OUNK - AN AUDIO SCRIPTING ENVIRONMENT FOR SIGNAL PROCESSING AND MUSIC COMPOSITION

Olivier Bélanger
Laboratoire informatique acoustique et musique (LIAM)
Faculté de musique, Université de Montréal
Centre for interdisciplinary research in music and media technology (CIRMMT)
Montréal, Québec, Canada

ABSTRACT

In this paper, an audio scripting environment, called *Ounk* [1] is presented. Ounk uses Python [3] as a programming language and Csound [2] as an audio engine. It can be used for a variety of tasks such as composing, sound design, live performances, developing signal processing chains and much more. In addition to its powerful synthesis and sampling capabilities, it supports MIDI, Open Sound Control and Human Interface Device protocols. An interface is provided to facilitate writing of scripts and managing complex musical projects. The Ounk environment includes many functions to easily create loops, sequencers, midi synthesizers and more. Ounk combines the large and varied set of Csound unit generators and the power of the Python programming language. Since its components are all multi-platform, Ounk runs on OS X, Windows and Linux systems.

1. INTRODUCTION

Many computer music systems, newer than Csound, are available today to develop signal processing chains. Some provide graphical patch editors (e.g. Max/MSP, PD) and some others use more advanced techniques of software engineering, with their own language for programming modules (e.g. SuperCollider, ChucK). The main reasons for choosing Csound as an audio engine are that it runs on almost every platforms, has a huge unit generators library, features very good sound quality and can be run independently of any interface, making it easy to use as a library inside any programming language. The development of a scripting environment for Csound was motivated by the need for algorithms to rapidly generate musical and sound descriptors without having to write Csound orchestras and scores. The script takes care of both, as well as of all flags, headers and syntactic particularities of Csound. Another advantage of this environment is the use of Python to write scripts. Python is very easy to learn, is used for all kinds of applications, and comes with a large high-level functions library. Learning programming with Python can be helpful, not just for musical purposes, but for a lot of programming tasks. In this context, homemade classes and functions can be written and imported into scripts to give more flexibility when designing musical projects.

2. OUNK STRUCTURE

The core of Ounk is a library of functions, called *ounklib*. The Ounk library can also be imported in a Python program outside of the Ounk environment. Functions are classified in various categories : *sources*, *table generation*, *table process*, *controls*, *algorithmic*, *analysis*, *effects*, etc. Each category provides functions to perform a specific kind of action. Ounklib functions are used inside a common Python script and compile everything needed to generate the desired sound processing code in a .csd file (Csound unified file format). The function **startCsound** calls Csound to run the .csd file. All functions can output mono, stereo, quad or octophonic signals by selecting the number of channels in the function **setAudioAttributes**. Although Ounk comes with an interface, it is also very easy to run a script from a command-line. One simply needs to place the Ounk resources folder in the current working directory. The interface gives more flexibility to the user with shortcuts, documentation window, the possibility to run multiple scripts at the same time and more.

In order to give full control over music performance, Ounk supports many protocols : MIDI, OSC and HID. MIDI is perhaps the most common for musicians. It allows manipulation of MIDI synthesizers created in the script and all support for live transformation with controllers. Open Sound Control offers a simple and efficient way to communicate between Csound and Python or any externals software that support OSC protocols. Some hardware devices are starting to offer native OSC support as well. Human Interface Devices, commonly a USB game device, can be retrieved inside Python, and Ounk provides functions to easily map their values and use them as musical controllers.

3. OUNK INTERFACE

The interface is built with wxPython [4], a multiplatform graphical toolkit based on wxWidgets that takes native themes of the system on which it is running.

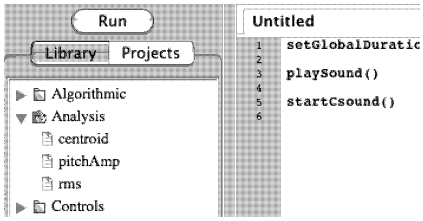

Figure 1. Control and Functions tree panel.

The interface is divided in four panels. The first one from the left (see Figure 1) is the command panel with a single button to run scripts and a tree that contains all functions provided by Ounk. If a function is selected in the tree, its documentation page is automatically displayed in the reference panel on the right. A double-clic on a function will automatically insert it in the text at the position of the cursor. There is a folder with all the examples, in tutorial format, that come with Ounk and a double-clic on any of them will open it in a new window.

The middle panel (see Figure 2) is a script editor. It handles multiple files and allows colorization and autocompletion. Like the multicore rendering capabilities of Ounk, scripts in the editor can run their process in parallel, allowing the distribution of complex musical structures to multiple sections each of which can be started and stopped at any time during the performance.

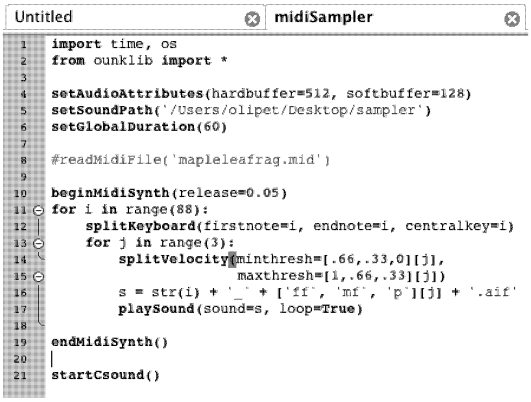

Figure 2. Script editor panel.

The right panel (see Figure 3) shows the online documentation about the function under the cursor location. It explains what the function does and shows its default parameter values.

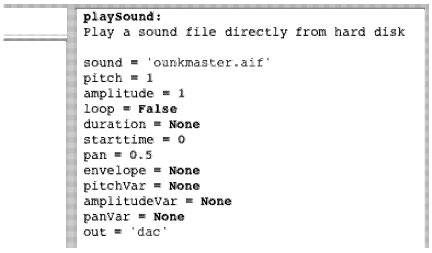

Figure 3. Function documentation panel.

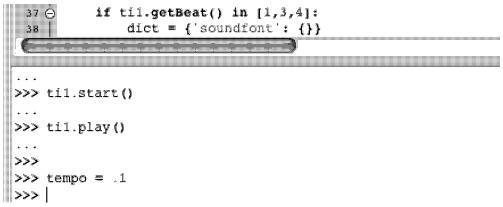

Figure 4. Python interpreter panel.

At the bottom of the window is a Python interpreter console (see Figure 4) where the Python commands are parsed. This is done line by line instead of simply calling Python to run the script, in order to allow direct user interaction with the musical processes during performance. For example, it is possible to create a metronome with a given tempo in the script and then modify the tempo during the performance by assigning a new value in the interpreter. The console is also useful to see what errors occur in the execution of a script.

4. FUNCTIONS STRUCTURE

There are different kinds of functions available to build complex musical projects. Some of these are flags to define a particular kind of instrument structure : midi synthesizers, tap sequencers, loops or instruments controlled by Python's call (see section 5). A group of functions, called *algorithmic*, are designed to generate musical material. Rhythmic and melodic material or Markov chain processes are easily produced with embedded classes and methods.

An overview of the functions provided by the Ounk library can be founded on the development web site [5].

Most functions in the Ounk library are dedicated to handling Csound syntax and creating lines of .csd files. Control values are easily passed from one function to another. The sending function uses a unique name to the parameter **bus**. This unique name can be used by any receiving function through the use of a parameter **xxxVar** (**xxx** is the name of the parameter to be modified). Audio samples can be passed between functions in the same way as control values. The sending function uses a unique name to the parameter **out**. This unique name can be used by any receiving function through the use of a parameter **input**. By default, a function generating audio samples will send to an audio signal converter (dac). The example shown below will play the sound *ounkmaster.aif* for a duration of 10 seconds through a lowpass filter with a little variation of pitch.

```
setGlobalDuration(10)
randomi(bus='pit', mini=.95, maxi=1.05)
playSound('ounkmaster.aif', pitch=1, pitchVar='pit',
          out='snd')
bandpass(input='snd', cutoff=3000)
startCsound()
```

Each function can have its own duration, but if no duration is specified (duration = **None**), it will take, as the default value, the global duration of the script defined with

setGlobalDuration(dur). If this function is omitted, the script plays endlessly. The start time defaults to a value of 0, corresponding to the beginning of the performance.

One the most powerful features of these functions is the management of lists as parameter values. First, the function measures the length of the longest input and then creates as many events as necessary to match. If a shorter list is given to some other parameter, the values will wrap around in this list during the creation of the events. The example below will play 100 sine waves indefinitely with frequencies randomly chosen and an alternating pan.

```
pits = [random.randint(100,900) for i in range(100)]
sine(pitch=pits, amplitude=.05, pan=[0,1])
startCsound()
```

Here is an example of the use of MIDI controllers to modify, in real-time, pitch and amplitude of a sine wave.

```
midiCtl(bus=['pit', 'amp'], ctlnumber=[74,71],
        minscale=.5, maxscale=2)
sine(pitch=500, pitchVar='pit', amplitudeVar='amp')
startCsound()
```

5. INSTRUMENTS

Some part of the code can define a special kind of instrument, to be controlled from different sources. These instruments are interleaved between two function flags, **beginXXX** and **endXXX**, where **XXX** corresponds to a particular instrument. For example, all functions between **beginLoop** and **endLoop** will be played in loop for the duration specified as a parameter to **beginLoop**. This feature allows the creation of very powerful processes with only a few lines of code.

5.1. MIDI synthesizer

A midi synthesizer is a process that will respond when a midi note is sent to Csound. It is defined between the function flags **beginMidiSynth** and **endMidiSynth**. At this time, up to 16 synthesizers, each assigned to a different midi channel, can be defined in one script. Polyphony and overlap are completely handled inside Csound. Volume and pitch bend controllers are always mapped in the synthesizer and are scalable by the user. Other controllers can be used with the **midiSynthCtl** function. It is also possible to get control values from a controller bus defined outside the synthesizer with **midiSynthGetBus**. Inside a midi synthesizer, pitches of instruments are transposed on the basis of a user-defined centralkey.

```
beginMidiSynth(centralkey=60, release=2, pitchbend=2)
freqMod(pitch=[100,50], modulator=.498, pan=[0,1])
pluckedString(pitch=100, amplitude=.5)
endMidiSynth(out='toReverb')
```

Ounk provides functions to split the keyboard into regions (**splitKeyboard**) and to assign different processes to different velocity layers (**splitVelocity**).

5.2. Step sequencer

A Step sequencer is controlled by a metronome and reads one or more rhythm table. When a positive value is triggered in the current table, the sequencer plays whatever is defined inside the functions **beginSequencer** and **endSequencer**. A sequence can be triggered on and off during performance, allowing live mixing of multiple sequences running in parallel. Any specific parameter of any function can be assigned to a given table for dynamically transforming the musical data of a sequence. In the example below, the starting point of a line controlling the modulation index of a FM synthesis is decreased on each step of the sequence.

```
env = genAdrs()
tapTable = genDataTable([1]*16)

# values to assign to a parameter on each tap
var = genDataTable([1,.96,.91,.85,.8,.75,.7,.64,.58,.5,
                    .42,.34,.26,.2,.14,.1])
metro(bus='metro', tempo=120)
beginSequencer(input='metro', table=tapTable)

# modify a parameter on each tap
seqParameterTable('i1', var)

# 'i1' is the parameter to modify
linsegr(bus='index', i1=1, dur1=.1, i2=.1, duration=.2)
freqMod(pitch=100, amplitude=.1, duration=.25,
        envelope=env, index=30, indexVar='index')
endSequencer()
startCsound()
```

5.3. Python instrument

A Python instrument is constructed between functions **beginPythonInst** and **endPythonInst**. It is designed to receive events directly from Python. A project can handle as many Python instruments as desired. Each is assigned to a different voice and can be called independently. A new value can be assigned to any function's parameter on each call, by passing a dictionary to the function **sendEvent**. A dictionary is a special Python data type often called **associative arrays**. Python dictionaries are indexed by keys.

```
# a simple soundfont instrument
beginPythonInst(1)
soundfont('piano.sf2', duration=1)
endPythonInst()

startCsound()

# call note(x) to play a note with control on pitch
def note(pitch):
    dict = {'soundfont': {}}
    dict['soundfont']['midipitch'] = pitch
    sendEvent(1, dict)
```

While the script above is running, function *note* can be called in the interpreter or in a Python loop to play a stream of notes with controls assigned to pitch. This instrument is intended to be used with **pattern** objects. Pattern objects create a Python timer and follow user-defined rhythmic values to send events. See more in section 6.

5.4. Loops

Loops, as the name implies, create looped patterns. Each event, including those created by passing a list as a

parameter's value, plays for its specified duration and repeats for the duration specified for the loop in **beginLoop**. Over the duration of the loop, variations can be applied to modify the duration of individual notes, in order to create legato or staccato effects. A list of two values is passed to specify first and last duration multipliers of the loop. A linear scaling between these two values is applied over the length of the loop. Amplitude can be controlled in the same manner.

```
env = genLineseg([0,10,1,25,.25,100,0])
wave = genWaveform([1,0,.3,0,.2,0,.143])
pits = chord('Cm7', octave=10)

beginLoop(amplitude=[1,.05])
waveform(pitch=pits, table=wave, duration=[.4,.6,.8,1],
         envelope=env)
endLoop()
```

In the script above, each note called by the chord will loop after its duration has finished, creating a subtle rhythmic sequence.

6. ALGORITHMIC

Ounk provides functions to compose algorithmically inside the Python environment and gives full control of the defined Csound instruments (see section 5.3). **Pattern** objects create their own timers and call a function on the basis of one or more rhythmic pattern defined by the user. Inside functions, a Python instrument can be played with the possibility to change all parameters on each call.

There are some functions that execute algorithmic processes, such as **markov** which implements Markov chains of an arbitrary order. Management of chords and scales are very easy with **chord** and **scale** functions. Users can customize the harmonic environment by adding their own chords and scales in the Ounk dictionaries. Rhythmic and melodic generation are handled by functions using controlled random number generators. Random walks, generation of loop segments, drone and jump and repeaters are very helpful in composing algorithmic musical projects.

```
scl = scale('Cm')
rnd = loopseg(25,35)
patterns = [[4,4,2], [4,2,2,1,1], [3,2,2,3], [2,3,2,3]]

def pit(min, max):
    if til.getBeat() == 1 and (til.getBar() % 4) == 0:
        til.changePattern(random.choice(patterns))
    dict = {'soundfont': {}}
    dict['soundfont']['midipitch'] = scl[rnd.next()]
    dict['soundfont']['pan'] = random.randint(0,7)*0.12
    sendEvent(1, dict)

tempo = .2
til = pattern(tempo, pit, [3,3,2,2], 24, 36)
til.start()
til.play()
```

7. MAKING A GRAPHICAL USER INTERFACE

Since Ounk is based on wxPython, unique graphical interfaces are easily created for purposes of live performance control. It is possible to program an interface from scratch directly in the script. However, Ounk provides some functions to make it easier. **beginGUI** is called to create a new window. More than one can be created at the same time. This is where widgets can be packed. Functions like **makeButton** and **makeSlider** create a new widget and assign a function to the widget, responding to control manipulation. The function **endGUI** closes the definition text and shows the created window. The GUI created with the code below is show in Figure 5.

```
frame = beginGUI(size=(260, 290))
playButton = makeToggle(frame, label='play',
                        pos=(90,20), function=onPlay)
makeSlider(frame, label='tr1', mini=-24, maxi=24,
           pos=(50,50), function=s1)
makeSlider(frame, label='tr2', mini=-24, maxi=24,
           pos=(50,120), function=s2)
makeMenu(frame, pos=(90,200), label='Choose...',
         function=handleMenu)
endGUI(frame)
```

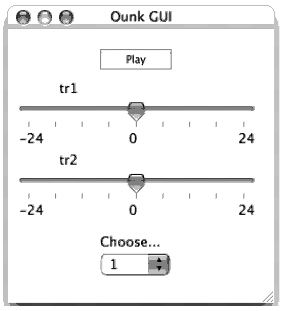

Figure 5. GUI produced by previous code.

8. CONCLUSION

Ounk is an audio scripting environment combining the Csound audio engine and the powerful programming language Python. The main frame of the software is now practically completed. The structure to handle signals and controller streams works very well and will be further developed. What needs to be done at this point is to complete the function library by adding modules for continuous controllers, generative synthesis, signal processing, sound analysis and algorithmic composition. Also, research with composers/musicians is planned to know what exactly is needed for different types of musical work. It is already clear that Ounk can be very helpful to composers and interpreters who want a simple and efficient environment for musical explorations.

9. REFERENCES

[1] http ://code.google.com/p/ounk/

[2] http ://csounds.com/

[3] http ://www.python.org/

[4] http ://wxpython.org/

[5] http ://code.google.com/p/ounk/wiki/Overview