Computer Music Languages and Systems

Homework n° 3
FM synthesis

10574752

10751438

10612929

10486570

Master of Science in Music and Acoustic Engineering

POLITECNICO
MILANO 1863

Abstract

Our group worked on **Assignment 3**: Create an instrument based on FM synthesis and an interface for controlling it. In order to use the application, run the server using Node.js with the command `node .\server.js`. Then connect to the url `localhost:55123` in a browser (it may take some seconds to load the ML model). Open the synth in SuperCollider and run all the code, it will start to listen for OSC messages through the port 57120. The link containing the source code can be found on GitHub at: `https://github.com/EllDy96/CarlGang/tree/Homework3`. Enjoy!

Chapter 1

FM synthesis development

SuperCollider implementation

Synthesizer definition

We design all the sound generation and manipulation via **Supercollider**. To create a Frequency modulation Synthesizer we create a simple pattern based on one Sinusoidal oscillator with feedback as a modulator an another sinusoidal oscillator as a carrier. In order to rightly tune this pattern we follow three main guideline rules:

1. The amplitude of the modulation oscillator controlls the number of the carrier's side-band frequency peaks that will appear in the frequency response.

2. The frequency interval at which the side-band peaks will appear is equals to the modulator frequency.

3. The output signal will tend to produce a clear sense of pitch when the carrier and the modulator frequencies form a simple ratio (e.g. 2:1).

Starting from a frequency midi input, to obtain a pleasant sound, we model the modulation and the carrier frequency using two tunable parameters called `carrierRatio` and `modulationRatio` that allows us to guarantee a integer ratio between these two. We scale the amount of the sub-bands components using another parameter called `index` that then will be controlled by the user. To clean the final sound and make it less muddy we apply an high-cut filter to the upper spectrum range and a low pass filter to the lowest part. Finally we add a Reverb and a tunable parameter that allow the user to control the Reverb amount in real time.

To make the live interaction more interesting we implement a way to allow the user to control in real time also the modulation feedback and the cut-off frequency of a Low pass filter.

MIDI control

We defined a global variable (*monoNote*) as an instance of the Synthesiser we defined, in order to easily have the access to the parameters of the synth and change them immediately. MIDI controls are composed by a *noteOnFunc* that receives the velocity, the note, the MIDI channel and the MIDI port and set the frequency and the amplitude. It is also defined a *noteOffFunc* that receives the same parameters of the noteOnFunc and set the amplitude to zero.

OSC messages

The OSC communication is set to the localhost (id "127.0.0.1) at port 57120. The OSC message is composed by 4 parameters, respectively:

- msg[1] is the horizontal value of the centroid
- msg[2]is the vertical value of the centroid
- msg[3] is the palm-index distance
- msg[4] is the palm slope

The parameters received have all a value between the interval $[0, 1]$ so we mapped them in proper intervals in order to give to the user a pleasant effect and interaction when the hand is moved. After the mapping the hand's parameters are set to the corresponding synth's parameters:

centroid x $\rightarrow$ feedback
centroid y $\rightarrow$ modulation amplitude
palm-index distance $\rightarrow$ Low Pass cut off frequency
palm slope $\rightarrow$ reverb

Chapter 2

The User interaction

Interaction design

Modules and libraries used: Node.js, Socket.io, Express, p5.js, ml5.js, osc.js.

The synthesizer can be controlled through hand gestures captured from a webcam. For the hand pose recognition, we used a pre-trained ML model from ml5.js (a javascript framework for creative coding built on top of TensorFlow.js), which takes frame by frame the video stream and return the coordinates of 21 points of the hand (this process is GPU intensive, even though the model is lightweight, a system with a dedicated graphic card is advised for best results).

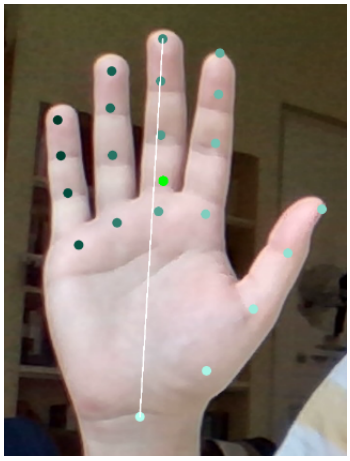

Figure 2.1: 21 hand points and control parameters

From these 21 points (x and y coordinates) we compute 3 parameters:

- the centroid (green dot) with coordinates:

$$x_c = \frac{\sum_{i=1}^{21} x_i}{21}$$

$$y_c = \frac{\sum_{i=1}^{21} y_i}{21}$$

- the distance between the tip of the middle finger (x_{mf}, y_{mf}) and the base of the palm (x_{pb}, y_{pb}) (length of the white line):

$$d = \sqrt{(x_{pb} - x_{mf})^2 + (y_{pb} - y_{mf})^2}$$

- the orientation of the hand (the slope of the white line) between $[0, \pi]$:

$$s = \left| \arctan\left(\frac{y_{mf} - y_{pb}}{x_{mf} - x_{pb}} \right) \right|$$

The user interface is hosted as a web page/application in an Express server, the connection is set up through the framework Socket.io. All the control parameter mentioned above are computed in the client and then sent to the server. From the server, the parameters are written in OSC messages and forwarded to SuperCollider. This last past is handled through the library osc.js, which can generate OSC messages from javascript objects and establish a connection with a receiver (i.e., SuperCollider through an UDP connection). The OSC message has only one path "/params" in which are contained all the parameters as floats.

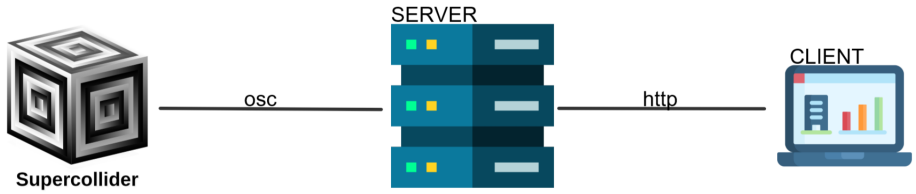

Figure 2.2: Application architecture

The user interface, as we just said, is a web application in which we imported the libraries ml5.js and p5.js. We set p5.js in Instance Mode in order to manage 4 different sketches which compose the main window. The bigger p5 sketch at the top left is the one visualizing the webcam, the 21 points of the hand and the control parameters. The other three are a representation of

the control parameters using psychedelic animations. At the bottom left we have a visualization for the hand orientation, at the top right for the x and y position of the centroid, and finally at the bottom right, for the distance between the middle finger and the palm base. Going into more depth on the animations implementation, we used as a reference the examples on the `https://p5js.org` website and a number of Youtube tutorials, in order to properly manage all the instructions in the code.

Beginning from the **"Squared Rose"** animation, we wanted to create an easily interpretable as visually impactful effect describing the variation of the LPF cut off frequency.
The main characteristics of the code itself can be riassumed in the following choices:

- mapping the behaviour of the changing colors with $sin()$ and $cos()$ functions, creating pleasant and smoothing transitions
- introduce a constant rotation of the figure using the `rotate` function, and considering as argument *frameCount*, which contains the number of frames that have been displayed since the program started. You can reduce the rotation speed dividing *frameCount* by a proper value. Without this rotation we only have a series of concentric squares, whose relative positions don't change over time.

About the **"Sun Sphere"** animation instead, the astonishing effect given by the cohesion between the central sphere (created with a for cycle of multiple ellipses) and the colorful rays (created with a for cycle of multiple triangles) is essentially possible thanks to the double rotation implemented, through the functions `rotateX` and `rotateY`. The behaviour of the colors is similar to the previous animation, except for the increased velocity in the transitions. This "2 in 1 animation canvas" is used to describe the variations of feedback and modulation amplitude.

Talking about the **"Double Square"** animation, we decided to implement an immediately readable effect, describing the variation of the reverb volume. The 2 squares gradually decrease & increase their dimension following the hand orientation, as we'll show in the demo. This effect is given mapping the parameters of the `rect` functions with the changing positions of the hand over time. In particular, two variables $r1$ and $r2$ (one variable per square) are used for controlling the width and height of the squares in the `rect` functions. The $r2$ variable depends from $r1$ (actually $r1$ is related to the square on the left) and $r1$ maps the values representing the hand orientation with

the width and height of the square: decreasing the width will consequently decrease the height of the first square. In addition, $r1$ and $r2$ are also used for creating a vanish effect while a square is decreasing its size, as arguments of the the two `fill` functions in the code.