POLITECNICO
MILANO 1863

Music And Acoustic Engineering

Computer Music - Languages and Systems
Homework #3
Group: Algorhythmics - A.Y. 2022/2023

YASC

Group Members:

Alice Sironi
Cecilia Raho
Stefano Ravasi
Yan Zhuang

Contents

1 Introduction

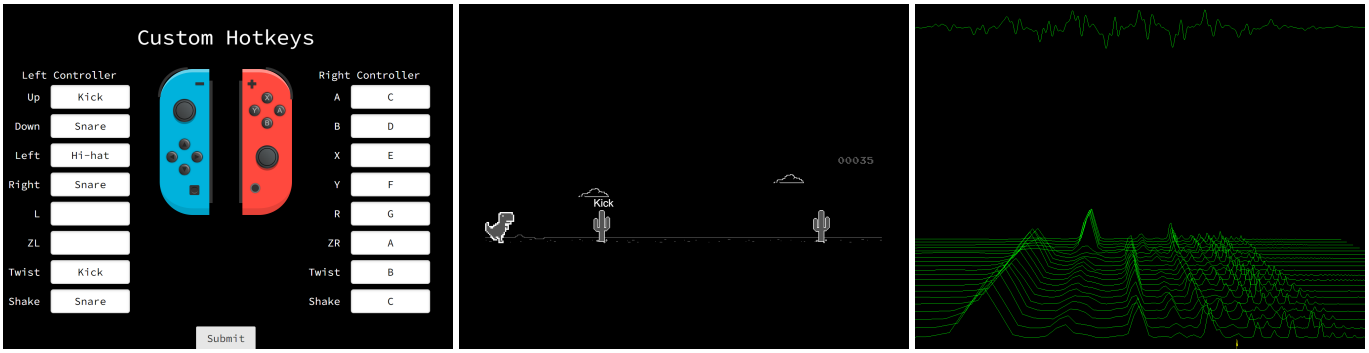
The goal of the project is to to develop an interactive beatboxing recorder performance tool, incorporating Joy-Con controllers via a web interface for gesture-based inputs, Node.js for data handling, and SuperCollider for sound synthesis. The system is able to emulate the beatboxing experience with real-time, responsive feedback to enable an immersive musical performance.

2 Features and GUI

For the whole system, it divided into web system and sound synthesis. The web part involves user interaction, visualization, and real-time communication. The sound Synthesis involves sound generation.

2.1 Web system

- *The hotkey page(fig. 1a)*: it serves as an interface for users to personalize their beatboxing experience. Through this page, users can customize their Joy-Con controller buttons to map to different beatboxing and beatboxing sounds.
- *The dino game page(fig. 1b)*: it serves as an interactive training tool designed to help users familiarize themselves with their custom hotkeys. The key point in our implementation is that the jumping action is triggered only through the correct Joy-Con button that the user has previously mapped in the Hotkey Page.
- *The Spectrum Page (fig. 1c)*: it provides an immersive audio-visual experience by representing sounds as visual spectra.



(a) Hotkey (b) Dino (c) Spectrum

Figure 1: Web System

2.2 Sound synthesis

The sound synthesis handled by SuperCollider, offers a couple of sounds including recorder,hi-hat,snare,kick with relative parameters for customization.

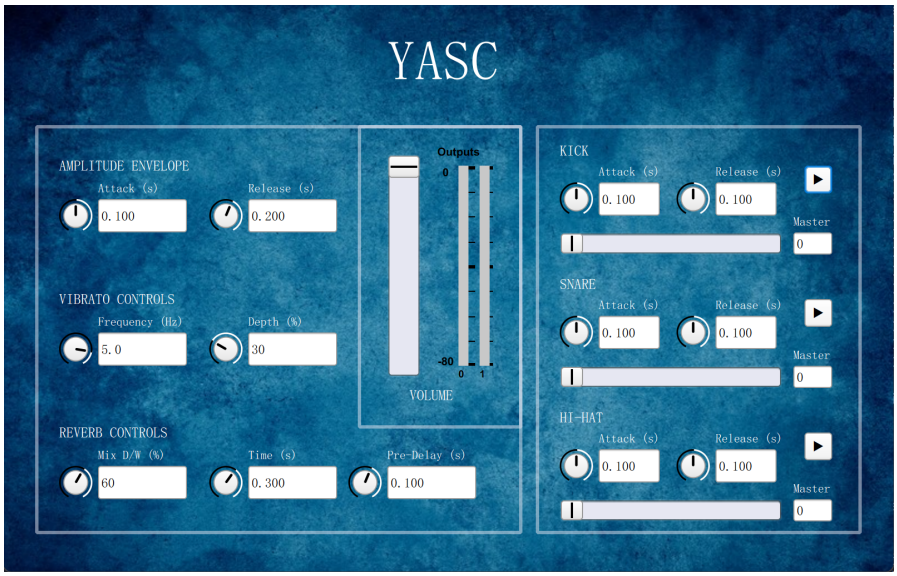


Figure 2: Sound synthesis

3 Implementation

3.1 Overall structure

In this structure(fig. 3), the system tracks users' interactions with Joy-Con controllers and use them as input. The web system acts as the center interface, providing users with an intuitive and interactive to set up and customize their performance experience. SuperCollider serves as the sound synthesis engine, generating the sounds according to the commands from web system via the Open Sound Control (OSC) protocol and providing detailed parameters for users to adjust.

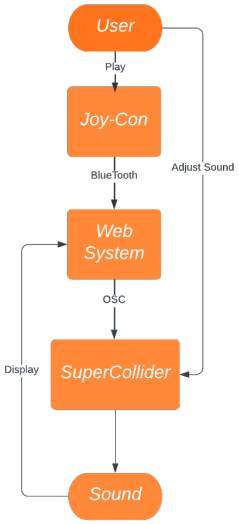


Figure 3: Overall structure

3.2 Web system

The web system mainly uses Websockets for fast communication and Meyda with Three.js for visualizations.

- *Web Socket*: it provide a persistent connection between the client and the server, allowing data to be transmitted as soon as it is available. This significantly reduces the delay between a

user's action on the Joy-Con controller and the sound produced by SuperCollider.

- *Meyda and Three.js*: the Spectrum Page uses the Meyda library to perform audio feature extraction, which provides a spectrum analysis of the sounds generated by the user. The analysis data is then visualized using Three.js, which can help users visualize the frequency spectrum of their performance in real-time.

3.3 SynthDef *flute*

The SynthDef "flute" emulates the sound of a recorder through digital waveguide synthesis. This model is based on Perry Cook's one with the addition of a simple vibrato.

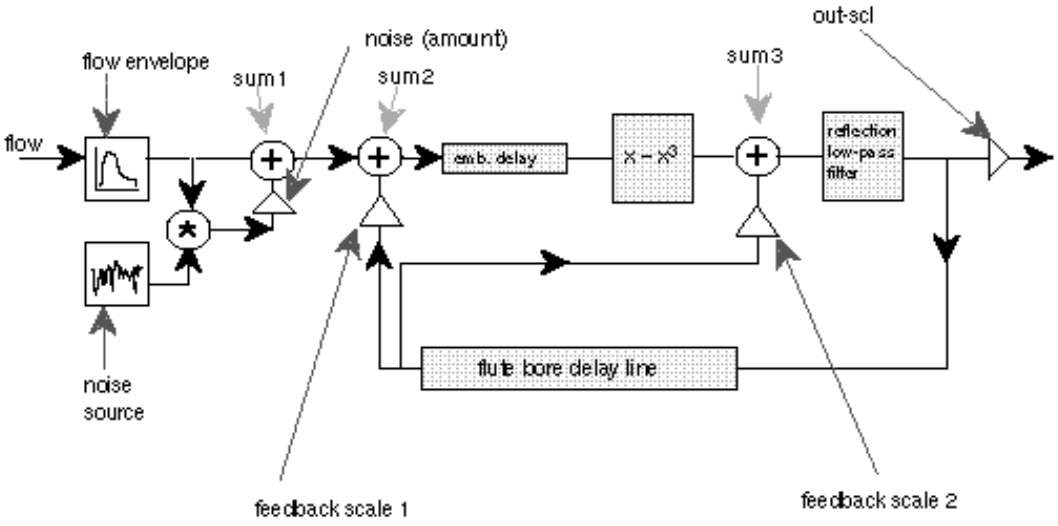


Figure 4: Perry Cook's digital waveguide flute model

The procedure is summarized by the following steps:

- The first part of the sound (*sum1*) is generated through a clipped noise modulated by an envelope and summed with a low frequency sinusoidal oscillator to model vibrato.
- The second sound (*sum2*) is obtained as the sum of the first one and the actual output processed

by a cubic interpolated delay line (used for bore effect emulation, one period's worth of samples long), modulated by a coefficient smaller than one (to prevent instability in the feedback loop).

- The second sound (*sum2*) is processed by another cubic interpolated delay line (used for embouchure effect emulation, half period's worth of samples long) and the cubic polynomial $x - x^3$ is computed for each sample: this result is then summed to the actual output (processed as described in the previous point) modulated by a new small coefficient (always smaller than one for the same reason stated before) to give the third sound (*sum3*).
- A low-pass filter is applied to the the third sound (*sum3*) (to emulate the faster decay of high frequencies) and the result is finally sent to the output.

```
SynthDef("waveguideFlute", {
    arg scl = 0.2, freq = 440, bend = 0, ipress = 0.9, ibreath = 0.09, ifeedbk1 = 0.4, ifeedbk2 = 0.4, dur = 20000, gate = 1, amp = 2, out =
0, atk = 0.1, rel = 0.2, vdep = 0.3, vfr = 5;
    var kenv1, kenv2, kenvibr, kvibr, sr, cr, block;
    var poly, signalOut, ifqc;
    var aflow1, asum1, asum2, afqc, atemp1, ax, apoly, asum3, avalue, atemp2, aflute1;
    var fdbckArray;
    sr = SampleRate.ir;
    cr = ControlRate.ir;
    block = cr.reciprocal;
    ifqc = freq * bend.midiratio;
    kenv1 = EnvGen.kr(Env.new([ 0.0, 1.1 * ipress, ipress, ipress, 0.0 ], [ 0.06, 0.2, dur - 0.46, 0.2 ], 'linear' ), gate);
    kenv2 = EnvGen.kr(Env.new([ 0.0, 1.0, 1.0, 0.0 ], [ atk, dur - 0.02, rel ], 'linear' ), gate, doneAction: 2);
    kenvibr = EnvGen.kr(Env.new( [ 0.0, 0.0, 0.3, 0.3, 0.0 ], [ 0.5, 0.5, dur - 1.5, 0.5 ], 'linear'), gate);
    aflow1 = LFClipNoise.ar( sr, kenv1 );
    kvibr = SinOsc.ar( vfr, 0, 0.1 * kenvibr );
    asum1 = ( ibreath * aflow1 ) + kenv1 + kvibr;
    afqc = ifqc.reciprocal - ( asum1/20000 ) - ( 9/sr ) + ( ifqc/12000000 ) - block;
    fdbckArray = LocalIn.ar( 1 );
    aflute1 = fdbckArray;
    asum2 = asum1 + ( aflute1 * ifeedbk1 );
    ax = DelayC.ar( asum2, ifqc.reciprocal - block * 0.5, afqc * 0.5 - ( asum1/ifqc/cr ) + 0.001 );
    apoly = ax - ( ax.cubed );
    asum3 = apoly + ( aflute1 * ifeedbk2 );
    avalue = LPF.ar( asum3, 2000 );
    aflute1 = DelayC.ar( avalue, ifqc.reciprocal - block, afqc );
    fdbckArray = [ aflute1 ];
    LocalOut.ar( fdbckArray );
    signalOut = avalue;
    OffsetOut.ar(out, [ signalOut * kenv2, signalOut * kenv2 ] );
}).add;
```

Figure 5: Recorder implementation

3.4 Drum Kit

3.4.1 SynthDef *kick*

The SynthDef "kick" emulates the sound of a kick through a *SinOsc* implements with these elements:

- The percussive envelope used as amplitude of the sinusoidal oscillator (*mul*). It has three values, attack time, release time and peak amplitude, that can be modified with the knobs on the GUI.
- The ramped signal, implemented with a line generator, is used as frequency of the sinusoidal oscillator (*freq*).

```
SynthDef("kick", {arg out = 0, amp = 0.3, sinfreq = 60, glissf = 0.9, att = 0.01, rel = 0.45, pan = 0;
    var env, snd, ramp;
    env = Env.perc(att, rel, amp).kr(doneAction: 2);
    ramp = XLine.kr(
        start: sinfreq,
        end: sinfreq * glissf,
        dur: rel
    );
    snd = SinOsc.ar(freq: ramp, mul: env);
    snd = Pan2.ar(snd, pan);
    Out.ar(out, snd);
}).add;
```

Figure 6: Kick implementation

3.4.2 SynthDef *snare*

The SynthDef "snare" emulates the sound of a snare, implemented through the sum between a SinOsc (with frequency equal to a defined sinfreq and amplitude equal to the percussive envelope) and an high-pass filter applied to a Pink Noise source.

```
SynthDef("snare", {arg out = 0, amp = 0.1, sinfreq = 180, att = 0.01, rel = 0.2, ffreq = 2000, pan = 0;
    var env, snd1, snd2, sum;
    env = Env.perc(att, rel, amp).kr(doneAction: 2);
    snd1 = HPF.ar(
        in: PinkNoise.ar,
        freq: ffreq,
        mul: env
    );
    snd2 = SinOsc.ar(freq: sinfreq, mul: env);
    sum = snd1 + snd2;
    Out.ar(out, Pan2.ar(sum, pan));
}).add;
```

Figure 7: Snare implementation

3.4.3 SynthDef *hi-hat*

The SynthDef "hi-hat" emulates the sound of a hi-hat, implemeted through a percussive sound using a white noise source and high-pass filter. The white noise signal is passed through a high-pass filter, with a cutoff frequency specified by the ffreq variable. The resulting filtered signal is then multiplied by the control-rate envelope env.

```
// hi-hat synthDef

SynthDef("hihat",
    {arg out = 0, amp = 0.5, att = 0.01, rel = 0.2, ffreq = 6000, pan = 0;
    var env, snd;
    env = Env.perc(att, rel, amp).kr(doneAction: 2);
    snd = WhiteNoise.ar;
    snd = HPF.ar(in: snd, freq: ffreq, mul: env);
    Out.ar(out, Pan2.ar(snd, pan));
}).add;
```

Figure 8: Hi-hat implementation

4 Conclusions and further improvements

In conclusion, the system effectively captures users' inputs, offers an intuitive interface for customization and practice, and generates rich, dynamic sounds, resulting in a novel and engaging user experience.

Some of the following ideas could be implemented for the future work.

For the web system:

- In order to further reduce the delay, use JUCE with WebSocket to replace SuperCollider.
- Explore the possibility of integrating with other motion-sensing devices or MIDI controllers, expanding the performance options.
- Implement a feature that allows users to record, save, and replay their performances.

For the sound synthesis:

- Offer more sophisticated sound parameters for users to manipulate, allowing for greater customization of the sounds generated by SuperCollider.
- The current recorder model uses a cubic non-linearity to simulate the effect of the mouthpiece. More accurate results might be achieved by incorporating higher-order non-linearities.