COMPUTER MUSIC: LANGUAGES AND SYSTEMS

---

# Homework Assignment Report

# "AIR GUITAR"

---


*Students*

Stefano Liera
Marco Bruni
Francesco Panettieri
Youssef Abouelazm


POLITECNICO
MILANO 1863

# Introduction

The system implemented for the *Computer Music: Languages and Systems* assignment involves an **'Air Guitar'**, i.e. an imaginary acoustic guitar. Through the use of the **Kinect** sensor, the vertical movement of the right hand is picked to simulate a guitar stroke in the air, while the left hand picks a single note on a **MIDI** keyboard. The latter, via **Supercollider**, generates the audio synthesis. The **JUCE** software allows modulation of the audio through chorus, compressor, phaser and panner effects. The **GUI** (Graphical User Interface) is created with **Processing**. Through it, the user can change the parameters of the effects and can choose the chord of the note chosen on the MIDI keyboard, so that even those who have no theoretical knowledge of chords can use this system. Major triad, minor triad, major $7^{th}$, minor $7^{th}$, dominant $7^{th}$, half-diminished $7^{th}$, and diminished $7^{th}$ have been realised.

The following is a block diagram of the system architecture.

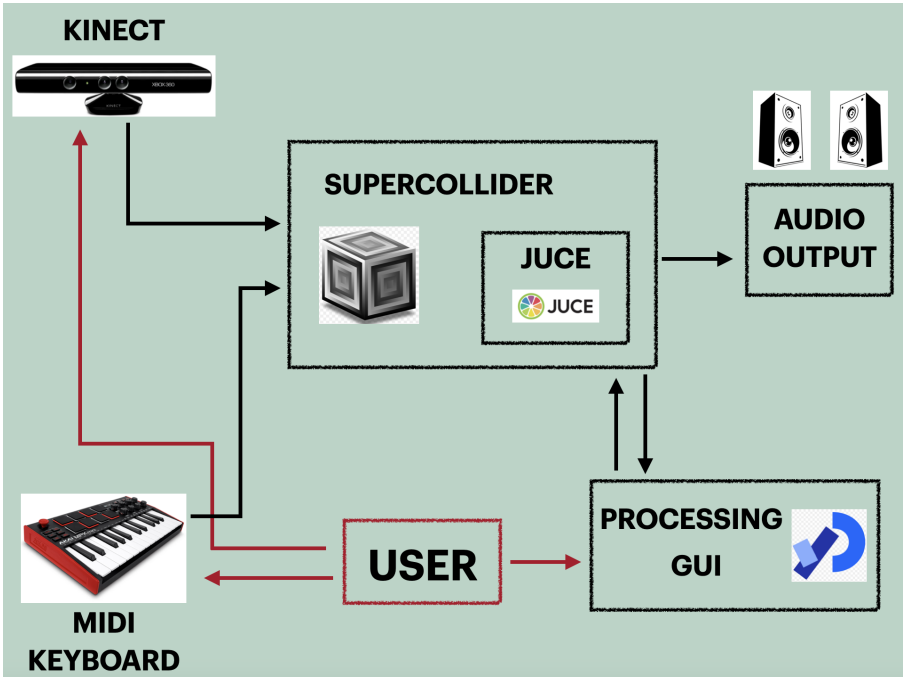


Figure 1: Block Diagram

# Interaction System Unit

In order to capture the user inputs, we utilized the **Kinect** to detect the movement of the right hand to simulate the action of playing a guitar. This sensor, originally developed by Microsoft for the Xbox console, was chosen for its advanced motion detection capabilities and compatibility with various interactive design software. This sensor allows tracking of the human body without holding or wearing special objects.

### Tool Used

For data acquisition and processing, we selected **TouchDesigner**, a powerful visual programming tool for creating real-time interactive content. TouchDesigner integrates smoothly with the Kinect, allowing us to leverage its motion-sensing capabilities and process the data efficiently.

### Installation and Configuration of Kinect

The Kinect was connected and configured following Microsoft's guidelines. This included installing the necessary drivers and ensuring the hardware was functioning correctly.

### Integration with TouchDesigner

In TouchDesigner, we utilized several **CHOP (Channel Operator)** nodes for our setup. Specifically, we used the Kinect CHOP to access skeletal data from the Kinect camera.

This node provided detailed information on the body movements captured by the Kinect.
We set up a system to track the right hand specifically.
This was done by isolating the joint corresponding to the right hand from the skeletal data provided by the Kinect. We focused on the movement along the Y-axis, which represents the vertical movement of the hand. This movement is crucial for simulating the action of strumming a guitar.
The raw movement data was processed within TouchDesigner using a series of CHOPs:

- **Kinect CHOP**: To access and capture skeletal data.
- **Select CHOP**: To isolate the data of the right hand.
- **Math CHOP**: To normalize and process the data for smooth and accurate tracking.
- **OSC Out CHOP**: To send the processed data to SuperCollider.

Real-time visualization of the captured movement was created to allow immediate verification of the system's accuracy and responsiveness.
TouchDesigner provided a robust and flexible platform for processing the data intuitively and efficiently. The system accurately detected the vertical movement of the hand, faithfully replicating the action of playing the guitar. The real-time visualization enabled quick adjustments and optimizations, ensuring a smooth and responsive user experience.

# Computer Music Unit

## SuperCollider

**SuperCollider** is an engine for sound synthesis and algorithmic music composition. It is a real-time sound-based object oriented programming language. It is extensively used for music and acoustic research, algorithmic music, dynamic programming and live audio coding. SuperCollider consists of two components: a server (scsynth) and a client (sclang), which communicate through OSC (OpenSound Control) messages, i.e. using TCP/UDP protocol. The SuperCollider part of the project is the main part that connects everything together. In addition to generating synthesized guitar sound, it receives information from the **Kinect**, MIDI keyboard, and the user interface. Moreover, this is where the effects unit is instantiated.

### Synthesis

The primary synthesis technique used is **Karplus-Strong synthesis**, which involves generating a noise burst to represent a pluck, and using a comb filter to simulate string resonance. This method could be seen as a form of subtractive synthesis, and is effective for creating realistic string instrument sounds in digital audio synthesis. The code is as follows:

```
SynthDef(\string, {arg out=0, freq=440, pan=0, sustain=0.5, amp=0.5;
        var pluck, period, string;
        pluck = PinkNoise.ar(Decay.kr(Impulse.kr(0), 0.05));
        period = freq.reciprocal;
        string = CombL.ar(pluck, period, period, sustain*6);
        string = LeakDC.ar(LPF.ar(Pan2.ar(string, pan), 12000)) * amp;
        DetectSilence.ar(string);
        Out.ar(~myBus, string)
}).add;
```

A burst of pink noise is generated triggered by an impulse, decaying over 0.05 seconds. This is then passed through a comb filter with delay time equal to the reciprocal of the frequency to simulate string resonance. The signal is then panned to two channels, passed through a low pass filter at 12 KHz, and the DC offset is removed. The silence is detected to free the synth when there is no more sound, and the 2 channels are sent to a bus. A second `SynthDef` was used to model the sound of muted strings using enveloped noise.

### Receiving OSC and MIDI messages

OSC messages are received with information from the **GUI**, from the **Kinect**, and triggers from within SuperCollider to generate string sounds.
Messages from the **GUI** are used to adjust parameters for the effects, as well as assign chords to **MIDI** keys.

Handling messages from the **MIDI** input was implemented by using an array to store key presses. Whenever a key is pressed, its value is inserted as the last element in an array. And, when a key is released, its value is removed from the array. This ensures that the last element of the array is always the intended value to play a current chord.

### Strumming Mechanism

The signal from the **Kinect** is received via OSC messages and sent to the `~my_right_hand` bus. There are anonymous functions that detect when the value in the bus passes certain boundaries set for the strings. Whenever a boundary is passed, a trigger message is sent with the string number and whether it is a down or up stroke. This could be used to trigger different sounds based on the stroke direction, although this isn't used in the current state of the project. Here is the code for the functions:

```
6.do({ |i|
        {
                var down, up;
                down = (In.kr(~my_right_hand.index) > ~stringThreshs[i]);
                up = (In.kr(~my_right_hand.index) < ~stringThreshs[i]);
                SendTrig.kr(down,i,0);
                SendTrig.kr(up,i,1);
        }.play.postln;
});
```

These trigger messages are received, and then the corresponding sound is played, based on the current preset of chords, and the signal from the MIDI keyboard.

### Chord Voicings

An important part of sounding like a guitar is to use typical guitar voicings. This was done by listing each possible chord as an array of fret numbers, or `'x'` to resemble a muted string. A function then converted these arrays into arrays of **MIDI** values. These are then converted to frequency values using the built-in `midicps` function when it is time to play a note. The chords are split into different arrays based on type of chord. The available types of chords are: major triad, minor triad, major $7^{\text{th}}$, minor $7^{\text{th}}$, dominant $7^{\text{th}}$, half-diminished $7^{\text{th}}$, and diminished $7^{\text{th}}$. Combining these arrays of chords creates a 3-dimensional array containing all the chords. The default preset assigns diatonic $7^{\text{th}}$ chords in the key of C to the white keys, and secondary dominant chords to the black keys. This, of course, could be adjusted within the **GUI**.

### Effects

A **SuperCollider** extension **VSTPlugin** was used to encapsulate the **JUCE** code inside a module with input and output. By exporting the **JUCE** project as a `vst3` plugin, this allowed the extension to load the plugin into **SuperCollider**. The output from the synths was routed into the plugin, and the output from the plugin was routed to the output bus. The plugin parameters were adjusted based on OSC messages from the **GUI**.

## JUCE

**JUCE** is an open-source cross-platform C++ application framework for creating high quality desktop and mobile applications, including VST, VST3, AU, AUv3, RTAS and AAX audio plug-ins.
By using its project generation tool, Projucer, we've realized useful sound effects. A chorus, panner, compressor and reverb have been implemented.

### Project Settings

First of all, the project settings must be chosen ***Plugin->Basic***. Then, the project needs to be named as desired.
The plugin need to have special formats. Any characteristic doesn't have to be choosen for our final goal, but if one want to test the Plugin separately with AudioPluginHost one has to select the characteristics of Plugin MIDI Input and Plugin MIDI Output.
You select the **.vst3** format because it will then go into Supercollider and this is convenient, also because the plugin can be tested independently on any DAW.

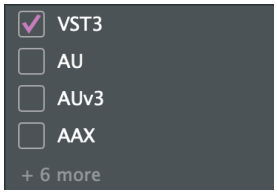


Figure 2: Plugin Formats

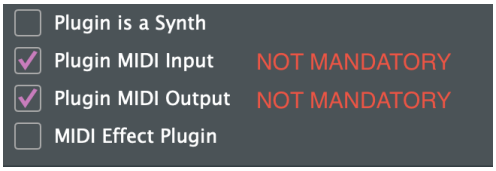


Figure 3: Plugin Characteristics

With these settings the files *Editor.h*, *Editor.cpp*, *Processor.h*, *Processor.cpp* are created by default.
Also by default, all modules that are needed for the type of your project are added. In addition, one must add the module ***juce_dsp***.

### Project Effects

The audio effects that have been implemented are a chorus, panner, compressor and phaser.

- **Chorus**: the DSP chorus modulates the delay of a delay line in order to create sweeping notches in the magnitude frequency response. It features several parameters such as:
  - *Rate*: it is set in Hz and it controls the LFO modulating the chorus delay line. This rate must be lower than 100 Hz.
  - *Depth*: it is the volume of the LFO modulating the chorus delay line (between 0 and 1).
  - *Centre Delay*: of the chorus delay line modulation measured in milliseconds. This delay must be between 1 and 100 ms.
  - *Feedback*: it is the feedback volume (between -1 and 1) of the chorus delay line. Negative values can be used to get specific chorus sounds.
  - *Mix*: it is the amount of dry and wet signal in the output of the chorus (between 0 for full dry and 1 for full wet).

  NOTE: To get classic chorus sounds try to use a centre delay time around 7-8 ms with a low feedback volume and a low depth. This effect can also be used as a flanger with a lower centre delay time and a lot of feedback, and as a vibrato effect if the mix value is 1.
- **Panner**: it is the processor to perform panning operations on stereo buffers. Its parameter is:
  - *Pan*: it controls the current panning value between -1 (full left) and 1 (full right).
- **Compressor**: it offers a range of functions to dynamically manipulate the level of the audio signal with some features:
  - *Threshold*: it sets the level at which the compressor starts working on the signal. Any part of the signal exceeding this threshold level is compressed. If the signal level is below the threshold, no compression is applied. Its unit of measurement is dB.
  - *Ratio*: it determines how much the signal above the threshold is compressed. For example, a compression ratio of 3:1 means that for every 3 dB of input above the threshold, the output will only be 1 dB. A higher ratio indicates stronger compression. All the values must be higher or equal to 1.
  - *Attack*: it sets the speed at which the compressor responds to the signal once it exceeds the threshold. A faster attack in milliseconds means that the compressor starts working faster after the signal exceeds the threshold.

- *Release*: it determines how quickly the compressor stops working after the signal returns below the threshold. A longer release time in milliseconds means that the compressor will continue to work after the signal has returned below the threshold, while a shorter release time means that the compressor stops working more quickly.

- **Phaser**: it modulates first order all-pass filters to create sweeping notches in the magnitude frequency response. This audio effect can be controlled with standard phaser parameters that are:
  - *Rate*: it sets the rate (in Hz) of the LFO modulating the phaser all-pass filters. This rate must be lower than 100 Hz.
  - *Depth*: it sets the volume (between 0 and 1) of the LFO modulating the phaser all-pass filters.
  - *Centre Delay*: i sets the centre frequency (in Hz) of the phaser all-pass filters modulation. It has been set a maximum value of 10000 Hz in order to have an effect over a sufficient frequency range.
  - *Feedback*: it controls the feedback volume (between -1 and 1) of the phaser. Negative can be used to get specific phaser sounds.
  - *Mix*: it is the amount of dry and wet signal in the output of the phaser (between 0 for full dry and 1 for full wet).

**Project Development**

- In the `Processor.h` as a member of the `AudioProcessorValueTreeState` class, it's implemented `apvts`, this variable is used to manage the parameters of an audio plugin centrally and synchronously and also for communication between `Editor` and `Processor`. Then, it is implemented the function `createParameters()` to define plugin parameters. This function returns a `ParameterLayout` object that contains all the parameter definitions.

  Each DSP effect are declarated to implement parameters as `private`, and also the `float lastSampleRate` must be declared.

- In the `Processor.cpp` there is the constructor on which to initialise `apvts` and each effect. The `apvts` is also used to manage the state of an audio processor, in particular to manage parameters that can be controlled by the user interface (e.g. sliders) and automation in a digital audio workstation (DAW).

  In the `prepareToPlay` the `ProcessSpec` has to initialise to use each DSP module. In addition one need to specify `sampleRate`, `maximumBlockSize`, `numChan`. Then each DSP method must be initialised, e.g. for the compressor `compressor.prepare(spec); compressor.reset();`

  In the `processBlock` each parameters of the effects has to be read as a parameter of the AudioProcessValue-TreeState. `juce::dsp::AudioBlock<float> block(buffer)` is used to create an AudioBlock object that references the data of an audio buffer. This is particularly useful in the JUCE DSP library for processing audio efficiently and flexibly. Each parameter was chosen as `float`. Example for the Centre Delay of the Chorus: `float chorusCentreDelay = *apvts.getRawParameterValue ("ID_ChorusCentreDelay")`.

  Then each parameter has to be setted, for example `chorus.setRate(chorusRate);`, and also each effect has to be `process` using `dsp::ProcessContextReplacing<float>(block)`.

  Then in the definition of parameters a vector of unique pointers `(unique_ptr)` to `RangedAudioParameter` has been created, an abstract class for different types for `AudioParameter`. The `parameters.push_back` function is used to add `AudioParameterFloat` to the parameter layout in particular **ID**, **name**, **min value**, **max value**, **default value**. Finally we return two iterators pointing to the first and one to the last element of the vector.

- In the `Editor.h` and in the `Editor.cpp` may also not be written at all, as it only serves for the graphic part. In our code there is a simple JUCE GUI to actually test the parameters in the AudioPluginHost. For the purposes of our project this GUI is not important, it was important in the effects testing phase.

**Interaction between Supercollider**

In Supercollider is imported the **.vst3** plugin.

Due to this implementation, JUCE are receiving messages from Supercollider, therefore in the `Processor.h` is added after the class declaration `private juce::OSCReceiver` and `private juce::OSCReceiver::`

`ListenerWithOSCAddress <juce::OSCReceiver ::MessageLoopCallback>` that handles messages in the context of the JUCE message thread, which is useful for updating the user interface or performing other operations that need to be done on the main thread.

After the constructor, in the `Processor.cpp` specifying the same port (**8000**), `addListener` for each effect and parameter are specifying which messagge with its address is received. At the end of the `Processor.cpp`, `void ProjectAudioProcessor::oscMessageReceived(const juce::OSCMessage& message)` is created for interpreting and logging the received OSC messages for specific addresses, allowing the system to appropriately respond to commands sent via OSC. The function starts by verifying if the received OSC message has exactly one argument and if this argument is of type float. If there is a match, the float value of the message is printed using `std::cout`.

## Graphical Feedback Unit

The GUI has been implemented with **Processing**, with the main goal of providing to the user an easy way to controll and play around the Air Guitar system.

The GUI can be seen as two main parts: one dedicated to the selection of the chords that are played by the keyboard, and one dedicated to controlling the effects.

In the first one, the image of a keyboard is provided to let an easier visualization of what the user are setting on the MIDI keyboard, and the selection of the chords is provided by the use of a droplist for every key, from which the user can select a chord between all the major ones. It's also possible to change the type of that chords by the implementation of another droplist for each key, which determine the type of each chord between *Maj*, *min*, *Maj7*, *min7*, *dom7*, *min7b5*, and *dim7*.

The second part is dedicated to controlling the effects parameters. Several knobs have been implemented in order to controll every effect:

- 5 knobs for the **Chorus** which are controlling the parameters: *Rate*, *Depth*, *Delay*, *Feedback* and *Mix*.
- 5 knobs for the **Phaser** which are controlling the parameters: *Rate*, *Depth*, *Frequency*, *Feedback* and *Mix*.
- 4 knobs for the **Compressor** which are controlling the parameters: *Threshold*, *Ratio*, *Attack* and *Release*.
- 1 knob, bigger with respect to the others, to controll the **Panning** of the audio output.

Each knob can be set by the user as preferred.

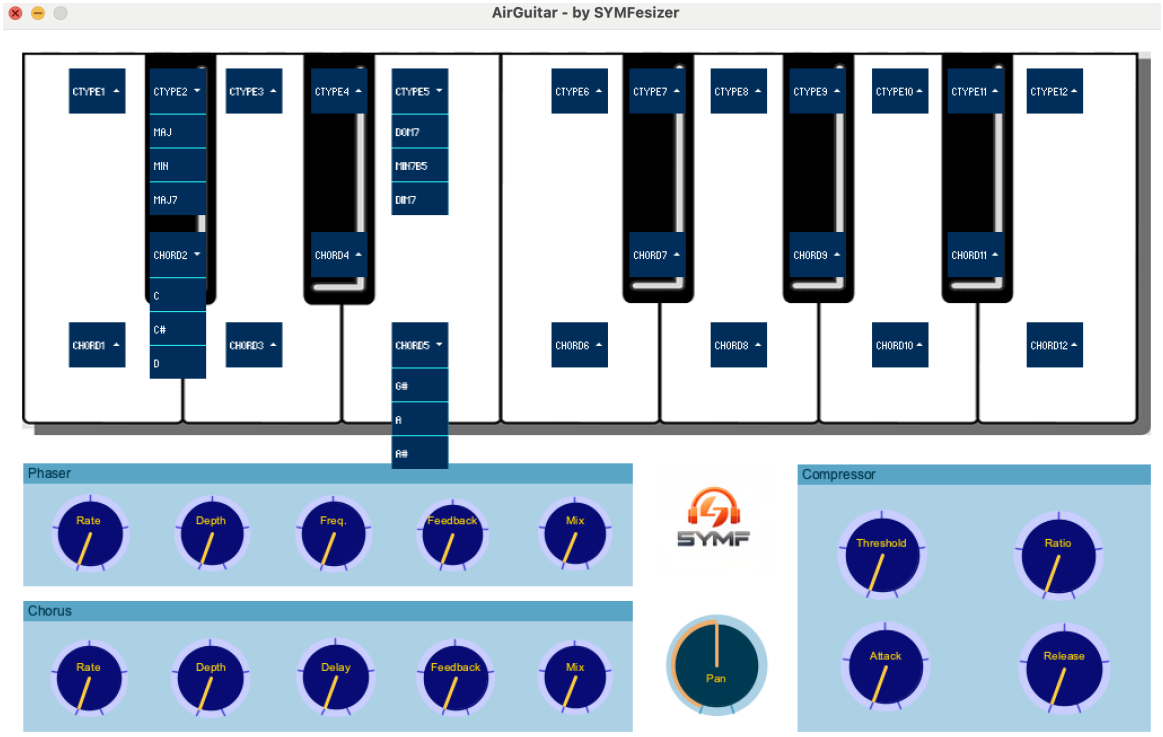


Figure 4: Final GUI

## Further Developments

There are several possible implementations to the project, which for reasons of time we were unable to implement. In terms of sensor usage, the x and z-coordinate could also be used to make the system more complete: for example by detecting a difference in intensity for certain movements or a change in effects for others. Since the Kinect can quote at a maximum distance of 4 meters, one could associate moving away from it with a corresponding fading of sound and vice versa.

Further, other parts of the body such as the neck, fingers, feet could be implemented in the Kinect and its Touch Design software. This would allow greater traceability, perhaps even sound orchestration, and with particular movements a change of effects in real time. Another possibility we thought of, but did not manage to implement, is to create zones in the space captured by the sensor with a different intensity, modulation, or even different chords that would replace the MIDI keyboard.

It would have been interesting to create much more elaborate plugins with JUCE, creating different effects such as WAH-WAH or particular reverberating different rooms, but as the philosopher Seneca said in *The Art of Being Happy and Living Long*: ***"One's strengths are not learned except by experiencing them."***