Buchla 700 Recreation Project

Platforms used: Max/MSP, Gen, C++/JUCE

Video presentation: https://youtu.be/JWtb1A3kAoY

Github repository:
JUCE: https://github.com/Rishikeshdaoo/vBuchla-700
Max/MSP: https://github.com/sandcobainer/buchla700-MAX

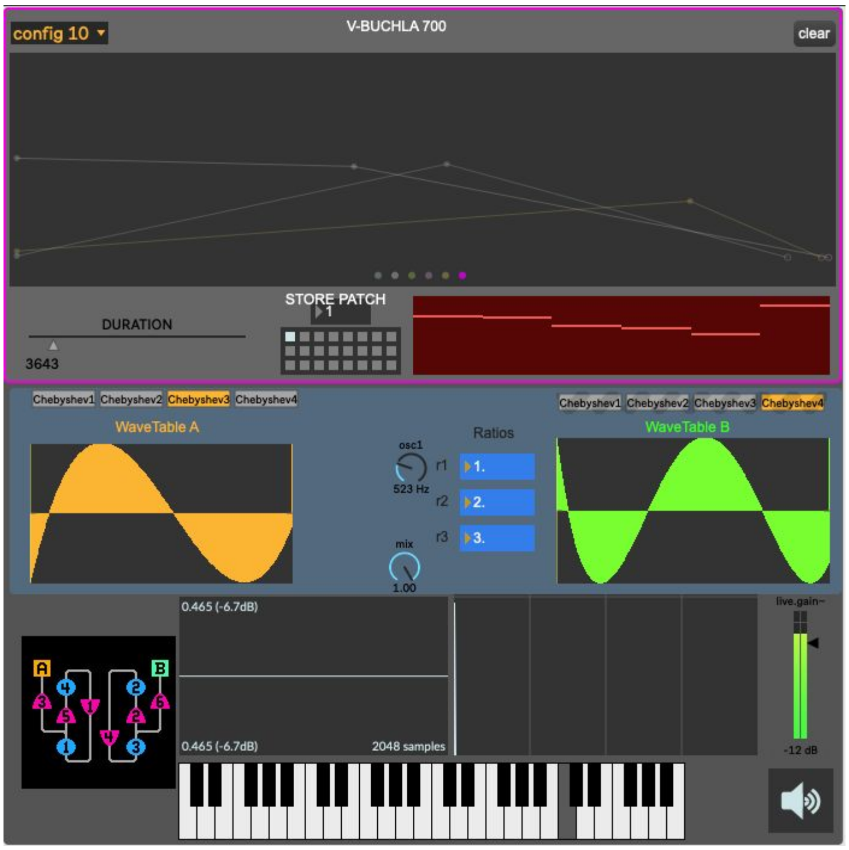
Background:

The architecture of the project is taken from previous sources:
1. Buchla 700 emulator: https://bob.lopatic.de
2. Aaron's Supercollider patch: https://www.youtube.com/watch?v=iXP6akk2FaU
3. randomvoltage.com extracted Buchla config diagrams: http://randomvoltage.com/700/

Approaches:

1. **Max/MSP:**

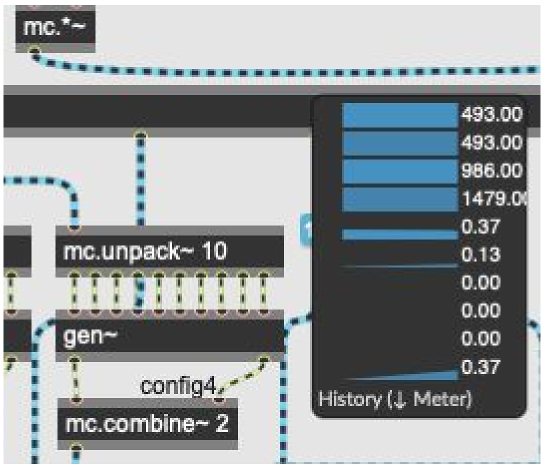
TheMax/MSP implementation uses Max's UI tools like
1. Sliders
2. MIDI roll
3. Function object to draw VCAs
4. Buffer object for waveshaping



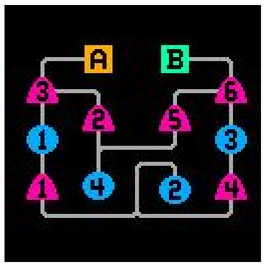
Initially, the user has to select the duration of each note and a config. The user can draw the envelopes for each VCA at the top and select waveshapers. The user is also allowed to draw into the waveshaper buffer (careful gnarly sounds!) and set the osc2,3,4 using relative ratios to osc1. The mix dial allows you to switch between waveshaper A and B.

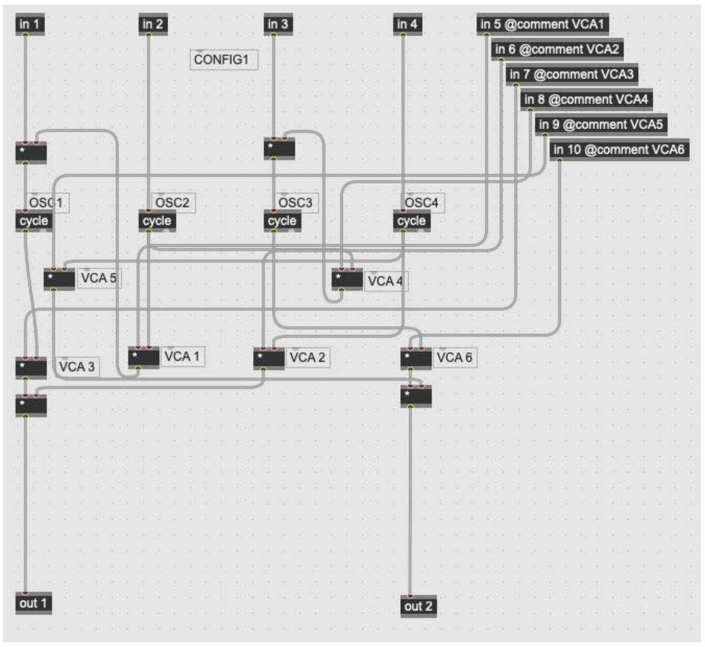
Additionally, a nifty feature is added to save any config and recall it using the 3x8 grid of configs.

The audio is generated using sig~ and line~ for VCOs and VCAs respectively and sent to the gen~ patch for each config. All the config are enclosed in the same patch. Switching between configs causes pops and clicks as the signal is being routed in real time. The image below shows an example of the precalculated oscillator signals (top 4) and envelope (bottom 6) signals being passed into gen~



This bundle of 10 signals is passed into the gen~ patch which is one of the 12 configs provided by Buchla. The gen~ patch can be visualized similar to the traditional Buchla configs due to the visual programming style of Max/MSP. Below is an example of config from Buchla's classic images and an implementation in gen~:





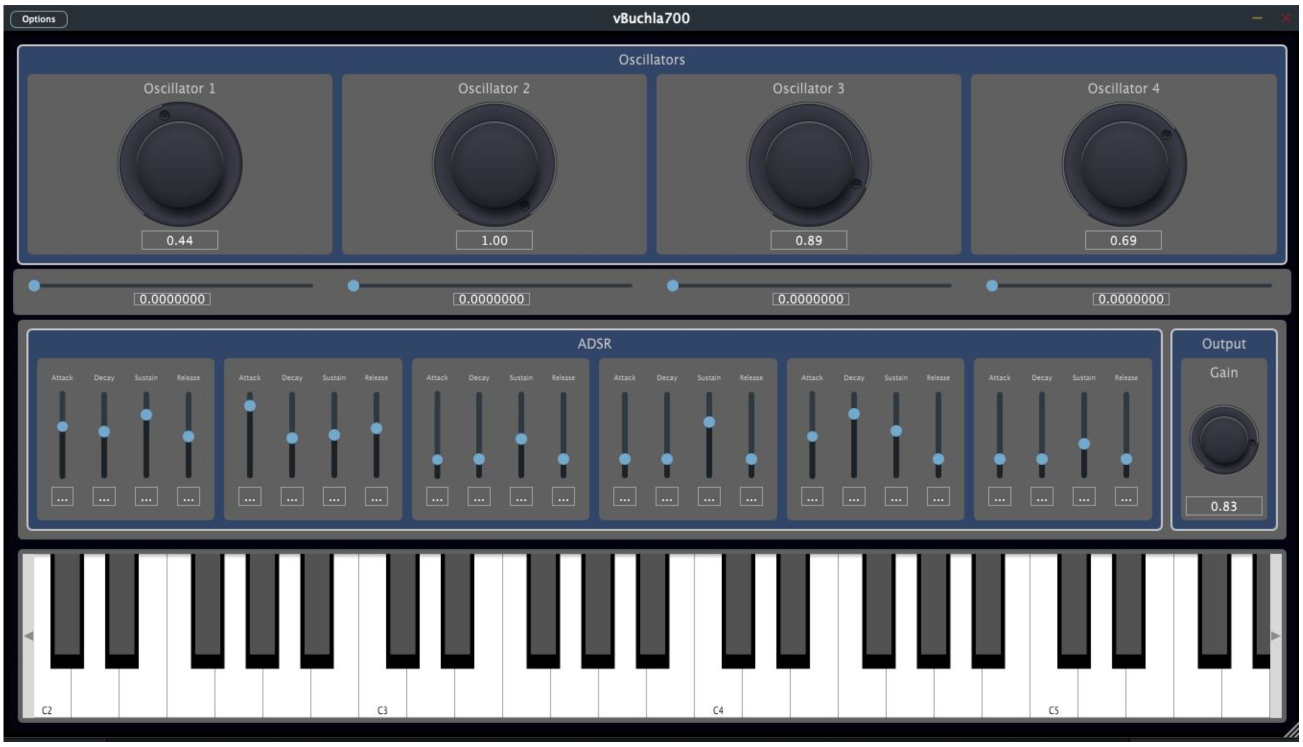
LIMITATIONS:

1. Given, the heavy precalculation and lack of note-on and note-off envelopes, the computation is expensive despite the use of gen~. The application doesn't crash but definitely struggles to run efficiently
2. Clicks and pops! As the signal is being routed in and out as parameters are being changed on the fly, audible clicks and pops are heard especially while changing the configs.
3. Polyphony: Pops are also heard when a note is triggered before the previous note's duration is completed. This can be solved by using Max's poly~ to introduce 16 voices of polyphony (TODO)
4. Waveshapers currently are basic Chebyshev polynomials of 1,2,3 and 4 order and the lookup table creates unwanted high gain on the output signal in some configs. Especially configs that have recursive feedback networks like config3. This is an active problem and requires careful debugging to identify the issue.
5. Building a VST3 out of this gen~ code is outdated as of 2015. Thus, we can use this patch to prototype the network, debug it and build a standalone app OR convert this patch into a Max4Live plugin that can be used as an Ableton Live plugin only

TODO:

1. Polyphony
2. Smooth out clicks and pops
3. Implement more waveshapers
4. Add gain limiter, LPF

JUCE:



The JUCE implementation is created as a VST3 plugin and Standalone application. We used sliders and knobs for setting the different parameters. On the top, there are 4 knobs that set the gain for the 4 oscillators. There are 6 ADSR envelopes (or VCAs) which have sliders for setting the values. The gain knob sets the gain of the output sound from the synthesizer.

JUCE has support for creating a Synthesizer by providing classes for SynthesiserVoice, SynthesiserSound, ADSR, Oscillators, Filters and MIDI message handling.

The following library, that works with JUCE, which provides a DOM based architecture for creating GUI was used - https://github.com/ffAudio/foleys_gui_magic

MIDI is a ubiquitous protocol used for communicating between hardware instruments and plugins/softwares. The data from hardware is encoded in MIDI messages that are sent to the software. Incoming MIDI messages encode a key begin pressed or released as Note-on and Note-off events. This triggers a *startNote* and *stopNote* functions respectively, in the 'Synthesizer Voice' class in JUCE. All the parameters are set in the *startNote* function and the MIDI parameter clearing is done in *stopNote*.

The note-on and note-off events are also used for applying the ADSR envelopes to the sound in JUCE. The ADSR envelope is applied on-the-fly in JUCE, where the envelope value is calculated for each sample and multiplied to the sound. This process is split into stages that link to these two events. The Attack, Decay and Sustain stages are triggered by the note-on event. As soon as a note is pressed, it triggers the note-on event and the Attack stage followed by the Decay stage are started. The sound remains in the Sustain stage until the key is pressed. Once the key is released, the note-off event is triggered and the Release stage of the envelope starts.

The *renderNextBlock* function in the SynthesiserVoice class, is where the main buffer-based audio processing happens. The audio buffers for the Oscillators are created and processed here. The ADSR envelope is applied in real-time and audio is pushed out buffer-by-buffer from here.

The Gen utility in Max gives the option to export the Gen patch as a C++ code. We exported the Gen patch and used it as the config processing unit in the plugin. The *renderNextBlock* should create all the buffers and pass them to the process function from the Gen utility, which would give the config output. This would then fill the output buffer in JUCE.

LIMITATIONS:

The Buchla 700 architecture implements the ADSR (or VCAs) in a different way. The Buchla 700 has a note duration, which can be set in the interface. This duration is used to get the buffer size. Ten buffers (4 oscillators and 6 VCAs) are created depending on the parameters set on the interface and passed on to the config to do the processing.

Creating a complete ADSR envelope as a buffer in JUCE is a bit tricky, since the MIDI protocol does not give any information about the duration of the note. For implementing the

Buchla 700 configs, we need to provide 10 buffers to the algorithm, and it outputs the resulting audio. We hit a limitation with the JUCE approach in this sense.

TODO:

The Buchla 700 implementation is possible in C++ but it would need a different approach by either using a different audio framework or using native C++ for doing the audio processing and using JUCE only for the GUI.