opportunity for using the interfaces as a more conventional continuous controller.

One significant advantage of using accelerometers is the ability to create "percussion" gestures from short, quick movements. Another consideration is coupling of the XYZ axis making it difficult or impossible to change values on one axis without affecting the values of another. This can be viewed in the same light as many traditional acoustic instruments where parameters such as timbre are coupled with amplitude and frequency.

Mapping the Accelerometer Data to Sound Synthesis Parameters

In Max the composer creates an instance of the ***ec05*** object with arguments specificying interface location and the desired axis. The output of that object is attached to the input of any other Max object meaning the interfaces can control other events such as lighting, sound spatialization, etc. One goal of this project was to explore mappings to synthesis parameters other than "one-to-one" pitch mapping. The synthesis technique developed for the prototype is one based on taking a ramp waveform and dividing it into a number of steps. The technique is very similar to "down sampling" but the approach is different and affords different resulting timbres. Amplitude variation is created by mapping an accelerometer axis to the cutoff frequency of a low pass-resonance filter. Finally, reverberation time and wet/dry mix level are also mapped to the interfaces.

Left Hand	**Destination**	**Right Hand**	**Destination**
X	Base Frequency	X	Number of steps
Y	Reverb Time	Y	Reverb wet/dry mix level
Z	Rate of random high resonance LPF cutoff frequency	Z	Filter wet/dry mix level

Table 1. Sample mapping of accelerometer data to synthesis parameters

Software

Arduino Code

The Arduino IDE (Integrated Development Environment) coding for this system makes use of libraries for the ESP8266, LIS3dH boards as well as the OSC and I2C protocols. The straightforward code for this dance interface simply reads the accelerometer information and sends it wirelessly to the computer listening for the OSC messages. Much of the code deals with networking issues. The complete code is given in the Appendix of this document.

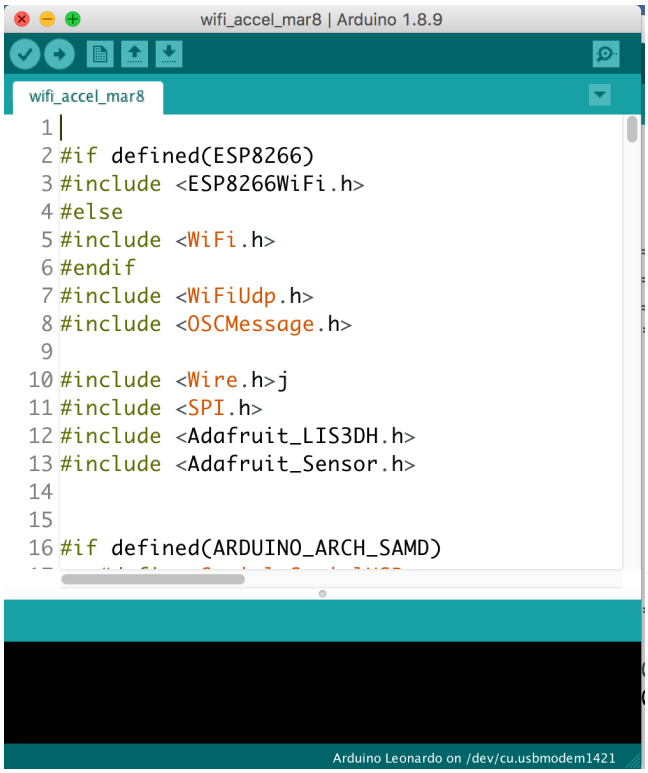

Figure 13: Arduino IDE sample.

The wireless communication requires the SSID and Password. In addition, the OSC messaging requires the IP address and port number of the destination computer. Due to its low cost (each interface approximately USD $30), it is possible to have several interfaces on multiple performers. However, configuring the above-mentioned information on each interface would be cumbersome and is made more difficult for many users considering the Arduino-based boards are only programmable through the Arduino IDE and therefore inaccessible to many end users. Thus, a critical feature aimed at making this interface system easy to use is allow for dynamic or at least simply assignment of all of the necessary information. The port assignment is essentially the location of the accelerometer on the user (e.g. right hand on port 9000, left hand on port 9001, etc.). A hardware switch is possible on the interface to set the port/location however, setting the network SSID, password and the destination computer's IP address would still require plugging in the devices and programming from the Arduino IDE.

The ESP8266 firmware handles logging in to the access point as well as receiving the accelerometer data from the LIS3DH and sending it wirelessly as OSC messages. Other messages include accelerometer tap, double-tap, button presses as well as the interface's dynamically-assigned IP

address so that messages can be sent from the computer to the interface to toggle the two on-board LED's (red and blue).

Max-MSP Interface Code

In order to make a flexible interface, the author coded a patch for mapping and scaling incoming accelerometer data. The user can call on these abstractions and quickly route data to the synthesis parameter of choice. The interface assumes there will be at most five interfaces each with XYZ axis data.

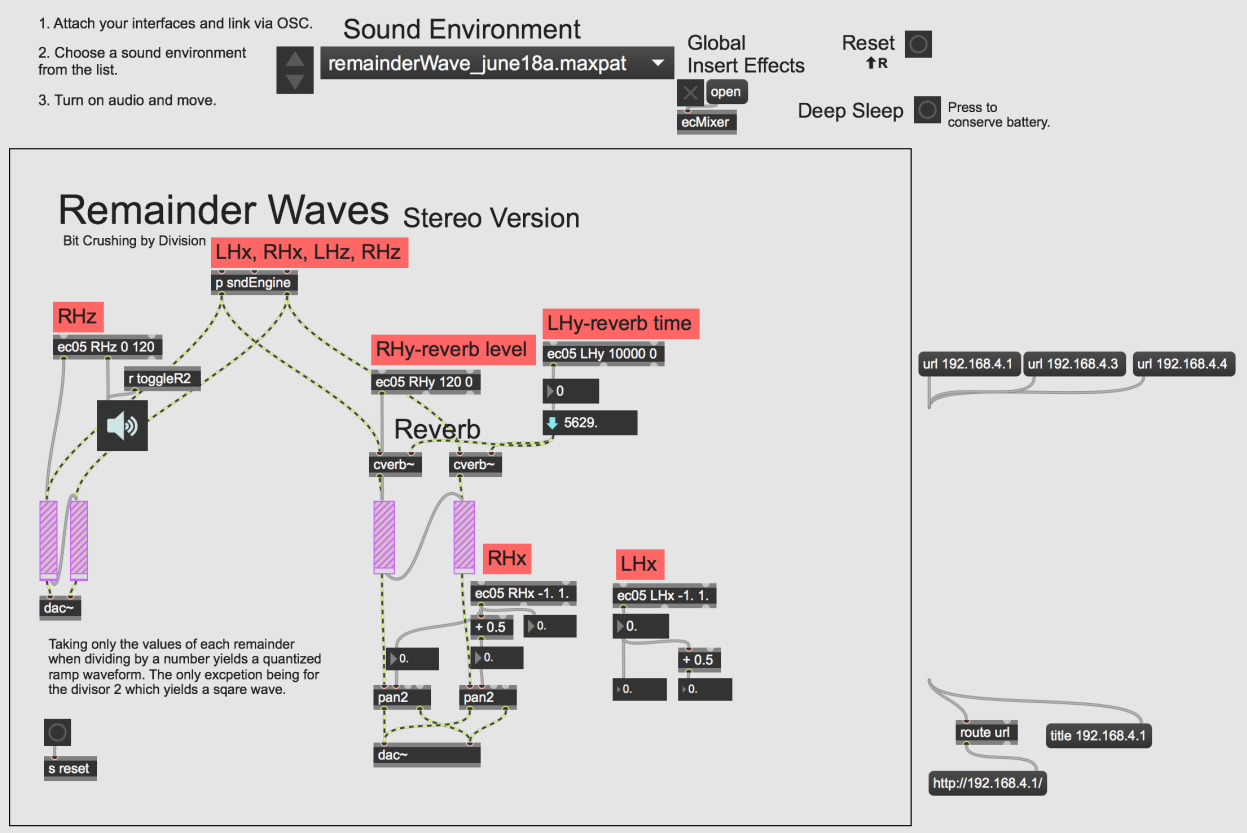

Figure 14: The sound environments portal showing "Remainder Waves" as one of several Sound engines.

For example, the software synthesis engine for the initial version of the interface uses a technique similar to bit-crushing resulting in a "noise music" that transitions from pitched to non-pitched sound allowing for the creation of percussive as well as ambient textures. This flexibility proved very beneficial since it allowed a single engine to be used for the entire performance without the need to switch to a different synthesis engine.

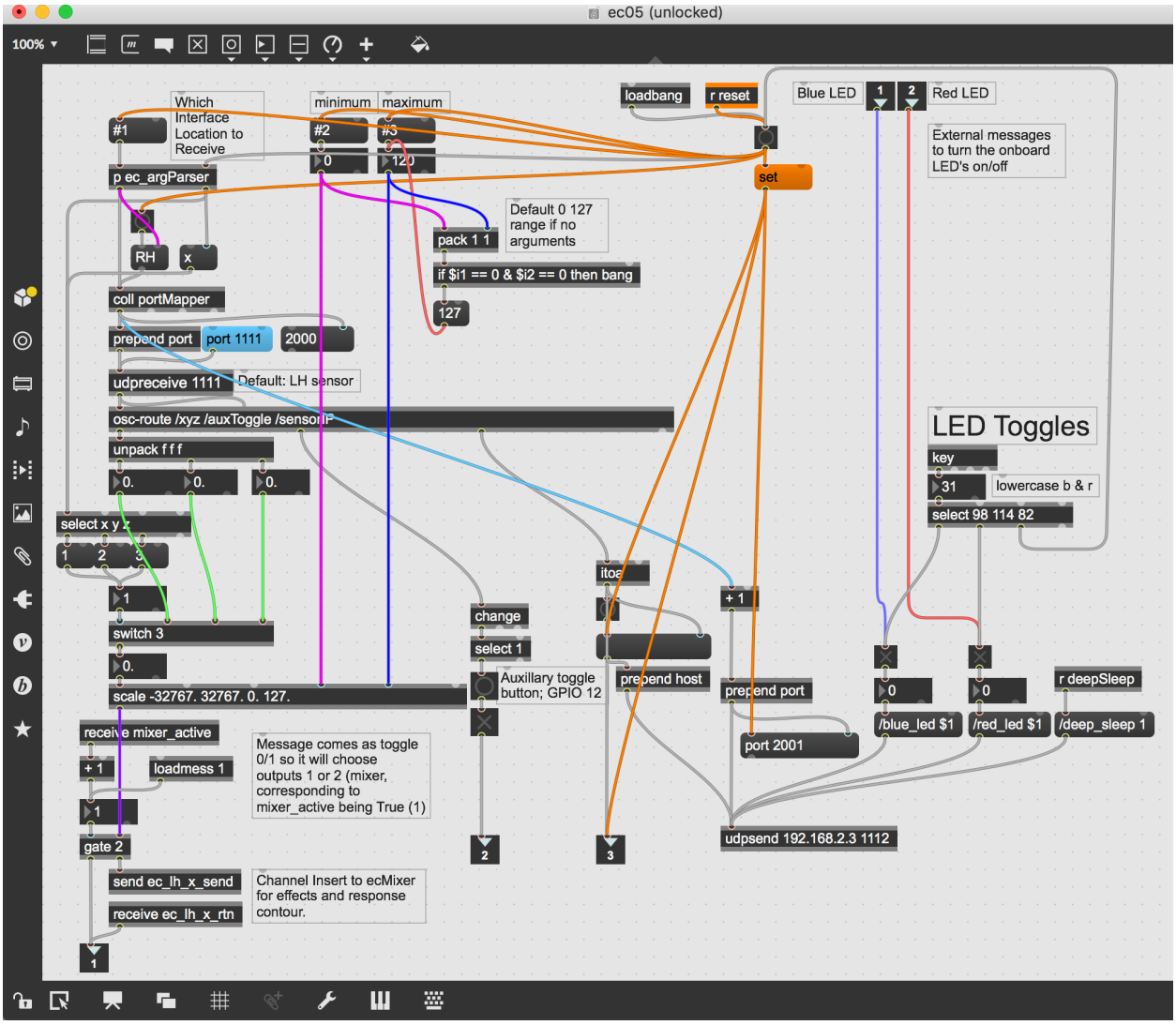

Figure 15: The EC05 abstraction showing connectivity and controls.

Max-MSP Sound Synthesis Code

The synthesis technique utilized in the initial version of the interface contributed to the initial reception and "proof of concept" of the interface due, in part, to the variety of timbres possible. Particularly noteworthy, is the lack of mapping pitch to any of the accelerometer axis. The software engine can be divided into three sections:

- Sound generation
- Filtering
- Reverb

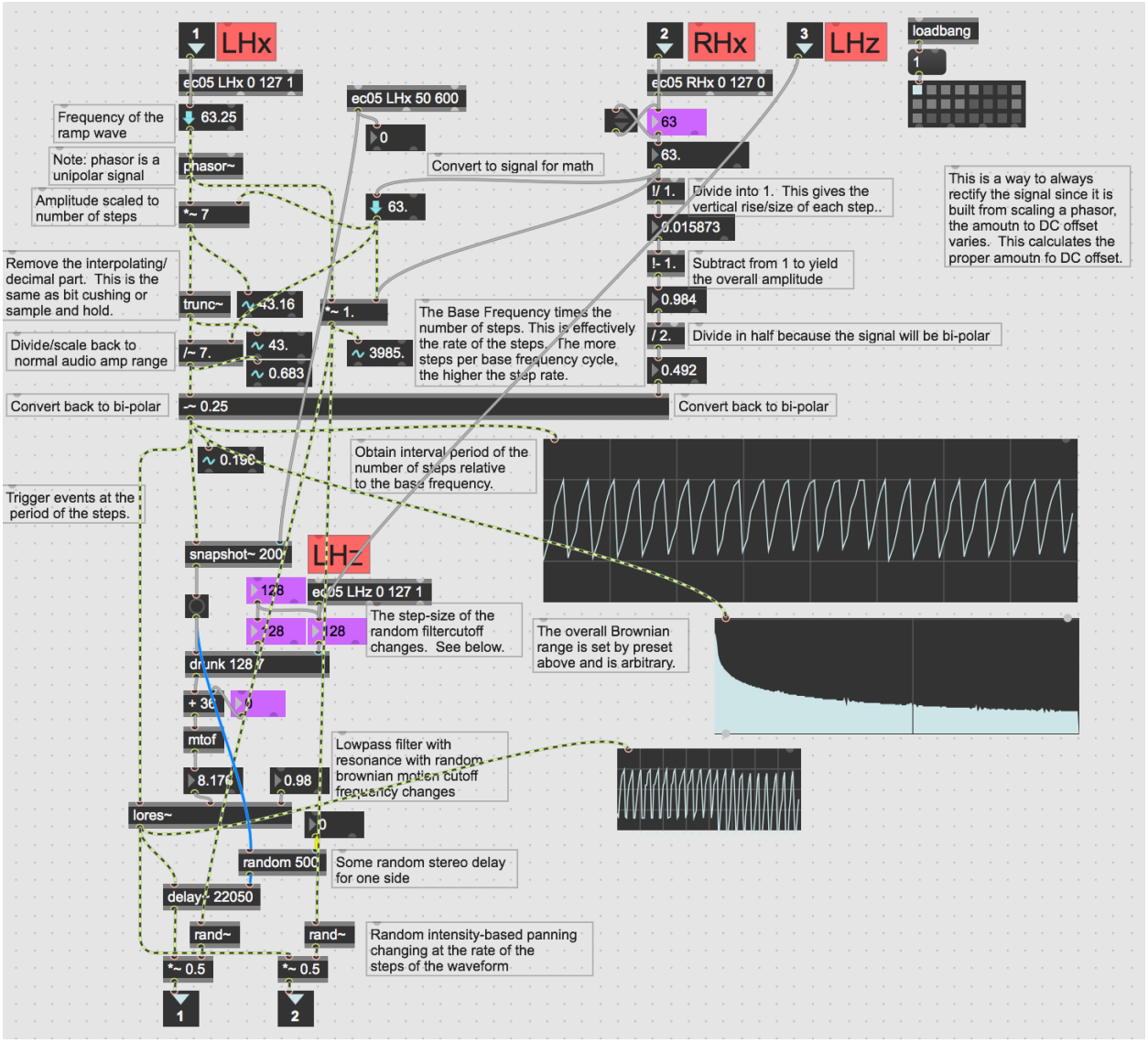

Figure 16: A detailed view of the Remainder Waves synth. For an example of mapping, the red "LHz" is the left-hand z-axis.

As noted earlier, each of these sections is controlled by two accelerometer axis each. The sound generation uses a technique similar to bit-crushing but not actually changing the bit depth: a ramp wave is converted into a stair-step wave by using what amounts to a sample-and-hold function. The results of the resampled ramp wave are then passed through a low pass filter with a random cutoff whose rate is determined by the number of steps thereby coupling those two parameters. The final section is the reverb who range of values go from heavy to light reverberation.

Additional Max Abstractions and Patches

The abstraction at the heart of the system is ***ec05*** shown connected in Figure 15. It receives the XYZ accelerometer data, IP address and button events from each interface. It can also send messages such as toggling two of the onboard LEDs.

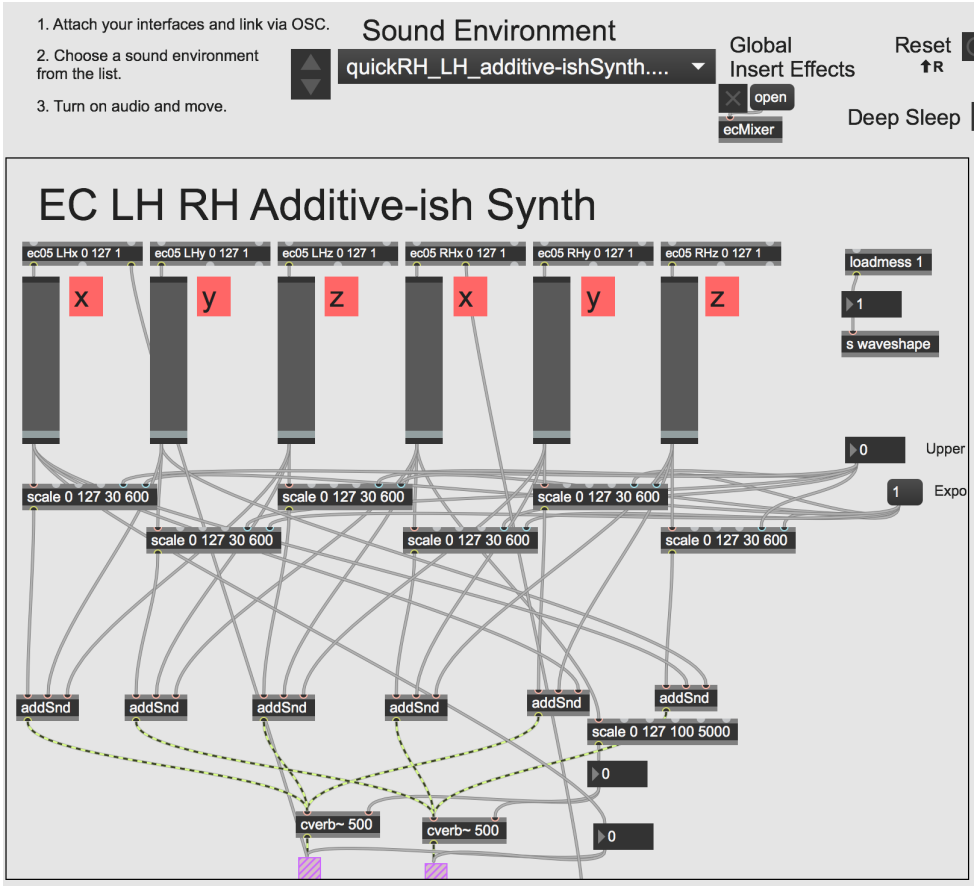

Figure 17: Another "sound environment" this time using additive synthesis as the sound engine.

Abstraction arguments are the interface location (e.g., Left Hand = LH) and the desired re-scaled output range. To get the data from the left hand X axis, one would enter "LHx" as an argument. The next two arguments are the minimum and maximum output range which are rescaled from the accelerometer's raw output range of -16384 to +16383. Without these arguments the default range is rescaled 0-127.

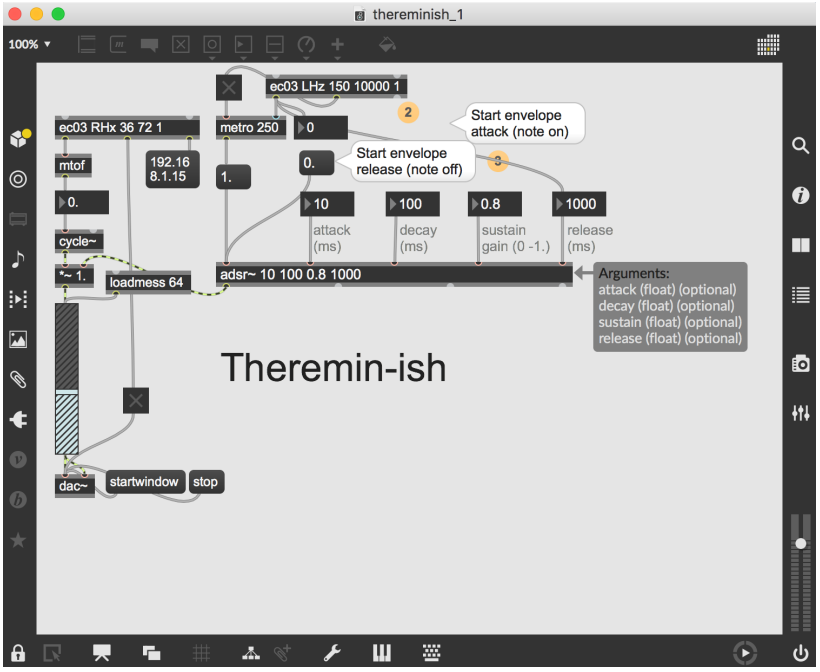

Figure 18: A "Theremin" sound environment tracking x-axis pitch and z-axis amplitude.

Much of the programming involves making the *ec05* abstraction generic so the user need only specify the interface location on the body. A table (Max *coll* object) correlates the location with the OSC port number. This "hard wiring" allows the interface to broadcast its IP address to confirm connection and allow the computer to send messages back to the interface. For the composer, the *ec05* abstraction allows for quickly attaching an interface's XYZ accelerometer and Aux button output as a control input in a Max patch.

Figure 14 is a high-level patch that functions as a portal for multiple sound engines which are loaded into the Max *bpatcher* object. Users can quickly switch between environments/sound engines. This patch can also handle global events such as a button to send the "Deep Sleep" command to all of the ESP8266 interfaces simultaneously putting them into very low power mode. Lastly, the *ec05* abstraction also allows for an "insert effect" break in the accelerometer data signal flow.

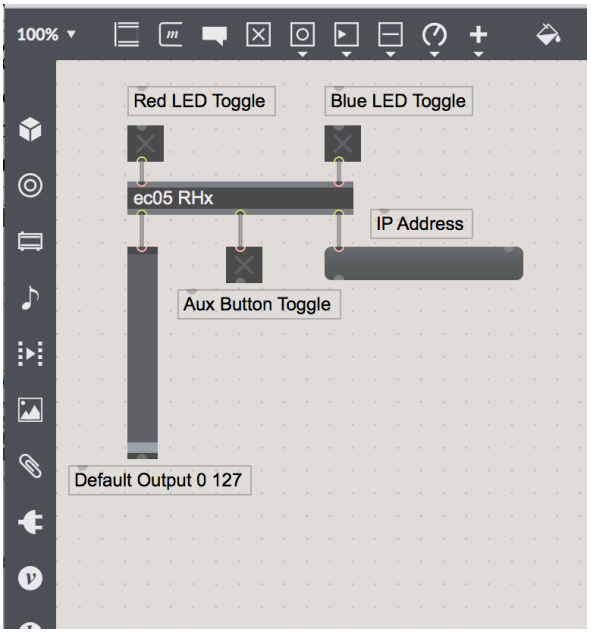

Figure 19: EC05 abstraction demonstrating the possible input and outputs for the Right hand X-axis (RHx).

The *ecMixer* abstraction is similar to a plugin/insert effect module of a DAW. The raw data from the accelerometers can be processed with a shaping function, echo effects with the possibility of time dilation, a looper and others in future

development. The insert strips for each interface axis and the inserts themselves are created dynamically with Javascript.

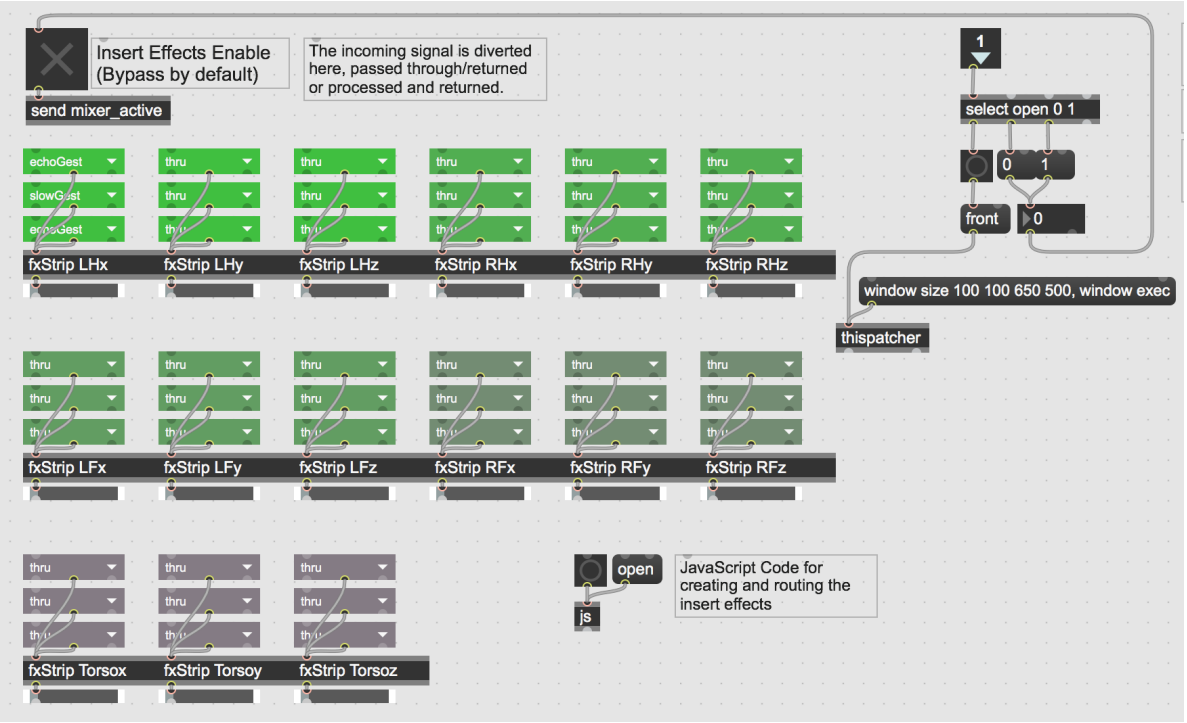

Figure 20: The mixer showing a complete matrix of five interfaces with possible insert effects.

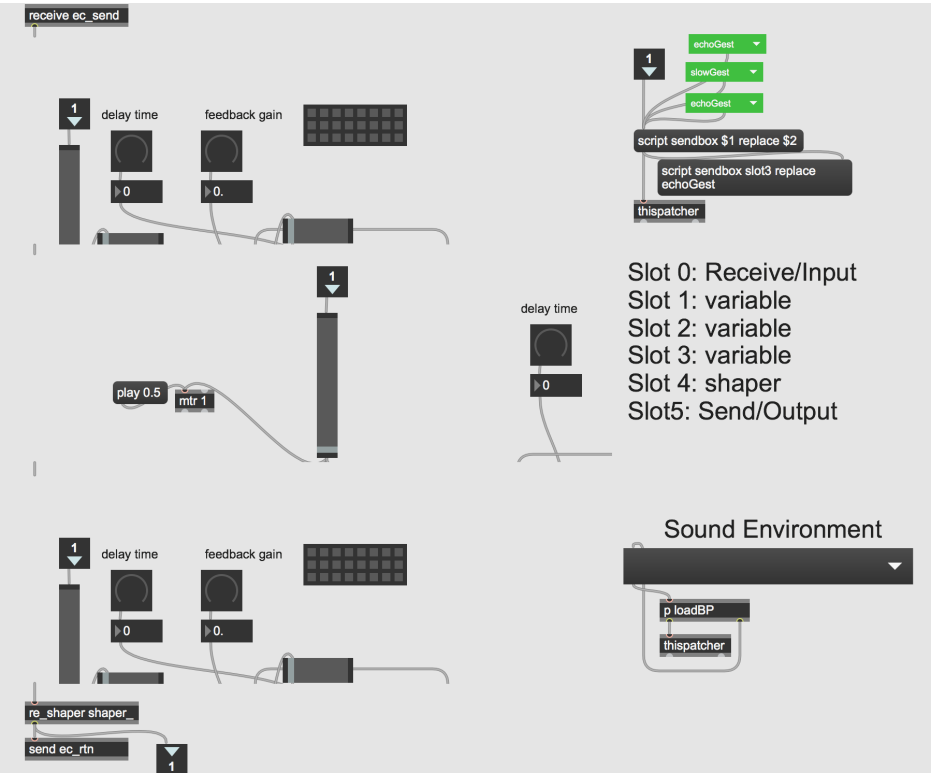

Figure 21: An example of a insert effect with dynamic object creation.

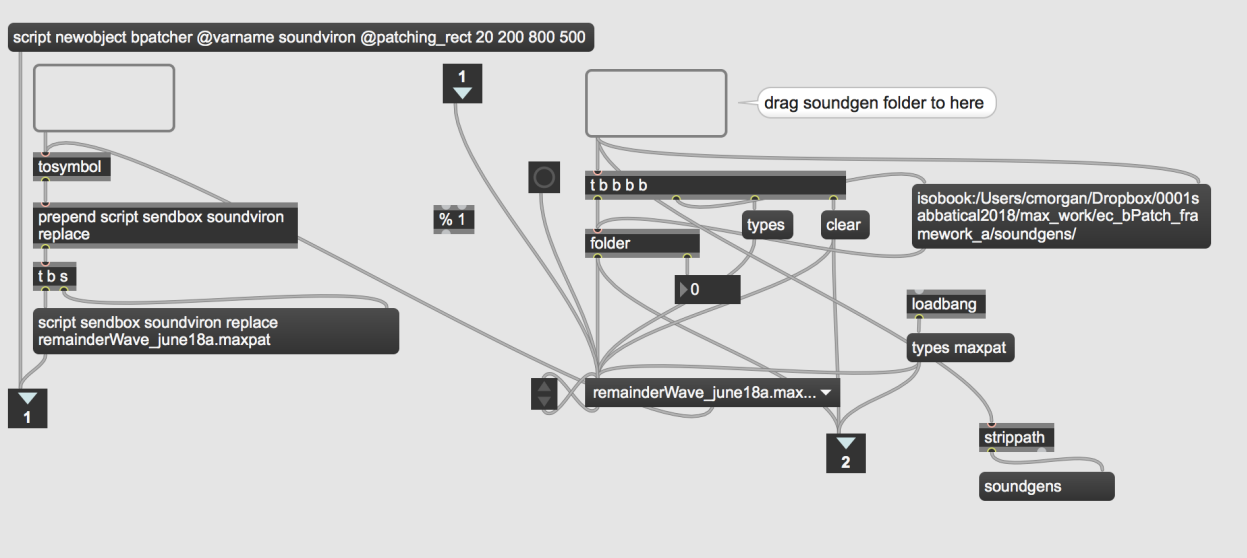

Figure 22: Dynamic menu loading with all of the possible sound environments stored in the application folder.

Future Work

Future work will focus on continuing to reduce the size of the interface by using smaller/simpler versions of the ESP8266, using flexible 3D printing filaments that will conform to the hand and foot, increasing battery life and lastly, using the Raspberry Pi to serve as the access point as well as a Pure Data synthesis environment.

Appendix

Appendix A: Arduino Interface Code:

```
#if defined(ESP8266)
#include <ESP8266WiFi.h>
#else
#include <WiFi.h>
#endif
#include <WiFiUdp.h>
#include <OSCMessage.h>

#include <Wire.h>j
#include <SPI.h>
#include <Adafruit_LIS3DH.h>
#include <Adafruit_Sensor.h>

#define CLICKTHRESHHOLD 90

#if defined(ARDUINO_ARCH_SAMD)
   #define Serial SerialUSB
#endif

IPAddress gateway;

//char ssid[] = "XXXXXX";                 // your network SSID (name)
//char pass[] = "XXXXXXX";     // your network password

//Code added from OSC Receive Example

OSCErrorCode error;
unsigned int redLedState = HIGH;              // LOW means led is *on*
unsigned int blueLedState = HIGH;              // LOW means led is *on*

/* MorganNotes: Buttons and the Adafruit Huzzah Feather ESP8266 Built-in LEDs
Orange LED is for battery charging and is not accessible via pins
Red LED is GPIO 0
Blue LED is GPIO 2
Reset Button is GPIO 0
Audio Toggle is GPIO 12
*/

const int RED_LED = 0;
const int BLUE_LED = 2;
const int AuxToggleButton = 12;
int old_AuxToggleButton_state = 1;
int new_AuxToggleButton_state = 0;
int sleepState = 0;


//Code added from OSC Receive Example

WiFiUDP Udp;            // A UDP instance to let us send and receive packets
over UDP
//*****************************The following IP comes from the computer via
DHCP ****************************************
//*****************************Attempting to use a Manually entered fixed IP
of 192.168.2.2 *******************************
```

```
const IPAddress outIp(192,168,4,4);           // remote IP of your computer:
This is subject to change based on Router DHCP: See above.
//OSC/UDP Port Mapping
//LH, 1111;
//RH, 2000;
//LF, 3000;
//RF, 4000;
//Torso, 5000;
const unsigned int outPort = 2000;            // remote port to receive OSC;
see above legend
const unsigned int localPort = outPort+1;         // local port to listen for
OSC packets (actually not used when only sending)
Adafruit_LIS3DH lis = Adafruit_LIS3DH();


void setup() {

//Setting up initial LED states and Button/Pin modes
  pinMode(RED_LED, OUTPUT);
  pinMode(BLUE_LED, OUTPUT);
  digitalWrite(RED_LED, redLedState);    // set initial Red LED state
  digitalWrite(BLUE_LED, blueLedState);    // set initial Blue LED state
  pinMode(AuxToggleButton, INPUT_PULLUP);



#ifndef ESP8266
  while (!Serial);     // will pause Zero, Leonardo, etc until serial console
opens
#endif
//******************************Begin        LIS3DH        Startup        and
Setup*******************************
  Serial.begin(9600);
  Serial.println("LIS3DH test!");
  if (! lis.begin(0x18)) {     // change this to 0x19 for alternative i2c
address
    Serial.println("Couldnt start");
    while (1);
  }
  Serial.println("LIS3DH found!");
  lis.setRange(LIS3DH_RANGE_2_G);   // 2, 4, 8 or 16 G!
  Serial.print("Range = "); Serial.print(2 << lis.getRange());
  Serial.println("G");
//******************************End         LIS3DH         Startup         and
Setup*******************************

//     ******************************     Connect     to     WiFi     network
******************************
    Serial.println();
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, pass);
    while (WiFi.status() != WL_CONNECTED) {
        delay(100);
        Serial.print(".");
    }
    Serial.println("");
```

```
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
//  ******************************  Max-OSC  will  ask  for  this  local  IP
address ********************
    Serial.println(WiFi.localIP());
// ******************************
    Serial.println("Starting UDP");
// ****************************** Max-OSC will ask for this local port to
determine location ********************
    Udp.begin(localPort);
    Serial.print("Local port: ");
#ifdef ESP32
    Serial.println(localPort);
#else
    Serial.println(Udp.localPort());
#endif
// ****************************** click set up  **********************
lis.setClick(2, CLICKTHRESHHOLD);

//String tester = "hello";
//char charTester = '5';
//int charTester2 = int(charTester);
//
//OSCMessage msgOUT("/sensorIP");
//    msgOUT.add(charTester2);
//    Udp.beginPacket(outIp, outPort);
//    msgOUT.send(Udp);
//    Udp.endPacket();
//    msgOUT.empty();

  gateway = WiFi.gatewayIP();
  Serial.print("GATEWAY: ");
  Serial.println(gateway);


}        //    ****************************************         END    SETUP
**************************************************
//
*********************************************************************************
************************************

//****************************** Start Code added from OSC Receive Example
void red_led_reader(OSCMessage &msgIN) {
  redLedState = msgIN.getInt(0);
  digitalWrite(RED_LED, !redLedState);
//  Serial.print("/redLed: ");
//  Serial.println(redLedState);
}
void blue_led_reader(OSCMessage &msgIN) {
  blueLedState = msgIN.getInt(0);
  digitalWrite(BLUE_LED, !blueLedState);
//  Serial.print("/blueLed: ");
//  Serial.println(blueLedState);
} // ****************************** End Code added from OSC Receive Example

void put_to_sleep(OSCMessage &msgIN) {
  sleepState = msgIN.getInt(0);
  if (sleepState == 1) {
```

```
    sleepState = 0;
    ESP.deepSleep(999);
  }
  }
//  Serial.print("/blueLed: ");
//  Serial.println(blueLedState);


//
*****************************************************************************
**************************************
//          **************************************                    LOOP
*********************************************************************
//
*****************************************************************************
**************************************
void loop() {

  uint8_t click = lis.getClick();
//  if (click == 0) return;
//  if (! (click & 0x30)) return;
//  Serial.print("Click detected (0x");
//  Serial.print(click, HEX);
//  Serial.print("): ");
  if (click & 0x10) {
    OSCMessage msgClickOUT("/click");
    msgClickOUT.add(click);
    Udp.beginPacket(outIp, outPort);
    msgClickOUT.send(Udp);
    Udp.endPacket();
    msgClickOUT.empty();
    Serial.print(" single click");
  }
  if (click & 0x20) {
    Serial.print(" double click");
    Serial.println();
  }

  OSCMessage msg2OUT("/sensorIP");
String tempStr = WiFi.localIP().toString();
int i;
for (i = 0; i < sizeof(tempStr); i++) {
  char oneChar =  tempStr[i];
  int intChar = int(oneChar);
    msg2OUT.add(intChar);
 //   Serial.println(tempStr[i]);
  }
    Udp.beginPacket(outIp, outPort);
    msg2OUT.send(Udp);
    Udp.endPacket();
    msg2OUT.empty();

  new_AuxToggleButton_state = digitalRead(AuxToggleButton);
    OSCMessage msgOUT3("/auxToggle"); //Changed msg to msgOUT to distinguish
in vs. out.
    msgOUT3.add(new_AuxToggleButton_state);
    Udp.beginPacket(outIp, outPort);
    msgOUT3.send(Udp);
```

```
    Udp.endPacket();
    msgOUT3.empty();
//    Serial.println("Auxillary Toggle Button State is   ");
//    Serial.print(new_AuxToggleButton_state);
//    Serial.println();


  lis.read();      // get X Y and Z data at once
  // Then print out the raw data

//  Serial.print("X:  "); Serial.print(lis.x);
//  Serial.print("  \tY:  "); Serial.print(lis.y);
//  Serial.print("  \tZ:  "); Serial.print(lis.z);

// sensors_event_t event;
//  lis.getEvent(&event);


    OSCMessage msgOUT("/xyz"); //Changed msg to msgOUT to distinguish in vs.
out.
 //   msgOUT.add("hello, osc.");
    msgOUT.add(lis.x);
    msgOUT.add(lis.y);
    msgOUT.add(lis.z);
//    msgOUT.add(XLoc);
    Udp.beginPacket(outIp, outPort);
    msgOUT.send(Udp);
    Udp.endPacket();
    msgOUT.empty();
    delay(50);


//*******************************this is the code I am adding Feb. 19.
// the incoming message from Max over OSC/UDP will cause Deep Sleep

OSCMessage msgIN;
  int size = Udp.parsePacket();
//*********start for future use*********
// WiFiUDP.remoteIP();
//**********end for future use*********
  if (size > 0) {
    while (size--) {
      msgIN.fill(Udp.read());
    }
    if (!msgIN.hasError()) {
      msgIN.dispatch("/red_led", red_led_reader);
      msgIN.dispatch("/blue_led", blue_led_reader);
      msgIN.dispatch("/deep_sleep", put_to_sleep);
    } else {
      error = msgIN.getError();
      Serial.print("error: ");
      Serial.println(error);
    }
  }
}
```

Appendix B: ESP8266 WI-FI Access Point Arduino Code

```
/* Create a WiFi access point and provide a web server on it. */

#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>

/* Set these to your desired credentials. */
const char *ssid = "XXXXXX";
const char *password = "XXXXXXX";

ESP8266WebServer server(80);

/* Test message.  Go to http://192.168.4.1 in a web browser
 * connected to this access point to see it.
 */
void handleRoot() {
	server.send(200, "text/html", "<h1>You are connected</h1>");
}

void setup() {
	delay(1000);
	Serial.begin(115200);
	Serial.println();
	Serial.print("Configuring access point...");
	/* You can remove the password parameter if you want the AP to be open. */
	WiFi.softAP(ssid, password);

	IPAddress myIP = WiFi.softAPIP();
	Serial.print("AP IP address: ");
	Serial.println(myIP);
	server.on("/", handleRoot);
	server.begin();
	Serial.println("HTTP server started");
}

void loop() {
	server.handleClient();
}
```