THE DEVELOPMENT OF THE **4KLANG SOFTSYNTH**

BY GOPHER OF ALCATRAZ

zine BEHIND THE SCENE

As some of you might have read in my contributions to ZINE #12, the work on my 4k softsynth "4klang" started right after the release of Sprite-o-mat. The key motivation was to never ever again have to use those gm.dls samples, while still providing a decent sound somehow with a compact representation. Apart

> **THE KEY MOTIVATION WAS TO NEVER EVER AGAIN HAVE TO USE THOSE GM.DLS SAMPLES**

from the fact that the challenge to write a good 4k softsynth was quite tempting from a coder's perspective one particular interesting aspect for me was that I would call myself neither a good musician nor a synth expert. That coupled with the ambition to create one of the best 4k synths around was sort of my initial motivation.

In the rest of this article I'll try to describe the (evolved) concept behind 4klang as well as the pros and cons that come with it.

GATHERING INFORMATION

Before actually starting to think about the code itself I took some time to gather information on how other 4k synths work (know your enemy). I had already written a synth before, some sort of V2 clone, and I actually already tried to "port" it for use in 4k intros once around 2005. But I miserably failed, it was simply too big. So despite knowing what units (oscillators, filters, envelopes, LFO's, delay lines, etc) are in a normal synths and what's basically needed to create sounds I still needed more input, especially in terms of what design and features other 4k synths use.

Some sources of my research and inspiration were:
Stoerfall Ost (http://www.pouet.net/prod.php?which=743)
Fuxplux (http://www.pouet.net/prod.php?which=13016)
V2 (http://www.pouet.net/prod.php?which=15073, http://www.kebby.org)
http://in4k.untergrund.net
Also by the time of writing this article:
Buzzic (http://www.pouet.net/prod.php?which=48898)

And for general algorithms and stuff concerning digital audio processing I can recommend: http://www.musicdsp.org

Some figures I could gather from those and other (human) sources say that 1.3k is an ok size for a 4k synth including a complete song. Taking my gm.dls player and the few sound effects from sprite-o-mat as a reference, all written in C++, I was around 1k (~750 byte code, ~250 byte data). So it was quite obvious that my 4k softsynth had to be somewhat in that size range too, otherwise it would be of no big use if it consumed too much space. This led almost immediately, to the question of which language to use? I knew some people do their 4k synths with C++ and some start with C++ and convert to assembler afterwards and some simply start with assembler code from scratch. I decided to go down the assembler-from-scratch route, simply because it would give me more control over the resulting instructions than any compiler could and because I thought it would be more fun.

Next thing was the structure of the synth itself. In my opinion, you have two options:

- Use a fixed processing layout:

A fixed layout is something like you can find in Buzzic or Fuxplux. Basically utilizing a fixed order and amount of sound generating and processing units. I'd even call kb's V2 a fixed processing layout, though it's not totally true. Each instrument has the same number of units available (3 oscillators, 2 filters,

2 envelopes, 2 LFOs, etc.) and these units are given a predefined order of processing. But you can enable/disable units or flip the processing order at selected stages and especially make use of the flexible modulation routing mechanism allowing almost every unit's parameter to be modified by a modulator.

- Use a variable processing layout:

A variable layout is something like you can find in Stoerfall Ost, Gargaj's .kklangzeug or in TBC's synth. All use a flexible sequence/tree like data structure for their sound definition. This allows you to create complex and varying sounds while still being compact. The only thing missing in .kklangzeug is the possibility to have further sound variation through modulations (e.g. the filter cutoff). But one can overcome this restriction by including hardcoded oscillators into the filter (as pointed out in "ZINE #13: The making of Candystall").

Both types of layout have their pros and cons.

- A fixed layout will have the same number of parameters for each instrument, making it less flexible on the one hand but it may result in better compression due to similarity between instruments and especially because you don't need any additional information about unit relationship and to which unit a parameter belongs, since that is handled in the code. But depending on the amount of units you provide, the code for the fixed processing will grow quite easily. I guess that's one reason why synths using such a layout are mostly very limited in their amount of units and thus in their possibilities and quality of sound. Anyway this is quite easy to code, just make one big function that processes all units/parameters in the desired order and that's it.

- A variable layout is different on the code and data side. You need to make sure you can define arbitrary unit sequences/trees, thus you must store some sort of relationship data for the units (who depends on whom) and find some way to make sure you know which parameters belong to which unit in that sequence/tree. This automatically will increase your data section and additionally will force you to code some sort of virtual machine to process your sequence/tree (at least I don't see any other way of doing it). Also a variable layout can make it harder to define instruments for the musician, since you have no "pattern" you must follow when creating instruments.

After giving it some thought for a while and especially being influenced by the way the synth worked in Stoerfall Ost I decided to go for the variable layout approach with the possibility of having a flexible modulation mechanism as found in V2, because modulations are what really makes for interesting sounds (apart from effects like delay/reverb)

IMPLEMENTATION BASICS

Every (subtractive) synthesizer needs the same basic units:
- waveform generator (sine, saw, square, noise, etc)
- filter (lowpass, highpass, bandpass, etc)
- envelope (attack, decay, sustain, release)
- fx (delay, etc)

And in general each synth creates waveforms at the start of the signal processing pipeline, modifies those and finally outputs them.

TO PUT IT SIMPLY, I WANTED EVERY UNIT TO BE LEAN

So for an instrument definition having a sequence of units doing these different parts seemed natural to me. Also it's perfect for the processing on the VM side, because you simply would need to call one unit after another. The only problem is: how do you handle the case where you need more than one signal at a time, e.g. combining two or more waveforms, or multiplying the final signal with the main envelope of the instrument? One solution would have been to include more than one oscillator, as well as the main envelope in the waveform generator unit (as I know Gargaj and TBC do it that way today). But I didn't want to include the envelope in the waveform generator, since I also planned to have it as a modulation source as well. Nei-

ther did I want to include more than one oscillator in the waveform generator because it would need more parameters to specify their properties which are potentially not all needed for each instrument.

To put it simply, I wanted every unit to be lean; only doing what it is was meant to do, nothing bloated, nor a combination of different functionality. This finally led me to the idea of a signal stack. Each unit in my synth would operate on the stack, giving it the possibility to access more than one signal at a time, while the units themselves are just called in sequence.

Let me show you a simple example of a command sequence operating on the signal stack:

1. envelope (puts the envelope signal on the stack)
2. oscillator (puts the oscillator signal on the stack, on top of the envelope signal)
3. filter (filters the topmost stack signal, oscillator in this case)
4. mul (multiply envelope with filtered oscillator signal, store to envelope signal, remove oscillator signal)

All that was needed to make this flexible were some small units doing arithmetic operations on the signal stack (like add, mul, etc). And since I planned using floating point signals along the way and the FPU itself is using a stack I decided to make use of that instead of manually managing a stack. This of course meant I couldn't ever have more than 8 signals on the stack and going through all instruments in a song that way wouldn't have worked either. So I decided to use the stack only within each instrument definition. Additionally I tried to limit the unit's internal FPU register usage, so that a stack overflow wouldn't occur that easily. In fact I managed to make each unit consume no more than 3 FPU slots, so within an instrument definition it's possible to keep 5 signals at a time. After the signal for an instrument is processed it is stored in the instrument OUT buffer and the stack is cleared.

The above example now looked like this:

1. envelope
2. oscillator
3. filter
4. mul
5. out (store the final signal to the result buffer in the instrument and remove it from stack)

This would now be done for each instrument and finally, after all instruments are done, a global stack would be processed to gather, sum and output the combined signal to the soundbuffer.

An example for the global stack:

1. accumulate (accumulate all instrument signals, put summed signal on stack)
2. out (store to final output buffer, remove signal from stack)

This scheme proved to be straightforward especially considering the VM code. Since each instrument clears the FPU stack in the end the next instrument can follow immediately. Which means for the VM data all instrument instructions can be stored in sequence; even the global instructions (which are effectively another instrument stack).

For even further sound control I included an additional AUX buffer in the instrument and gave the out unit 2 gain parameters to adjust the amount of the signal that should be fed in the respective buffer. The accumulation unit was then modified to be able to select which of the buffer signals to collect and so allowing it to process those 2 signals individually. That's quite handy to specify which instruments should be processed by a global delay line or reverb and is similar to the way it's done in V2.

The global stack including a global delay:

1. accumulate (out) (accumulate all instrument out signals, put summed signal on stack)
2. accumulate (aux) (accumulate all instrument aux signals,

put summed signal on stack)

3. delay (delay effect on the aux signal)
4. add (sum out and aux signals)
5. out (store to final output buffer, remove signal from stack)

Now getting one sound sample for the whole synth came down to simply sequentially processing one large list of VM instructions!

Speaking of which, a VM instruction in the synth looks like this:

unit opcode (1 byte), unit data (0..* bytes)

Where unit opcode goes from 0 to MAX_UNITS (like envelope = 0, oscillator = 1, ...), the amount of unit data depends on the actual unit, e.g. the filter needs 3 bytes (type, cutoff and resonance), a mul instruction only needs the opcode. Also, as you can see, I decided to store all parameters as bytes which means I had to sacrifice some code to do the parameter mapping to floats for them.

I tried other possibilities like storing truncated floats or storing indices to a float table. In fact the code got a lot smaller, but eventually got overcompensated by the growth of the data section. In "ZINE #13: The making of Candystall" it was stated that using truncated floats was almost as compact as using bytes, it may be that it fits their overall design better than mine. Anyway, I tried those possibilities and decided to keep the byte parameters and remapping code since it always gave me the better results in the end; being slower in execution though. But execution time is not as important as compression ratio and final size of the code, since we're talking about a 4k synth here.

Now instead of only typing numbers in my data section for the VM instructions I created lots of defines which should make it easier to build instruments via code since I had not started with a GUI for the synth yet. The above instrument example then looked like this in my assembler file:

```
GO4K_BEGIN_INSTDEF(String1)
    GO4K_ENV ATTAC(10),DECAY(64),SUSTAIN(103),RELEASE(80),GAIN(50)
    GO4K_VCOFLAGS(PULSE),TRANSPOSE(0),DETUNE(8),COLOR(15),GAIN(57)
    GO4K_VCF VCFTYPE(LOWPASS), FREQUENCY(110), RESONANCE(127)
    GO4K_FMUL
    GO4K_OUT
GO4K_END_INSTDEF
```

Quite readable I think and actually it almost felt like a sound programming language. And just in case you wonder: Yes, the first 4k songs using 4klang were created like that; writing VM code with the use of those macros. Of course the songs were not composed that way. Actually pOWL always composed the songs with my V2 clone in Madtracker and I ported everything by hand to 4klang as good as possible. That's the way we did it until Breakpoint 2008. After that I decided to finally wrap a GUI around the synth core and also to make it a VSTi for easier use in any host application the musician is happy with. I think it was a good decision because I was not eager to also code a buggy tracker and additionally the musician doesn't have to learn a new music tool completely from scratch. It's already enough to understand how to create instruments with the synth plugin itself.

USING PATTERNS IS PRETTY SIMPLE ON THE CODE SIDE AND VERY COMPACT ON THE DATA SIDE

For the music information I decided to do it like most other 4k synths, just using patterns of a certain length (16 is a good value, but it's variable in 4klang) and using a list of pattern indices for each instrument to store the sequence. The only difference may be that I also added the possibility to have variable note lengths through a special "Hold" value in the patterns. Another option would have been to store some sort of delta encoded stream information similar to V2, but I somehow felt it would be

bigger in most cases but maybe I'll give it a try some day. Using patterns is pretty simple on the code side and very compact on the data side. The only disadvantage is that you can only store the information for one note at a time for a certain instrument. This means that chords need additional instruments of the same type with different patterns. But hey, you just cannot have everything; after all we're still talking about 4k music.

And that's pretty much the basic layout and technique behind 4klang.

Side note: Go4k was the project name until I had a chat with Helge/Haujobb the other day and he came up with the idea to call the synth 4klang (thank you for that).

MODULATIONS ARE THE KEY

Now that I had a working synth base which was easily extendable by new units I could move on to the stuff that really makes a good synth. I already mentioned that I wanted to have a flexible modulation mechanism similar to kb's V2, rather than having some hardcoded oscillators here and there. My approach was to implement a "Store" unit which basically just reads the topmost signal from the signal stack and writes that to some defined memory offset in the workspace of any unit in the current stack. So I simply added modulation target members to the unit structures and changed the code of the unit so that it not only would use the parameter value from the VM code but also add the value of its according modulation slot.

I think a small example will make it a bit clearer; I'll show my filter workspace definition here:

```
struc  go4kVCF
;// copies from val struct
        .type           resd   1
        .freq           resd   1
        .res            resd   1
;// work variables
        .low            resd   1
        .high           resd   1
        .band                  resd   1
;// modulation targets
        .fm             resd   1
        .rm             resd   1
        .size
endstruc
```

While processing the filter the first 3 slots contain the mapped byte parameters from the VM instruction. The following 3 slots are the work variables, the current state of the filter. I now added the last 2 slots to the struct and instead of using only "freq" and "res" to process the filter I then used "freq+fm" and "res+rm". So in the case I had a store command somewhere that writes a value in one of those targets it would be added to the base parameter accordingly. If nothing was stored to those targets it just would use the base parameters (since the modulation targets are 0 by default).

This way of storing the topmost signal from the stack to anywhere in the synth had some implicit benefits:

» you can use whatever signal is on the stack at that moment (allowing FM synthesis e.g.)

» you can combine (add, mul, ...) several signals before storing the result to a modulation target (allowing more advanced modulation signals)

» you can do self-modulation (a unit stores its result back to itself)

» you can store to any other instrument stack or even the global stack (cross-modulation or global modulation)

Especially the last point here is a really cool thing. Because it means I could now have instruments modifying other instruments. And even better, I could trigger when the modulation should apply by playing a note for the modifying instrument. And since instruments don't have to produce audible output (they just need to make sure the stack is cleared when they are done) it can be considered some reduced way of automation. Therefore I decided to call such instruments "Control

instruments". The only thing I had to take care of when using modulations was, that if the modulation signal was created e.g. by an own envelope somewhere in the stack, I had to remove that signal after the store command, because otherwise it would have affected the audible signal. That's why I added another unit which simply pops the topmost signal from the signal stack. And yet again I had one more unit basically doing one FPU instruction (fstp st0, st0), so I decided to combine all of those single arithmetic units like pop, add, mul, etc into one arithmetic unit with one parameter specifying the actual operation to perform.

Having that all was already great: only one important part was missing, the infamous delay line. A simple feedback delay was the way to go in the first version of 4klang.

ITERATIONS

Of course 4klang hasn't been unmodified since the first version. Actually I'm counting the 4th major incarnation right now. Some of the added features were done after pOWL made a request; some of them were included because I heard the results of some other synths using that stuff and some of them because I just wanted them to be in the synth. Basically all my 4k's since sprite-o-mat used one new version of 4klang and you can actually hear the progress if you listen to the songs in chronological order. The most important iteration steps or additions were the following:

» adding the possibility to have 2x polyphony per instrument to reduce note on/off clicks (but increasing processing time by a factor of ~1.5)

» extending the delay line to support reverb (mono, as all sounds have been until now)

» splitting the VM instructions into separate streams for opcode and data and a complete rewrite of the VM/player (for better packing ratio)

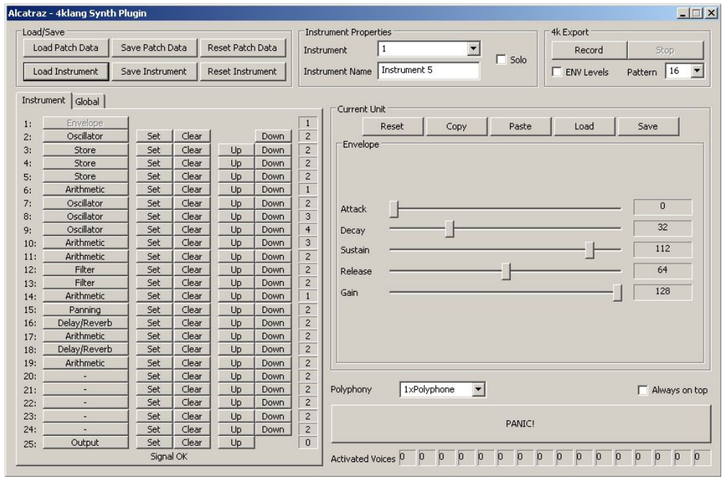
» creating the VSTi plugin so I wouldn't have to manually create/port instruments any longer

» making everything stereo at the end of each instrument, accomplished with a "Panning" unit before the output. (after I realized how much stereo adds to the richness of sound)

» extending the delay line to enable Karplus-Strong like plucked string sounds

And though 4klang was step by step extended by features it was somehow possible to increase the packing ratio and decrease the overall file size in the end. One thing that additionally helped was to wrap new features (and other parts of the code) in preprocessor define blocks, so that you can easily exclude code parts you don't need for a certain song.

The current version which includes the VSTi plugin is something I'd consider to be a final version. As with each tool it takes some time to make full use of its potential, so I don't see the urge to add new stuff at the moment, since the VSTi has been available to our musicians for only some months now. Nevertheless I will for sure include more stuff if it's needed or requested for some production (e.g. a compressor unit which isn't included by now), and now and then try to cut off even more bytes.



CONCLUSION

The weak part of 4klang is its execution time. There are two main reasons for this. First is the parameter mapping which is done each time a unit is processed. The second point is that for each sound sample the complete VM instruction sequence will be processed, resulting in many function calls per sample,

creating quite some call overhead. Gargaj and TBC use another approach I think, they process one complete buffer with all samples for each unit, resulting in very few calls needed overall. But I don't see a way of integrating that concept in 4klang, since it clashes with the modulation concept (especially with control instruments and self-modulation). Also the polyphony issue is nothing I really found a good solution for up to now. At the moment when the synth polyphony is set to 2 I simply have 2 voices per instrument which both are always processed. The player just inserts notes in one of the two voices (alternating) and then processes 2 voices per instrument, which just doubles the work for instruments also for those which don't need polyphony. I added some early outs in the units where possible, but still it increases execution or pre-calculation time a lot. But you get well along most of the time with 1x polyphony anyway.

Now to finally give you some numbers about the size, these are the code/data statistics of our NVISON 4k kevinspacy which used basically the newest version of 4klang.

Sound Init Code: 35 bytes
Sound Init Data: 25 bytes
Synth Code: 730 bytes
Synth Data: 50 bytes
Pattern Data: 230 bytes
Instrument Data: 250 bytes

———————

Sum: 1320 bytes

So a total of 1320 compressed bytes were needed to have 2:30 minutes of sound in that intro including sound init. Of course the synth only used the really necessary units and modulation possibilities, including all the features the synth code would be ~860 bytes instead, so 1450 bytes total. But I doubt we'll ever manage to include all options in one song, so that's only the theoretical upper bound. All of our 4k's since sprite-o-mat was utilizing 4klang in one of its versions and each consumed roughly around 750 bytes for the synth code. So I think this is a good empirical value of what to expect.

Comparing those sizes above with my gm.dls player from sprite-o-mat I would say it's a success. The size of the code is almost equal, only 4klang needs more space for the data. But that was something I was expecting anyway, and additionally the song from sprite-o-mat was really simple in structure, used very few patterns and is almost 1 minute shorter than the kevinspacy soundtrack.

So all in all 4klang is roughly consuming those initially mentioned 1.3k for a 4k synth. Keeping it within this limit is one of the main tasks for the musician, since it heavily depends on the amount and complexity of the instruments and the repetitiveness of the song. The more similar the instruments or the song patterns are, the better the packing ratio will be, so in the end the main problem will always be to compose a catchy song with cool instruments which compresses like hell.

IN THE END THE MAIN PROBLEM WILL ALWAYS BE TO COMPOSE A CATCHY SONG WITH COOL INSTRUMENTS WHICH COMPRESSES LIKE HELL

And that's one reason why I chose to make the GUI for 4klang just the way it is now. It's not hiding any complexity from the musician, which makes it a bit more "coder-esque" to define instruments, but on the other hand also shows the musician what his patch data will look like in the executable later on and what units he actually included as well as their parameters.

Providing any form of building blocks, which would hide the complexity of what's going on underneath would only lead to the musician often/always using those things without thinking about what he actually wants to achieve and how to achieve it with the available base units. But hey, why should only the coder have to think about getting things small.

FINAL WORDS

The work on 4klang up to its current state took over a year and I must say I really enjoyed every minute, even the hard ones. And without the help and efforts of pOWL it would certainly not be what it is now (one feature that would be missing is the “Panic” button!). But since it’s in a state now finally where I can dare to unleash it to the world, you can find the complete 4klang package (VSTi + example instruments/songs, example c++ project) here in ZINE #14.

If you have read this article until here I hope you already got a good overview on how things work, but I still recommend having a look at the readme.txt and especially the examples to get a feeling for it.

And that’s about it for this article. Thanks for reading, have fun trying out the VSTi, creating songs and perhaps using it in a 4k. I’m looking forward to it.

For further questions, discussion or feedback: do feel free to drop me a mail (gopherAThazard-designsDOTde) or catch me on IRC.

P.S.: Quote from ZINE #13: The making of Candystall: “Letting tone-deaf coders take control over the development of a synthesizer is a definite recipe for disaster.”

Actually it seemed to work quite well ;)