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 an 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 to a Python server, which then transforms the data into OSC messages. These messages are processed in Supercollider, where audio is generated.

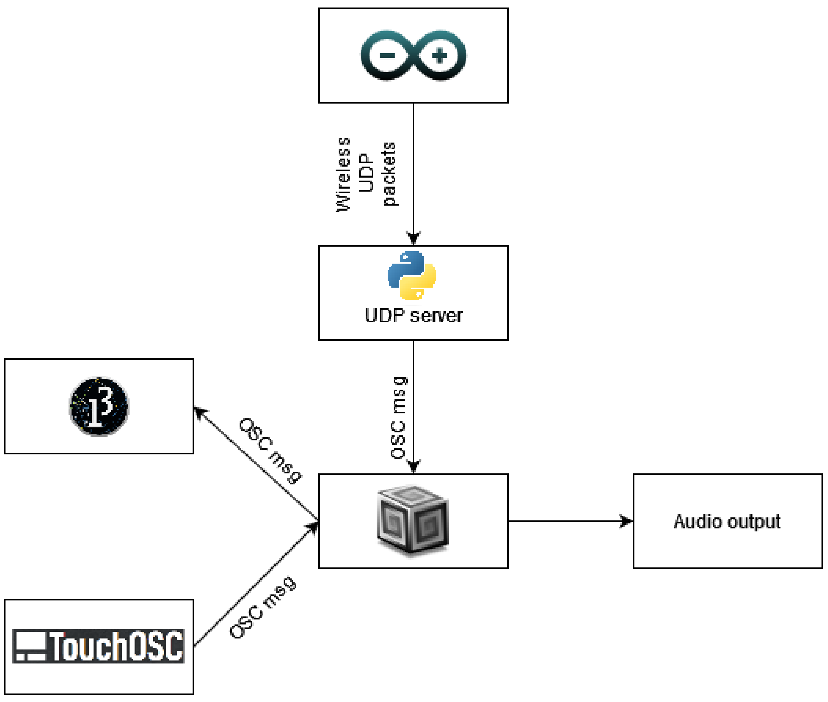


Figure1: FabStick block diagram

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

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 UDP packets to a Python server

Connection Setup

The system requires a Wi-Fi network to transmit data from the Arduino to the Python server. 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 Python server's IP address and port.

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

Stroke detection is based on acceleration and slope. These values are stored in buffers (`accelerationBuffer` and `slopeBuffer`). The system tracks the maximum and minimum values of acceleration and slope to adjust the sensitivity dynamically. The function `detectStroke()` implements the stroke detection logic, which is based on thresholds.

A stroke is detected if:

- The acceleration exceeds the `accelThreshold`
- The previous slope exceeds the `prevSlopeThreshold`

- The next slope does not exceed the `nextSlopeThreshold`
- A minimum time (`minStrokeInterval`) has passed since the last detected stroke

If a stroke is detected, the stroke's velocity is calculated based on the acceleration and the slope at the moment of the stroke and sended as UDP packet.

```
for (int i = endIndex - 1; i != startIndex; i = (i + bufferSize - 1) % bufferSize) {

    float acceleration = accelerationBuffer[i];

    if (abs(acceleration) > accelThreshold) {

      int prevSlopeIndex = i;
      if ((acceleration >= 0.0 && slopeBuffer[prevSlopeIndex] > prevSlopeThreshold) || (
    acceleration < 0.0 && -slopeBuffer[prevSlopeIndex] > prevSlopeThreshold)) {

        if ((acceleration >= 0.0 && slopeBuffer[nextSlopeIndex] <= nextSlopeThreshold)
    || (acceleration < 0.0 && -slopeBuffer[nextSlopeIndex] <= nextSlopeThreshold)) {

            previousStrokeTime = currentTime;

            float velocity = map(abs(acceleration), 3, 10, 0, 127);
            velocity = constrain(velocity, 0, 127);
            float prevSlopeVelocity = map(slopeBuffer[prevSlopeIndex], 3000, 10000, 0,
    127);
            prevSlopeVelocity = constrain(prevSlopeVelocity, 0, 127);

            int finalVelocity = (velocity + prevSlopeVelocity ) / 2;

            Serial.print("Stroke detected!: velocity = ");
            Serial.println(finalVelocity);
            sendValue(finalVelocity);
            return true;
        }
      }
    }
  }
```

Listing 1: Stroke detection

3 Code Overview - Python server scrypt

The Python script essentially works as an intermediary between the Arduino device and SuperCollider. The script uses a UDP server to receive data packets from the Arduino and an OSC client to forward this data to SuperCollider.

UDP Server Setup

The UDP server is set up to listen for incoming packets on port 8888. The IP address is set to ”0.0.0.0”, which in this context means that the server is listening on all available network interfaces. The buffer size for the socket’s recvfrom function is set to 1024 bytes, which should be more than enough for the incoming data packets.

Socket Timeout

The `settimeout()` function is used to ensure that the socket does not block indefinitely when waiting for incoming data. If no data is received within the set timeout period (1.0 seconds in this case), the `recvfrom()` function raises a socket.timeout exception, which is handled later in the script.

OSC Client Setup

The OSC client is set up to send messages to SuperCollider, which is assumed to be running on the same machine (IP address 127.0.0.1) and listening on port 57120.

Main Loop

The script enters an infinite loop where it continually listens for incoming data from the Arduino. When data is received, it is decoded from bytes to a UTF-8 string and then converted to a floating-point number. This value is then printed to the console and sent to SuperCollider as an OSC message. If the `recvfrom()` function times out, the script simply continues to the next iteration of the loop.

KeyboardInterrupt Exception Handling

If the script is interrupted by the user (typically by pressing Ctrl+C), it catches the KeyboardInterrupt exception, prints a message to the console, and closes the socket before terminating.

4 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 python server
  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]);
  }, '/sensor');
```

Listing 2: Receive OSC sensor data

- 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`, `/maxdelaytime`, `/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.

5 Touch OSC Layout

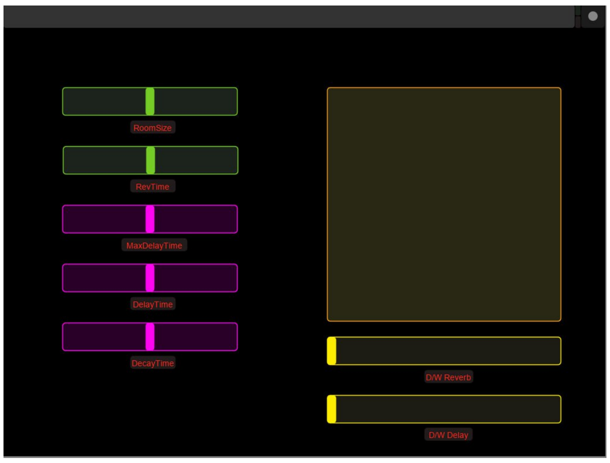
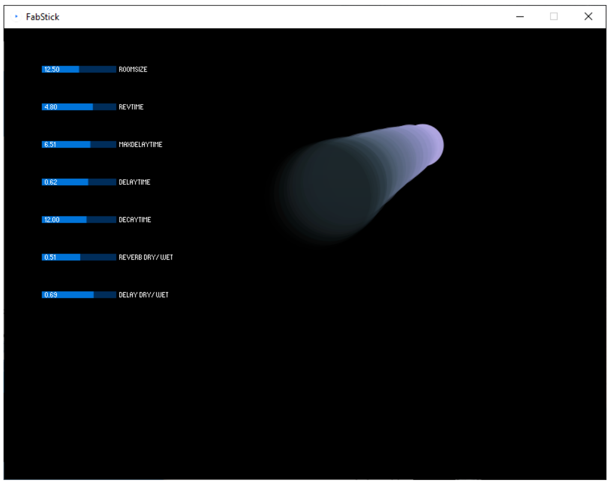


Figure2: TouchOSC Layout

Each controller send OSC messages as the corresponding labeled values.

6 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.



12.50 ROOMSIZE
4.80 REVTIME
6.51 MAXDELAYTIME
0.62 DELAYTIME
12.00 DECAYTIME
0.51 REVERB DRY/WET
0.69 DELAY DRY/WET

Figure3: 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 setted 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.

6. **setGradient**: This is a helper function used to add a subtle gradient to the 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.

7 Conclusion

A interesting project as been developed that fuses together different technologies such as Arduino, Python, 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.

- **Python server**: Potential enhancements to this script could include error handling for cases where the data received from the Arduino is not in the expected format. Also it could include more sophisticated OSC message construction (for example, including timestamps or sending messages to different SuperCollider endpoints based on certain conditions), or additional functionality for managing the state of the system (such as starting and stopping SuperCollider, or setting SuperCollider parameters).

- **Supercollider**: More 'pleasant' sound representation could be developed defining more controlled sound manipulation parameters.

- **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.