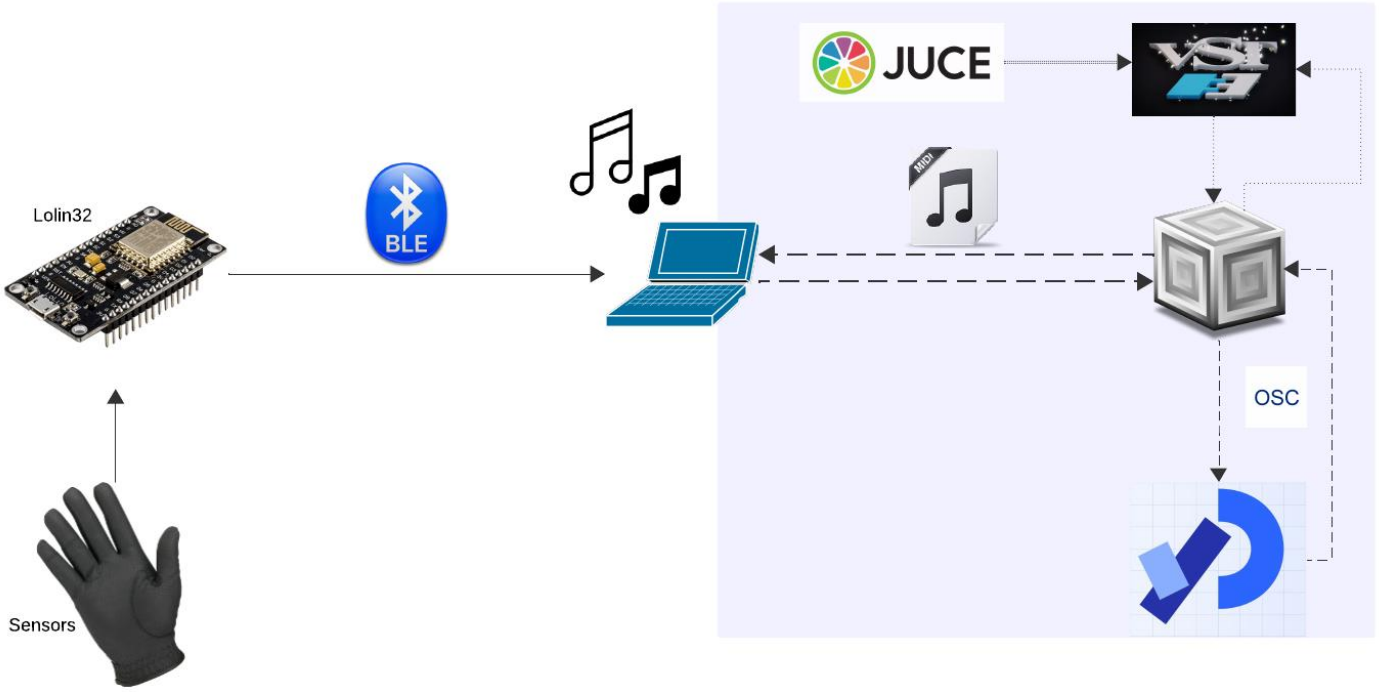
# REPORT CMLS

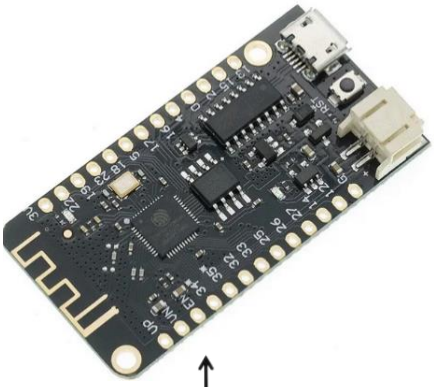
ABSTRACT:
The objective of this project was implementing a computer music system with interaction design principles. The constrains were to use at least these three following blocks:

- Interaction system unit: in our particular case ARDUINO;
- Computer music unit: using SuperCollider and JUCE;
- Graphical feedback unit: using Processing.

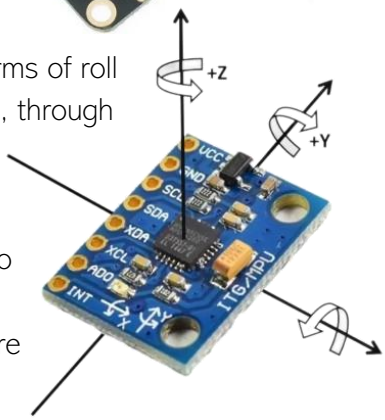



Our project consists of a glove with three sensors which detect the hand's movements and, by using a Bluetooth communication, the data is collected by our interaction system unit. The system unit elaborates the data received, to create a MIDI message suitable for SuperCollider. SuperCollider is used to generate different kind of waves that can be chosen by the GUI. The waves are then elaborated by the VST, created using JUCE, to obtain a sound similar to human voice. The graphic interface, other than choosing the type of wave, gives the possibility to change some parameters of the sound synthesis. This interface communicates through OSC messages with SuperCollider.

ARDUINO:



The Arduino board we used is the “lolin32”, which is connected through the I2C communication to the sensor MPU6050. This sensor consists of an accelerometer and a gyroscope that can estimate the hand’s angular rotation and, through the Bluetooth connection, the microcontroller can send a MIDI control message to SuperCollider. Our code utilizes the MPU6050's accelerometer to measure the device's acceleration along three spatial axes and its gyroscope to measure its angular velocity around the x




and y axes. These data are used to calculate the device's orientation in space, in terms of roll and pitch angles. This orientation information is then translated into MIDI messages, through a conversion from angles (-90,90) to integers from 0 to 127. Essentially, the device can control some music qualities based on its rotation in space.

In addition, we have a resistor sensor that estimates the hand’s flexion, it is linked to a pin located on the index finger and, by touching it, it defines the MIDI note we want to send. The glove contains also three buttons on the back of the hand that are used for choosing the octave we want the human voice to reproduce. This type of message is still a MIDI control message.

In order to have a Bluetooth communication, the BLE midi server must be initialized and then a connection must be established. This code snippet configures a Bluetooth MIDI server named "MIMU gloves". It then waits for a device to connect to the Bluetooth MIDI server. During the waiting period, it prints "Waiting for connections" on the serial port, and a dot is printed every second until a connection is established at which occurrence "Bluetooth connected." is printed on the serial port. The main loop of the program continues to check if the Bluetooth MIDI connection is active, repeating the waiting process if the connection is lost.

```
  BLEMidiServer.begin("MI.MU ");
  Serial.print("Waiting for connections");
  while(!BLEMidiServer.isConnected()) {
    Serial.print(".");
    delay(1000);
  }
  Serial.println("\n Bluetooth connected.");
}

void loop(){
  if (!BLEMidiServer.isConnected()) {
    Serial.print("Waiting for connections");
    while(!BLEMidiServer.isConnected()) {
      Serial.print(".");
      delay(1000);
    }
```

```
void getMidiNote() {
  curr_n = !curr_n;
  int out;
  float_t val = analogRead(FLEX);
  if (val < 1800) val = 1800;
  else if (val > 3400) val = 3400;
  out = int((val - 1800)/(1600/12));
  // Serial.printf("val = %f, out = %i\n", val, out);
  midi_note[curr_n] =  out + octave;
}

/* ---- TOUCH FUNCTIONS ---- */

void touchButton(bool *bt, uint8_t pin, uint8_t tresh, void (*when_pressed)(uint8_t),
                 void (*still_on)(uint8_t), void (*when_released)(uint8_t)) {
  if(touchRead(pin) < tresh && !*bt) {
    (*when_pressed)(pin);
    *bt = true;
  } else if (touchRead(pin) < tresh && *bt) {
    (*still_on)(pin);
  }
  else if (touchRead(pin) > tresh && *bt) {
    (*when_released)(pin);
    *bt = false;
  }
}
```

The first part, the `getMidiNote()` function, handles reading the resistance sensor. This sensor provides an analog value reflecting the flexion of the hand, in floating point values from 0 to 4096. The code adapts the range to values from 0 to 11, as the available notes in an octave. The value we just calculated is the added to an octave value being the starting C of the current octave (C3, C4 or C5).

The second part of the code concerns the `touchButton()` function. When the touch pin is pressed it get the current note given by the resistor. This function is also used for the octave selector buttons, averaging the input value to obtain a better output value. Thin implementation while being a little bit more complex avoids repetition of code which is crucial in embedded environments.

In this part instead we are sending MIDI notes, checking if there are new notes and stopping the notes if the touch button is released.

This set of functions is used as arguments of the touchButton function for the trigger, comprised of a jumper cable attached to an aluminium piece located on the thumb of the glove. Sending a new MIDI note at a first trigger, sending a new note, and stopping the previous one, in case of a change in the reading of the resistance and in the end stopping the last note when the trigger is released.

```
void sendNote(uint8_t arg) {
  Serial.printf("in sendNote\n");
  getMidiNote();
  BLEMidiServer.noteOn(1, octave + midi_note[curr_n], 127);
}

void checkNote(uint8_t arg) {
  Serial.printf("in CheckNote\n");
  getMidiNote();
  if(midi_note[curr_n] != midi_note[!curr_n]){
    BLEMidiServer.noteOff(1, midi_note[!curr_n], 127);
    BLEMidiServer.noteOn(1, midi_note[curr_n], 127);
  }
}

void stopNote(uint8_t arg) {
  Serial.printf("in StopNote\n");
  BLEMidiServer.noteOff(1, midi_note[curr_n], 127);
}
```

The implementation of the octave selector mimics the one just described but having 2 identical functions for adding the new value to a container variable which will be averaged to obtain the correct octave. The 3 obtained values depend on the difference conductivities measured in 3 spots of a single strip of aluminium covered in 3 different materials.

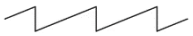
```
  getMidiCc();
  //Serial.printf("roll: %i --- pitch: %i\n", midi_cc[curr*2 + 0], midi
  touchButton(&strip, T_BUTS, thresh1, evaluate, evaluate, setOctave);
  //touchButton(&strip, T_BUTS, thresh1, evaluate, empty, setOctave);
  touchButton(&touch, TOUCH, t_tresh, sendNote, checkNote, stopNote);
  if(midi_cc[curr*2] != midi_cc[!curr*2])
    BLEMidiServer.controlChange(1, 12, midi_cc[curr*2]);
  if(midi_cc[curr*2 + 1] != midi_cc[!curr*2 + 1])
    BLEMidiServer.controlChange(1, 13, midi_cc[curr*2 + 1]);
  curr = !curr;
  delay(10);
}
```

In this part found at the end of the loop section, we are sending MIDI cc. These MIDI cc come from the elaboration of the data of the gyroscope and the accelerometer. These MIDI messages are then sent to SuperCollider which tells to the plug-in what is the peak frequency of the 2 filters F1 and F2.

The sensor data, comprised of 3 acceleration values for the 3 axes from the accelerometer and the information of pitch and roll velocity from the gyroscope, is retrieved from the MPU6050 allotted ram slots trough the Adafruit_MPU6050 arduino library and the valued are added together in a 90/10 ratio in favour of the accelerometer, since it has less noise. The row values are firstly converted in angles from -90 to 90 and then subsequently converted in values from 0 to 127.

## SUPERCOLLIDER:

Our SuperCollider code defines four different types of synthesizers, each corresponding to a different waveform:

- Sinusoid 
- Square wave 
- Saw tooth 
- Impulse train 

Each synthesizer has parameters such as frequency, pan position, gate, amplitude and detune. Each voice is played with different horizontal panning to create a spatialization effect in the stereo image, this parameter can be modified directly grom the GUI, and the same applies for the detune and amplitude. The frequency is given by the MIDI note that comes from the flexion of the hand and at the same time, the gate says to start and stop that note.

In essence, the code creates and manages multiple voices of synthesizers with four different waveforms to produce a polyphonic and spatialized musical composition. It is basically the core of our project as it manages the MIDI messages, the OSC messages coming from the GUI and interacts with the VST created with Juce.

```
//CONNESSIONE ALLE SORGENTI MIDI
MIDIClient.init;
MIDIIn.connectAll;
```

```
(
MIDIdef.cc(\filter1, {
    arg val;
    ~n.sendMsg("/handMovement/x", val);
    ~j.sendMsg("/handMovement/x", val);
}, 12);
MIDIdef.cc(\filter2, {
    arg val;
    ~n.sendMsg("/handMovement/y", val);
    ~j.sendMsg("/handMovement/y", val);
}, 13);
//SUONA IL BUS
~myBus.play;
)
```

Through a MIDI communication, SuperCollider is capable of receiving MIDI messages (that include MIDI notes and MIDI cc) from Arduino. The MIDI messages are sent from Arduino to the PC through a Bluetooth communication. Here we have the part of code on SuperCollider that is responsible for the MIDI communication between Arduino and SuperCollider.

```
//RICEZIONE DELLE NOTE MIDI DA ARDUINO
MIDIdef.noteOn(\nota, {
    arg vel, note;
    ~noteSet=note;
    ~b1.set(\freq, ~noteSet.midicps);
    ~b2.set(\freq, ~noteSet.midicps);
    ~b3.set(\freq, ~noteSet.midicps);
    ~b4.set(\freq, ~noteSet.midicps);
    ~b5.set(\freq, ~noteSet.midicps);
    ~b1.set(\gate1, 1);
    ~b2.set(\gate1, 1);
    ~b3.set(\gate1, 1);
    ~b4.set(\gate1, 1);
    ~b5.set(\gate1, 1);
    ~n.sendMsg('/MidiNote', ~noteSet);
    postln(note);
});
//ATTIVAZIONE DELLA NOTA CON IL GATE
MIDIdef.noteOff(\noteOn, {
    ~b1.set(\gate1, 0);
    ~b2.set(\gate1, 0);
    ~b3.set(\gate1, 0);
    ~b4.set(\gate1, 0);
    ~b5.set(\gate1, 0);
});
```

As we can see, SuperCollider receives MIDI note messages and in the second part MIDI cc messages in order to control the frequencies of the filter. The last part is related to the activation of the note, indeed if the message is a NoteOn message the function will switch on the gate and send an OSC message to Processing so that it can be visualized as a feedback for the user.

With these sections of code, SuperCollider will search an available VST plugins in the default VST search paths. To do this we use the Supercollider extension “VSTPlugin” to store the VST in a Ugen. In the second line instead, we create an audio bus to store the source audio signals. The plug-in will take this wave form, add its effect and then replace this final wave on the bus. With VSTPluginController it’s possible to manage the parameters of the filters changing their resonance frequencies.

```
// search of available VST plugins
VSTPlugin.search;
~myBus = Bus.audio(s, 2);
```

```
(
SynthDef(\effetto, { arg bus;
    //ReplaceOut.ar(bus, VSTPlugin.ar(In.ar(bus, 2), 2));
    ReplaceOut.ar(bus, VSTPlugin.ar(In.ar(bus, 2), numOut: `[2]));
}).add;
)
(
```

```
//APRO IL PLUGIN, LO ASSOCIO A UN UGEN E LO CONTROLLO CON VSTCONTROLLER
~fx = VSTPluginController(Synth(\effetto, [\bus, ~myBus])).open("CMLS-Juce.dll");
```

```
//SELEZIONA LA FORMA D'ONDA DA UNA PORTA OSC
~r=NetAddr.new("127.0.0.1", 57120);
OSCdef(\synthSelection, {arg selector;
    selector[1].postln;
    switch(selector[1],
        1, {
            ~b1.free;
            ~b2.free;
            ~b3.free;
            ~b4.free;
            ~b5.free;
            ~b1 = Synth.new(\Sin, [freq: ~noteSet, bus: ~myBus]);
            ~b2 = Synth.new(\Sin, [freq: ~noteSet, bus: ~myBus]);
            ~b3 = Synth.new(\Sin, [freq: ~noteSet, bus: ~myBus]);
            ~b4 = Synth.new(\Sin, [freq: ~noteSet, bus: ~myBus]);
            ~b5 = Synth.new(\Sin, [freq: ~noteSet, bus: ~myBus]);
        },
```

The SuperCollider code establishes an OSC connection to the address `127.0.0.1` on port `57120` and defines a listener that responds to messages sent to the address `/menu`. Upon receiving a message, it prints the value of the second element of the message `selector[1]`. Depending on this value, it releases five existing synths (`~b1`, `~b2`, `~b3`, `~b4`, `~b5`) and creates new ones with specific waveforms: `\Sin` if the value is 1, `\Impulse` if the value is 2, `\Sawtooth` if the value is 3, and `\Square` if the value is 4. All new synths are created with the frequency `~noteSet` and sent to the bus `~myBus`. This allows dynamically changing the type of sound generated by the synths based on OSC inputs. In the picture we plot only the section for choosing the sinusoid.

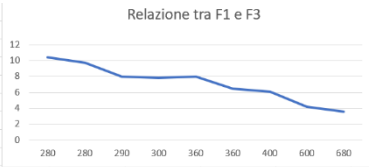
Detune, Unison and Volume are the parameters that the user can chose from the GUI through the three knobs.

```
OSCdef.all;
OSCdef(\detune, {arg detOsc;
    ~b1.set(\det, detOsc);
    ~b2.set(\det, detOsc);
    ~b3.set(\det, detOsc);
    ~b4.set(\det, detOsc);
    ~b5.set(\det, detOsc);
    postln(detOsc);
}, '/Detune', ~r);
OSCdef(\unison, {arg g1, g2, g3, g4, g5, p1, p2, p3, p4, p5;
    ~b1.set(\gate2, g1, \pan, p1);
    ~b2.set(\gate2, g2, \pan, p2);
    ~b3.set(\gate2, g3, \pan, p3);
    ~b4.set(\gate2, g4, \pan, p4);
    ~b5.set(\gate2, g5, \pan, p5);
}, '/Unison', ~r);
OSCdef(\vol, {arg volOsc;
    ~b1.set(\amp, volOsc);
    ~b2.set(\amp, volOsc);
    ~b3.set(\amp, volOsc);
    ~b4.set(\amp, volOsc);
    ~b5.set(\amp, volOsc);
}, '/Volume', ~r);
```

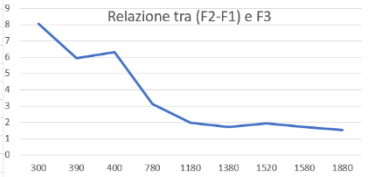
JUCE:

In order to develop the filters on the VST, using Juce, we consider F1 and F2 (the two cut-off frequencies of the two formants that correspond to two of the three band-pass filters) and through some deduction from online data we evaluated the third formant (F3) of the third filter. Through these filters the sound wave coming from SuperCollider can be reshaped in order to sound more similar to a human voice. While we are moving our hand (with the glove), F1, F2 and F3 change to modify the input signal. The sound's change is heard through the differentiation between vowels.

| | | | | | |
|---|---|---|---|---|---|
| i | 280 | | | 2920 | 10.428571 |
| e | 280 | | | 2720 | 9.7142857 |
| ae | 290 | | | 2320 | 8 |
| a | 300 | | | 2340 | 7.8 |
| o | 360 | | | 2880 | 8 |
| u | 360 | | | 2340 | 6.5 |
| uu | 400 | | | 2420 | 6.05 |
| i2 | 600 | | | 2520 | 4.2 |
| o2 | 680 | | | 2440 | 3.5882353 |
| | | | | | |
| i | | 680 | | 2320 | 3.4117647 |
| e | | 700 | | 2420 | 3.4571429 |
| ae | | 1000 | | 2520 | 2.52 |
| a | | 1460 | | 2440 | 1.6712329 |
| o | | 1540 | | 2340 | 1.5194805 |
| u | | 1680 | | 2340 | 1.3928571 |
| uu | | 1800 | | 2920 | 1.6222222 |
| i2 | | 1860 | | 2720 | 1.4623656 |
| o2 | | 2240 | | 2880 | 1.2857143 |
| | | | | | |
| | | | | | |
| i | 400 | 700 | 300 | 2420 | 8.0666667 |
| e | 290 | 680 | 390 | 2320 | 5.9487179 |
| ae | 600 | 1000 | 400 | 2520 | 6.3 |
| a | 680 | 1460 | 780 | 2440 | 3.1282051 |
| o | 360 | 1540 | 1180 | 2340 | 1.9830508 |
| u | 300 | 1680 | 1380 | 2340 | 1.6956522 |
| uu | 280 | 1800 | 1520 | 2920 | 1.9210526 |
| i2 | 280 | 1860 | 1580 | 2720 | 1.721519 |
| o2 | 360 | 2240 | 1880 | 2880 | 1.5319149 |
| | F1 | F2 | F2-F1 | F3 | Relation |








The most important part of our code on Juce is in the process block, in which, after creating 3 identical copies of the input buffer and, setting each of the 3 filter in bandpass at the right frequency (with resonances and gain set at fixed values). Then each copy is normalized before processing then filtered and then the 3 buffers are summed together and the original magnitude scaling is reapplied to the output signal.

The image above shows the relationships between some measurements of the frequencies for each formant (for some phonetic sounds) and their relationships, based on which we decided how to relate the first 2 frequencies and the third one. F3 is evaluated as being

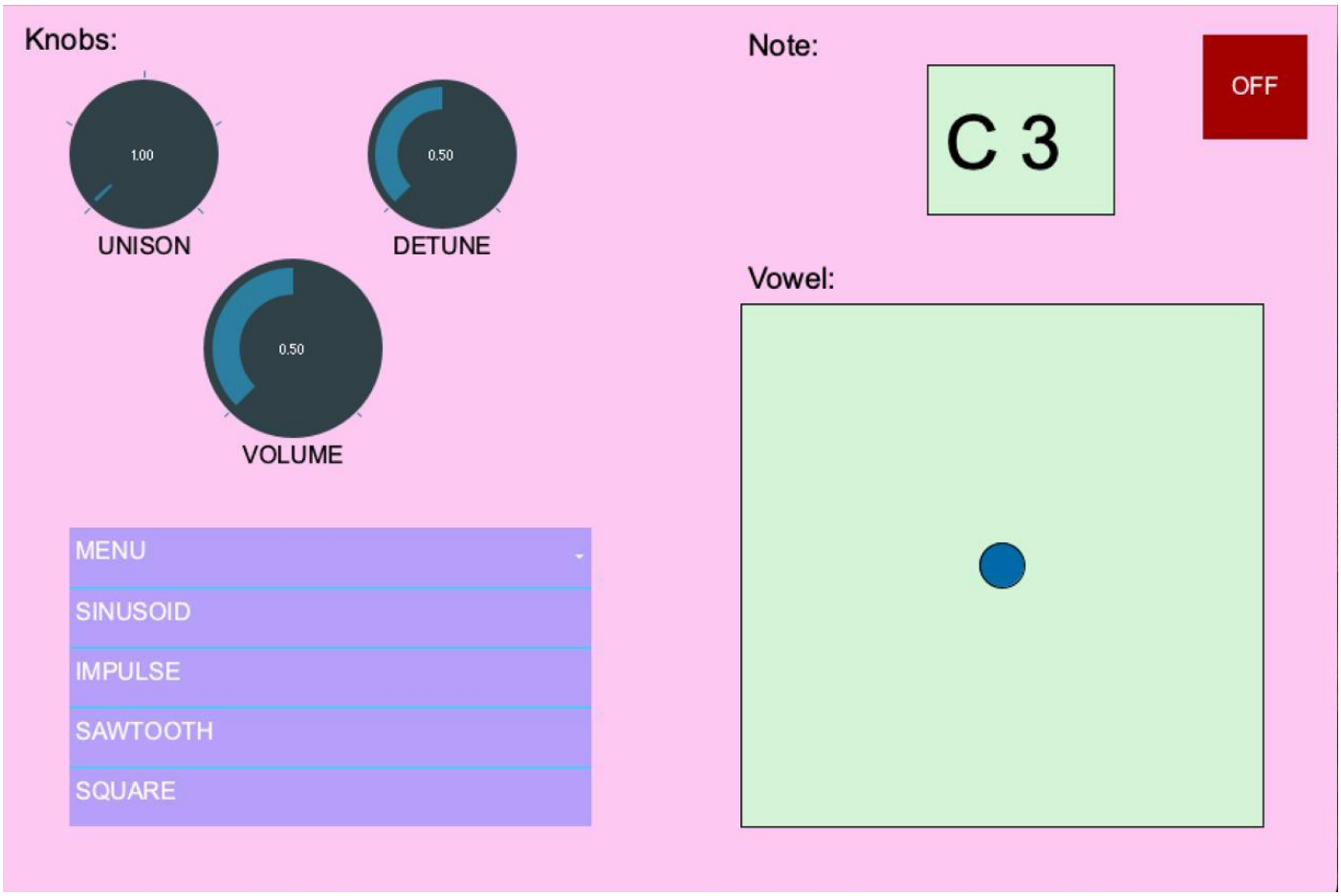
(1-dF/dFmax) (F1/F1max)*F3band + F3min where dF is the difference between the first 2 frequencies while F1 and F3 are the frequencies itself.

After some research trough the dsp libraries the filter that was decided to be used is the VariableStateTPTFilter found in the juce::dsp:IIR namespace which is a filter that can be set as bandpass, lowpass or high pass each time.

The previously described Frequency calculation, together with the calculation of the first to frequencies (the values from 0 to 127 are adapted to the ranges [F1min, F1max] and [F2min, F2max] respectively) is executed in a separate function called during the process block, while during the message Received OSC callback the value is just stored as is in an allotted variable for later computations.

The values that has been chosen as resonances and gains of the 3 filters are [10,8,2] and [1.5*,1.25, 1.1] respectively, which are based on experiments and not based on any model. Note that for lower frequencies the first gain is increased and decreases linearly with the value of the midi message to 1.5.

PROCESSING:




The picture above is our graphical interface created with Processing.
The three knobs are linked with SuperCollider trough the OSC protocol and control:

- The volume of the output
- The detune of the waves, it is a variable that creates a small oscillation of the frequency
- Unison, with this parameter the user can change the number of voices of unison each with a different pan, which combined with the detune adds harmonic content to the sound.

The drop-down menu instead lets the user select the wave shape.

The button OFF let the user shut down the interface and stops the interpreter of SuperCollider

Inside the higher green rectangular the user can see which note is played with the hand. The number explains the octave.

The green square shows a point that moves following the motion of the hand.

## CONCLUSIONS:

As we can hear, since we are controlling only three formants, the output in not exactly similar to a human voice; to reach this goal, are needed more than three formants, but in our case in not possible as we have not enough physical input to control them.

The note selection is not that precise, this is probably to be blamed on the flex sensor which wasn't rigid enough to maintain shape. This component could be substituted for a more rigid option in a revised version.

The initial goal was to detect the notes through the spatial position of the hand added to the orientation we are using now, but as we tested it with the MPU6050 sensor we discovered that it was not possible due to a high amount of noise and drift in the values that made the doubly integrated values of position, obtained by the acceleration, completely useless. We in the end decided to opt for using bot accelerometer and gyroscope values together for detecting only orientation in order to reduce the reciprocal drift and errors (to be viewed as uncorrelated).