A USER STUDY ON ELECTROMAGNETIC INTERACTIONS WITH REBUS

Sebastian Cristian Gafencu, Niccolò Perego, Rebecca Superbo

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano
[sebastiancristian.gafencu, niccolo.perego,
rebecca.superbo]@mail.polimi.it

COMPOSITION 1

`stochastic.v1` is a random waveform generator based on a technique designed by Iannis Xenakis. It exploits a set of breakpoints to define a wave shape. The breakpoints are initialized with random duration and amplitude. The waveform is continuously played by interpolating between the breakpoints, and is altered at every iteration. The alteration consists in slightly modifying the breakpoints based on a random walk model. The analog EMF values read from the `GPIO` pins are firstly mapped to a desired range (recommended by Dr. Eleonora Oreggia), and then linked to the duration and amplitude random walk bounds, so that the user controls the evolution of the waveform in time. The values read from the antennas are additionally logged onto the scope for visualization purpose. The generated sound is copied to both the output audio channels. The composition also contains some useful code that makes use of a GUI XY pad for debugging, by simulating the parameter space defined by the antennas.

COMPOSITION 2

Composition two was developed entirely during the workshop by Claude heiland-Allen, aka Mathr.

It is designed to exploit complex numbers in order to produce an interesting and sort of unpredictable interaction. The complex field rotates and decays, thus describing a spiral. When the spiral goes under a certain value, it gets re-triggered making it perpetual.

COMPOSITION 3

`Generatibus` aims at exploring complex interactions with simple timbre, mimicking an arpeggiator.

The user controls the speed and base frequency (fundamental) of the arpeggiator, along with the amplitude of the audio signal. The first 2 are linked to the EMF phase readings, while the volume is driven by the EMF amplitude values.

In the `setup` function, all preparatory methods are called to: initialize two low pass filters, prepare the set of notes to be played in the arpeggio, initialize the wavetable object that enables the oscillator to output a sinusoid, and setup the GUI and oscilloscope tools.

The array of notes to be played in the arpeggio is specified in the global variable `gNoteSet`. Each note, with semitone granularity, can be present or not in the arpeggio either setting its corresponding element in the array as 1 or 0. From `gNoteSet`, the method `convertNoteSet` builds the array `availableNotes`, containing all the specified notes as the increment (number of semitones) to add to the fundamental to output the selected tones.

The `render` function is called in a loop after the `setup` is completed. Its main task is to populate the output audio buffer with the desired sound. To achieve this, the aforementioned sinusoidal wavetable oscillator is used. Since we need to output only a short amount of samples at a time, it is necessary to repeat the same frequency for a certain number of iterations, to obtain a longer note. The duration of the tone depends on the `speed` variable.

Firstly, in `render`, the phase and amplitude readings are gathered from the analog pins `0` and `4` on the Bela boards. If the code is run in "test" mode (helper variable `USING_REBUS == 1`), these values are instead simulated using the GUI, quering the position of the mouse on the x and y coordinates of the screen. Different mapping operations are executed with respect to how the readings are generated.

Then `currBaseFreq`, the variable containing the current base frequency of the arpeggiator, is updated with a linear mapping from `phaseReading` to a frequency between 200 and 1500 Hz. `currBaseFreq` is then passed through the low pass filter `smoothPhase` (cutoff at 5 Hz) to avoid artifacts. A similar process (without the low passing operation) is used to map `phaseReading` to the variable `speed` in a range between 300 and 2500.

The new `currBaseFreq` value is converted to the corresponding MIDI note. This is done because, if it is time to update the currently playing tone `currFrequency`, i.e. `renderCounter` reached a multiple of the value of `speed`, a random note between the available ones in the parametric array `availableNotes` is picked, erasing it from the set. This avoids the same frequency being played again the next iteration. `lastIncrement`, containing the last obsolete tone, is put back in the array to be pickable in the next iterations. The new increment is added to the MIDI note, and `currFrequency` is obtained with the inverse mapping from MIDI notes to Hz. Lastly, the `currentIncrement` is saved in `lastIncrement` for the next iteration.

The value of `amplitude` is obtained mapping the variable `gainReading` to a range of `0` to `1`. As per `currBaseFreq`, `amplitude` is passed through the low pass filter `smoothAmp` (cutoff at `15` Hz) to avoid artifacts.

Finally, `currFrequency` is set to be the frequency of `gOsc`, the sinusoidal wavetable oscillator object. The method `process` of `gOsc`, produces the array `out`, then scaled by `amplitude` and by a factor of `0.2`.

`out` is written to all audio channels with `audioWrite` and then `gScope` is set to log both the EMF signals and the audio output signal.

At the end of the cycle, the `renderCounter` variable, keeping track of how much times the current frequency has been played, is incremented.

COMPOSITION 4

`Rhythmbus` attempts to recreate a drum rhythm machine exploiting the interaction with the space. The user's movement affects velocity and timbre of rhythmic samples and also triggers two ghost buttons, i.e. invisible points mapped to be circumscribed between the REBUS antennas. These buttons turn on and off different features of the drum machine:

- Ghost button 1 : triggers the samples to play normally or backwards, depending on the amplitude value
- Ghost button 2 : triggers a fill pattern, depending on both the timbre and velocity values

The `render.cpp` file performs all sound related operations on the Bela board. First, all the useful variables are declared along with the sensor data of the REBUS.

In the `setup` function some useful components are initialized:

- The two `GPIO` pins that communicate with the REBUS antennas
- The `audioOutputBuffer` that will store the drum samples
- The `gScope` oscilloscope for the visual output

In the `render` function the core audio operations are found:

- The values from the antennas are read from the `GPIO` pins and mapped as amplitude and timbre values
- `If` clauses that determine the triggering of ghosts buttons
- Counting of samples to determine which one to play next
- The `for` loop that writes in the audio output channel the desired sound
- `gScope` command to visualize the output in real time

The `startPlayingDrum` function implements if clauses and for loop to play a particular drum sound given by drum index.
The `startNextEvent` function starts to play the next event in the pattern.
The `eventContainsDrum` function checks whether the given event contains the given drum sound.
The `cleanup` function cleans up the `audioOutputBuffer` once the code has stopped running.