FabStick

Group ID: 6 - FabFour

Amico Stefano Antonio, 10937333

Anand Abhijeet, 10859656

Pletti Manfredi, 10493741

Portentoso Alice, 10664207

1 System Overview

The developed music system, FabStick (A Fabulous [drum] Stick) is a combination of hardware, software and passion.

The system generates 'hit data' based on a performer's movements, sensed by an Arduino MKR WiFi 1010 (equipped with a BMI160 accelerometer sensor).

The system is designed to capture movement and velocity data from a stick, much like a drumstick. The data is transmitted wirelessly through UDP packets in OSC messages format to Supercollider, where audio is generated.

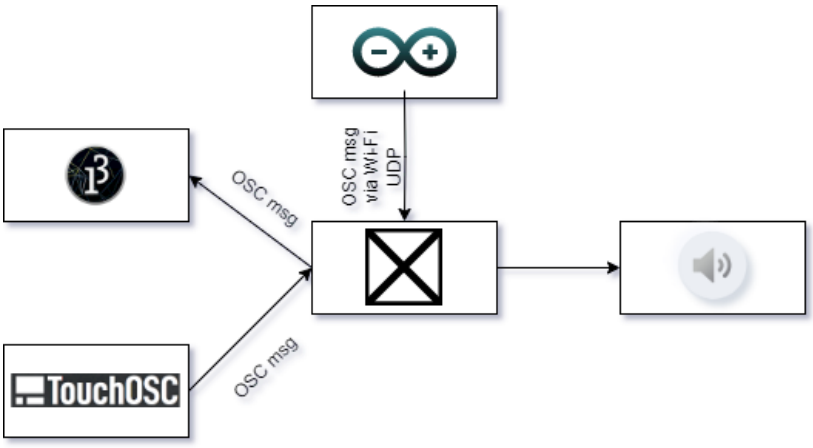

Figure1: FabStick block diagram

Simultaneously, the performer has real-time control over amplitude, filter, reverb, and delay via a wireless controller based on TouchOSC. A Processing visual gives feedback of the system state, representing the 'position' of the shaped sound and ambient controls.

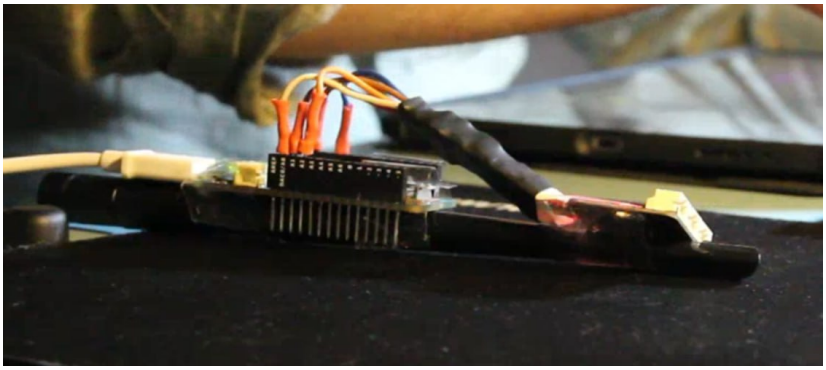

Figure2: FabStick prototype

2 Code Overview - Arduino Script

The Arduino script is the initial stage of the system. It is responsible for:

- Setting up the Wi-Fi module
- Initializing the BMI160 accelerometer and setting its range and rate
- Reading data from the accelerometer
- Detecting strokes based on acceleration and slope data
- Sending stroke data as OSC message via UDP packets to Supercollider

Connection Setup

The system requires a Wi-Fi network to transmit data from the Arduino to Supercollider. The script first checks if a Wi-Fi module is present. If the module is available, it attempts to connect to the Wi-Fi network specified by the ssid and pass variables. Once the connection is established, the UDP communication is initialized using the Supercollider machine IP address and port (default for Supercollider 57120).

Accelerometer Initialization and Data Acquisition

The BMI160 accelerometer is initialized with I2C communication, set to a 16g range, and set to a rate of 800Hz. Acceleration is measured by calling the `calcAccelModule()` function. This function reads accelerometer data, calculates the acceleration on each axis, and then returns the average value.

Stroke Detection

To detect a stroke, there are three specific conditions that must be met:

1. The magnitude of the average acceleration of the penultimate recorded sample must exceed a predetermined threshold.

2. In the last n' recorded samples, there must be a rapid increase (or decrease) in the average acceleration value. This must be followed by a quick reversal of the acceleration value previously recorded. To do so, we run through the second-to last and the last n-value (in this case, we chose 4 as the n-value) to observe if there is at least one spike that goes beyond the threshold value. If there is one, we verify that the corresponding spike in the last n-value is in the opposite direction and goes beyond its expected threshold.

3. No strokes should be detected within the last n milliseconds.

In summary, a stroke is detected when the acceleration undergoes a rapid change in direction due to the impact of the drumstick on the chosen surface.

Velocity calculation

Initially, the plan was to integrate the acceleration to calculate the impact velocity. However, this approach was deemed unfeasible due to multiple issues. One of the main problems was that the accelerometer would consistently measure an average value of 1 g when the drumstick was perpendicular to the x-axis, understood as the drumming surface. This provided imprecise values when measuring the speed with which the drumstick moves on a 3-axis basis.

Simultaneously, the accelerometer readings during the moment of impact had the tendency to be unpredictable. It was then decided to withhold any kind of smoothing to the signals received to ensure that rapid changes in acceleration could be precisely detected.

Once analyzing all relevant data, it was possible to observe that the acceleration values retrieved prior to impact (ca. 3-4 ms before) could provide an indication of the speed with which the drumstick impacts the surface.

Therefore, the following simplified approach was adopted:

When a stroke is detected:

1. The n value before acceleration is analyzed along each axis and with the respective calculation of their magnitude. The most recent m values are not taken into consideration due to their unpredictability, as stated.

2. To compensate for the m-values not taken into consideration, we employ polynomial regression interpolation to reconstruct the missing data based on the prior identified values.

3. Once the interpolated curve is obtained, the impact velocity is estimated through a process of calculation which allows us to obtain a resulting value that is transmitted via OSC.

By narrowing on the acceleration values just before the stroke and compensating for the skipped values through interpolation, a reliable estimation of the drumstick's impact velocity is achieved.

```
void detectStroke() {
  // Check if enough time has passed since the last detected stroke.
  if (currentTime - previousStrokeTime < minStrokeInterval) {
    return;
  }
    float acceleration = accelerationBuffer[(commonIndex - 1 + bufferSize) % bufferSize
    ];
    // check condition 1, which is whether the absolute value of the acceleration
    exceeds the threshold.
    if (abs(acceleration) > accelThreshold) {
      int prevSlopeIndex = (commonIndex - 1 + bufferSize) % bufferSize;
      for (int i = prevSlopeIndex; i != (prevSlopeIndex + bufferSize - 4) % bufferSize;
    i = (i - 1 + bufferSize) % bufferSize ) {

      if ((acceleration >= 0.0 && slopeBuffer[i] > prevSlopeThreshold) || (acceleration
    < 0.0 && -slopeBuffer[i] > prevSlopeThreshold)) {
        // Check condition 3, which is after the peak there must be a slope opposite to
    the one corresponding to the peak.
        int nextSlopeIndex = commonIndex;
        if ((acceleration >= 0.0 && slopeBuffer[nextSlopeIndex] <= nextSlopeThreshold)
    || (acceleration < 0.0 && -slopeBuffer[nextSlopeIndex] <= nextSlopeThreshold)) {
            previousStrokeTime = currentTime;  // Update the time of the last detected
    stroke.
            float velocity = constrain(map(calcVelocity(),0, 45, 0, 127), 0, 127)/127.0;
            sendValue(velocity);
            Serial.print("Stroke detected!: velocity = ");
            Serial.println(velocity);
        }
      }
    }
  }
}
```

Listing 1: Stroke detection

3 Code Overview - Supercollider

This SuperCollider script accomplishes a number of things, making use of OSC (Open Sound Control) to communicate with other applications, routing signals through audio and control busses, and creating different sound effects. Below is a detailed explanation of its various components.

Initial Setup

The script initializes several variables for further use: `sensorValue` and `sensorTrigger` for storing the values received from an external sensor via OSC, `x` and `y` as control busses to handle incoming touch OSC data, and `reverbBus` and `delayBus` as audio busses for handling audio signals passed through reverb and delay effects respectively.

Synth Definitions

This part of the script defines several synth designs (i.e., reverbEffect, delayEffect, hit). These are effectively the templates for sound generators and effects that will be used later in the script.

- **reverbEffect**: This synth processes an audio input with the GVerb unit generator, which creates a reverb effect. The parameters roomsize, revtime, and rwet (dry/wet control) can be set upon instantiation of the synth. The bypass option allows the wet (processed) and dry (unprocessed) signals to be mixed or the dry signal to be output directly.
- **delayEffect**: This synth uses the CombC unit generator to create a delay effect. Similar to the reverbEffect synth, it has parameters for the delay settings, a bypass option, and the ability to mix wet and dry signals.
- **hit**: This synth generates noise using a white noise generator and bandpass filter. The noise is triggered by an impulse, and its frequency and bandwidth parameters are controlled by the x and y control bus values. The output is sent to both the reverb and delay busses.

Effects Synth Instantiation

The script then instantiates the effects synths for reverb and delay.

OSC Management

This section of the script receives and handles OSC messages.

- The `/sensor` OSC message updates `sensorValue` and triggers the `hit` synth.

```
//receive OSC sensor values from Arduino
  OSCdef(\sensor, { |msg|
    ~sensorValue = msg[1].asFloat;
    ~sensorTrigger = 1;
    // Trigger a new hit synth with the sensor value as the amplitude
    Synth.new(\hit, [\amp, ~sensorValue, \trig, ~sensorTrigger]);
    // Send sensor value to Processing
    ~processingAddr.sendMsg("/sensor", ~sensorValue);
  }, '/sensor', recvPort: 57120);
```

Listing 2: Receiving OSC sensor data and forward to Processing

- The `/XY/1` OSC message updates the `x` and `y` control busses and sends their values to a separate processing application.

```
  //address for Processing
~processingAddr = NetAddr("localhost", 45777);
//receive TouchOSC XY
OSCdef(\touchOSCListener, { |msg, time, addr, recvPort|
  var x, y;
  if(msg[0] == '/XY/1') {
    x = msg[1].asFloat;
    y = msg[2].asFloat;
    // write x and y to the control buses
    ~x.set(x);
    ~y.set(y);
    // Send x and y to Processing
    ~processingAddr.sendMsg("/XY", x, y);
  }
}, '/XY/1', recvPort: 57120);

```

Listing 3: Receive TouchOSC XY data end forward to Processing

- OSC messages for `/roomsize`, `/revtime`, `/rwet`, `/delaytime`, `/decaytime`, and `/dwet` all update corresponding parameters in the reverb and delay synths, with feedback given via the post window and messages sent to the processing application.

4 TouchOSC Layout

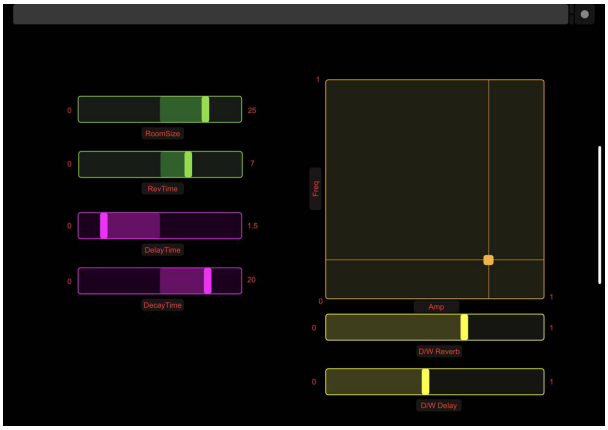

Figure3: TouchOSC Layout

Each controller sends OSC messages as the corresponding labeled values. Very important is the XY pad wher user can shape the stick's sound in amplitude and frequency, where x axis controls the amplitude and y axis controls the cutoff frequency of a high-pass filter.

Actually TouchOSC does not support value parameters visualization, this is why we included min/max values for each control.

5 Code Overview - Processing

The Processing script is essentially a program that visualizes sound parameters. It does so by communicating with the SuperCollider program through OSC messages.

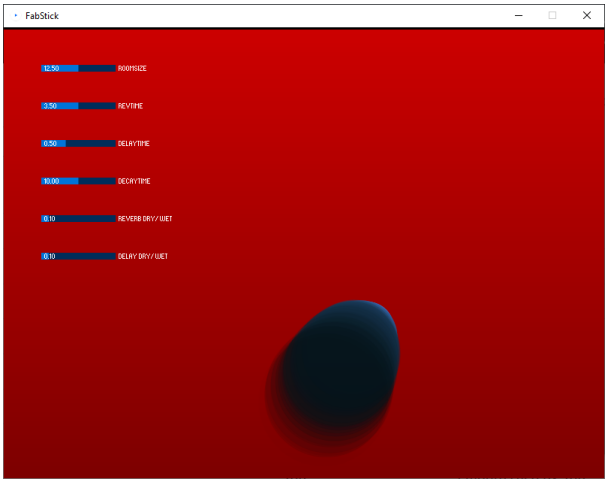

Figure4: Processing Visual

1. **Imports**: It starts by importing the necessary libraries: `oscP5`, `netP5`, and `controlP5`. `oscP5` is used for handling OSC communication, `netP5` is used for networking capabilities, and `controlP5` is used to add and control interactive UI elements like sliders.

2. **Global Variables**: The script then declares global variables that it uses throughout the program. `OscP5` and `NetAddress` are objects used for OSC communication. `x` and `y` will hold values received from OSC messages. The `ArrayList<PVector>` will hold a trail of points to visualize on the screen. `trailSize` controls how many points are stored in the trail. `ControlP5 cp5` is the object that manages the UI.

3. **Setup**: In the setup function, the script initializes the `oscP5`, `superCollider` and `cp5` objects, and sets up the initial look of the program with the `size()`, `background()`, `noStroke()` functions.

4. **Sliders**: It creates several sliders using the `cp5.addSlider()` function. These sliders permits sound parameters visualization and their value ranges are set using the `setRange()` method (same set in Supercollider).

5. **Draw**: Inside the draw function, which is run every frame, the script visualizes the trail of points that it's storing. It goes through each point in the trail and draws it on the screen as a colored circle. The color and size of each circle is determined based on the position of the point in the trail

and the x and y values of the point.

Also the background flashes as a /sensor value is received via OSC in warmer color as received value increase.

6. **setGradient**: This is a helper function used to add a subtle gradient to the vector background for a softer look.

7. **OSCEvent**: This function gets called every time an OSC message is received. It first checks if the OSC message has the address pattern `/XY`. If it does, the script gets the x and y values from the message, adds a new point to the trail, and if the trail is too long, removes the oldest point. If the message has other address patterns, it updates the corresponding slider value with the value from the OSC message.

This Processing script listens for OSC messages from SuperCollider that contain the x and y positions of the TouchOSC controller. These positions are then visualized as a trail of circles. Furthermore the script updates the positions of the sliders based on the values it receives from OSC messages.

6 Conclusion

A interesting project as been developed that fuses together different technologies such as Arduino, SuperCollider, TouchOSC and Processing to create a responsive audio-visual interface.

However, like all projects, this system has several areas with potential for improvement:

- **Arduino Stroke Calibration**: The system currently uses fixed thresholds for stroke detection. Depending on the specific use case, these thresholds might need to be calibrated to ensure reliable stroke detection.

- **Supercollider**: More 'pleasant' sound representation could be developed defining more controlled sound manipulation parameters. Infact, actually, WhiteNoise is used for sound generation.

- **Processing**: More complex or varied visualizations could provide a richer, more detailed understanding of the sound data. For instance, the inclusion of waveform visualizations could offer a real-time representation of the sound being produced or altered.

  Also, TouchOSC does not support (actually) value parameters visualization (this is why we have only included mix/max params in the layout) so more space should be assigned in the Processing GUI about them.