Develop

Insight:Inspiration:Networking

COCOON

Develop:Brighton 2024

Jakob Schmid

Geometric Interactive

Who am I?

Computer scientist from Aalborg University, Denmark

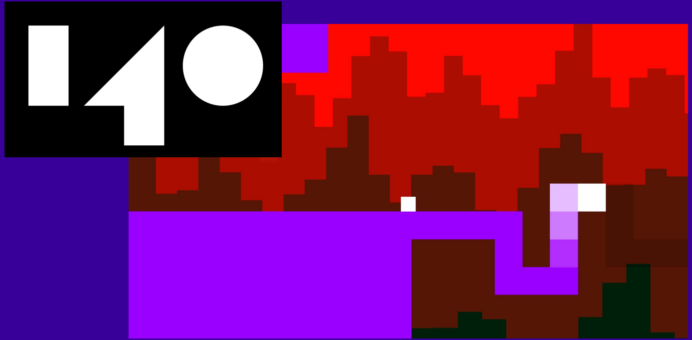
16 years experience in game development

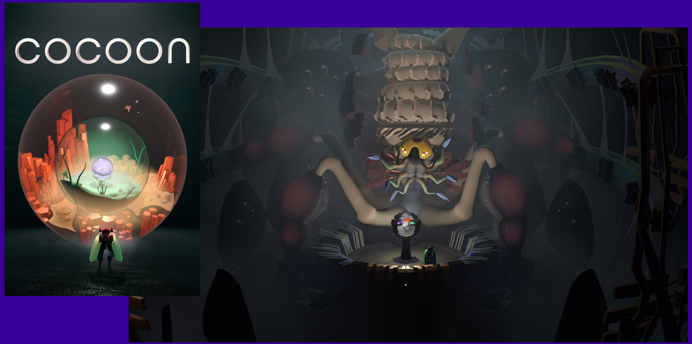
Audio programmer on Playdead's INSIDE

Co-founder, audio director, composer, programmer at Geometric Interactive

(10-person Copenhagen studio)

Studied music and created electronic music since the late 1980s *(SoundTracker and forward)*





What is COCOON?

A puzzle adventure game by

Geometric Interactive

Director:

Jeppe Carlsen

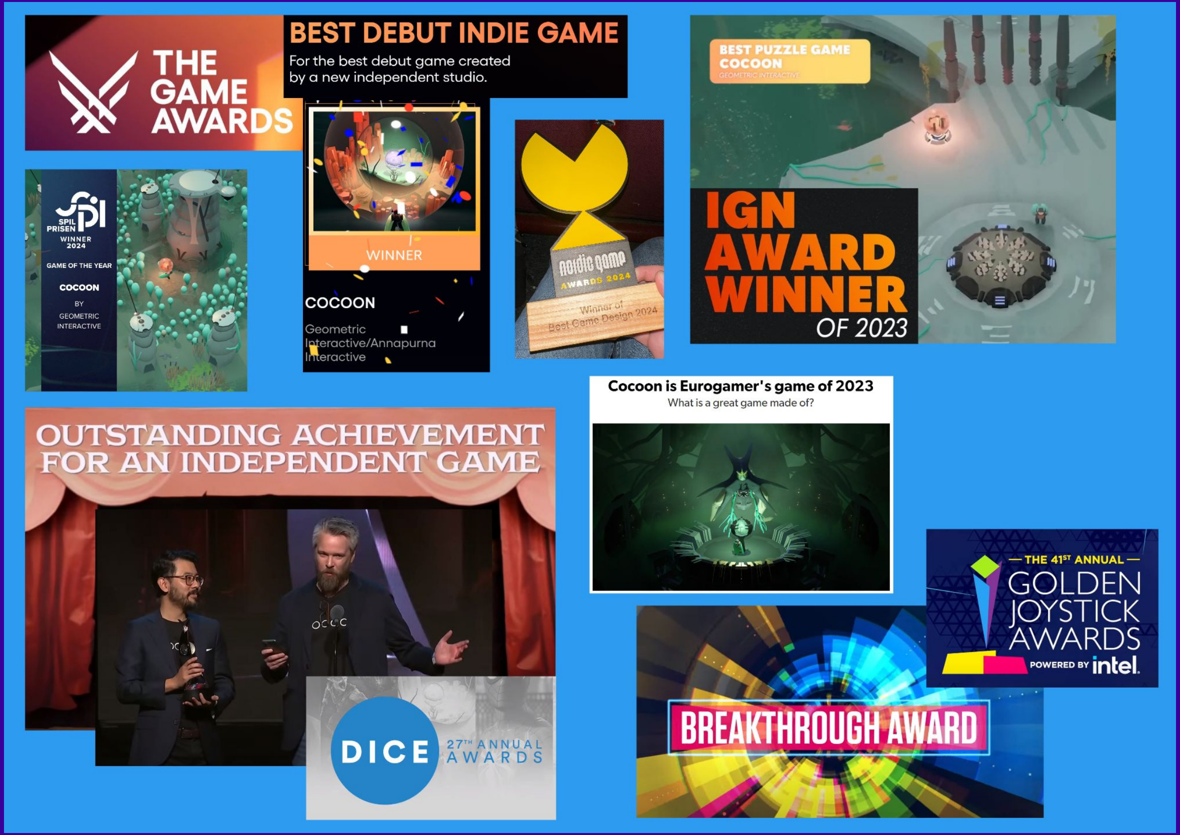
Art director:

Erwin Kho

Production time: 6.5 years

Play time: ~ 5 hours

Some People Seem to Like It



COCOON Audio Team

Audio direction / music:

Jakob Schmid

Sound design:

Julian Lentz
Mikkel Anttila

Topics

Audio concepts and production

The COCOON Instruments

Composing with Plugins

Plugin Implementation

Closing Thoughts

Audio Concepts and Production

Music Concept

Sound Design Concept

Artistic Framework

Real-time Synthesis

Music Concept

Pre-composed vignettes for the big moments

Real-time synthesized ambient music for puzzle gameplay

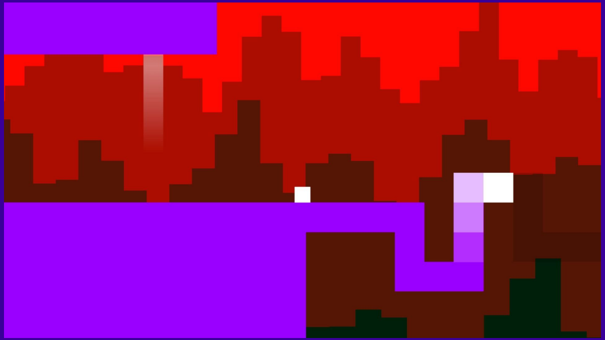
Big moment: Vignette

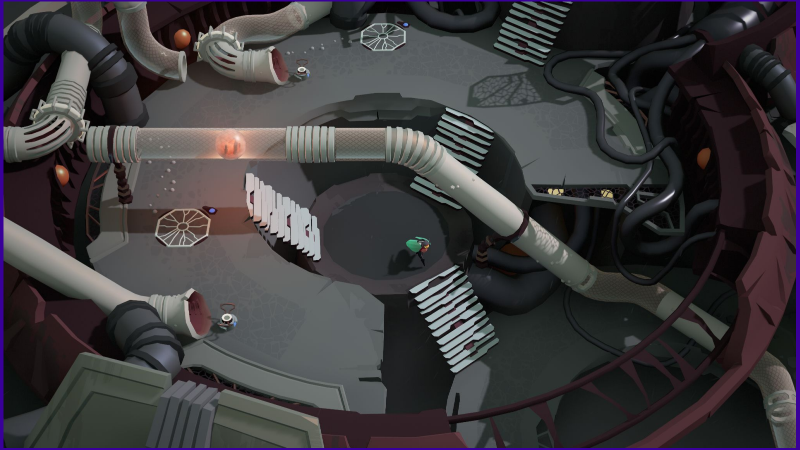
Puzzle gameplay: synthesized ambient music

Sound Design Concept

Synthetic sound design - no recorded sound!

- Fits aesthetics of synthesized ambient music
- Fits art style: artificial but alive
- Familiar process from ‘140’



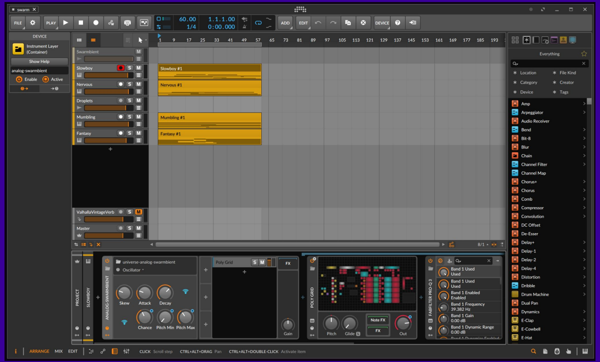


Music Software

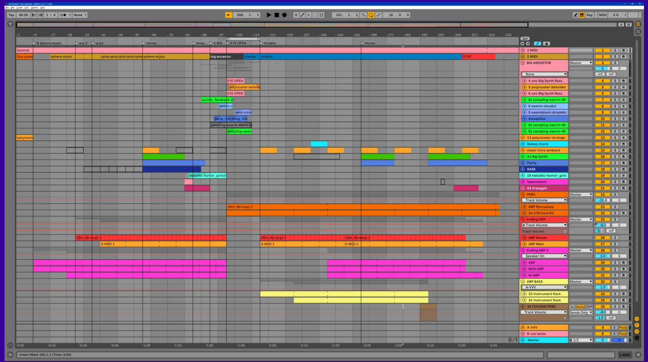
Bitwig Studio and Ableton Live was used for music production and sound design

Ableton Live designing sounds, especially the unique Corpus plugin

Bitwig Studio developing new synthesizers, sophisticated automation



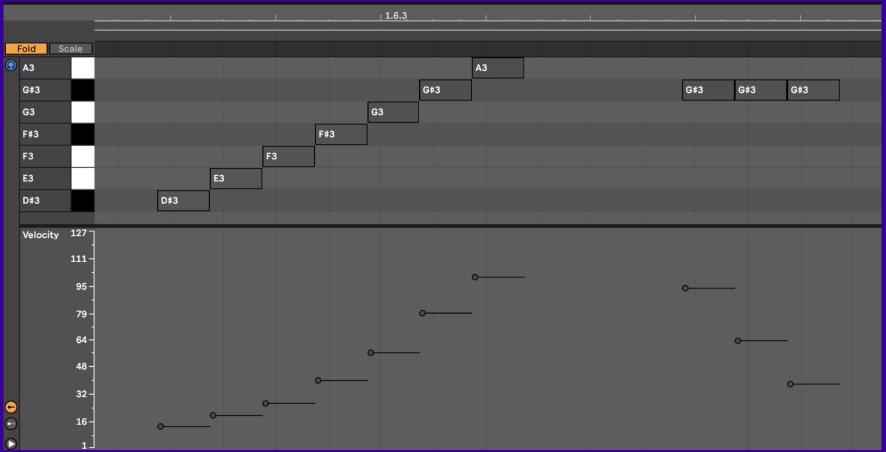
Bitwig Studio 5

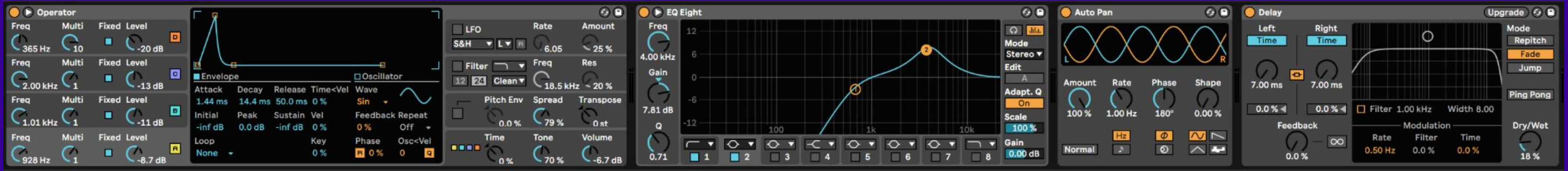


Ableton Live 11

Synthetic Sound Design Experiments

Frogs, footsteps, portals





▶ frog

Artistic Framework

Real-time synthesized ambient music for puzzle gameplay

Pre-rendered synthetic vignettes for moments

Pre-rendered synthetic sound design

Why the Constraints?

Creating an artistic framework with strict constraints is helpful to:

Avoid paralysis from too many options

Focus early work during the infancy of the project

Create coherence in the final work

Find references e.g. "Synthesized music without sequencer":

1970s New Age music

Tangerine Dream, Vangelis, Jean-Michel Jarre

Why Pursue Real-time Synthesis?

Real-time synthesis has interesting benefits:

Loop free during ‘thinking breaks’

Unique soundtracks for each player

Reactive music can react to game events, in terms of notes, timbre, effects

Tiny Ambient music for COCOON takes up 5 MB on disk in total (for a 5 hour game)

Why Pursue Real-time Synthesis?

Even more important reason?

I love designing and writing music systems!

Music Systems

Previous professional projects:

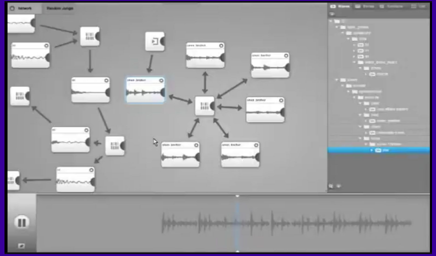
Lost Empire: Immortals -
Dynamic stem mixing system for a 4X game

Audioflow -
Graph-based game music middleware

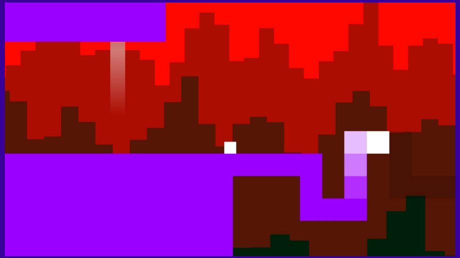
140 -
Music systems for Jeppe Carlsen's music platformer

Rytmos -
DSP plugins for Floppy Club's puzzle game

Lost Empire: Immortals (2008)



Audioflow (2010)



140 (2013)

Rytmos (2023)

Hobby Projects

Acorn Electron (~BBC Micro) one-channel music player

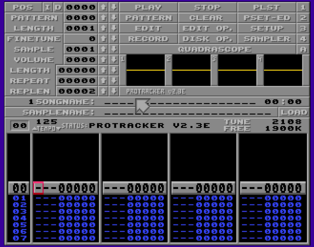
Amiga modified ProTracker replayer to 'mutate' samples during playback

Pico-8 AlgoTracker: 3-track sample-/synthesis tracker

Sega Mega Drive/Genesis music routine (WIP)

Defender arcade machine sound board emulator

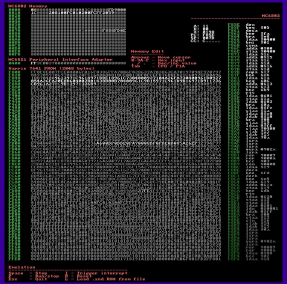

Acorn Electron



ProTracker (Amiga)

AlgoTracker (Pico-8)

AlgoTracker (Mega Drive)


DefendEmu

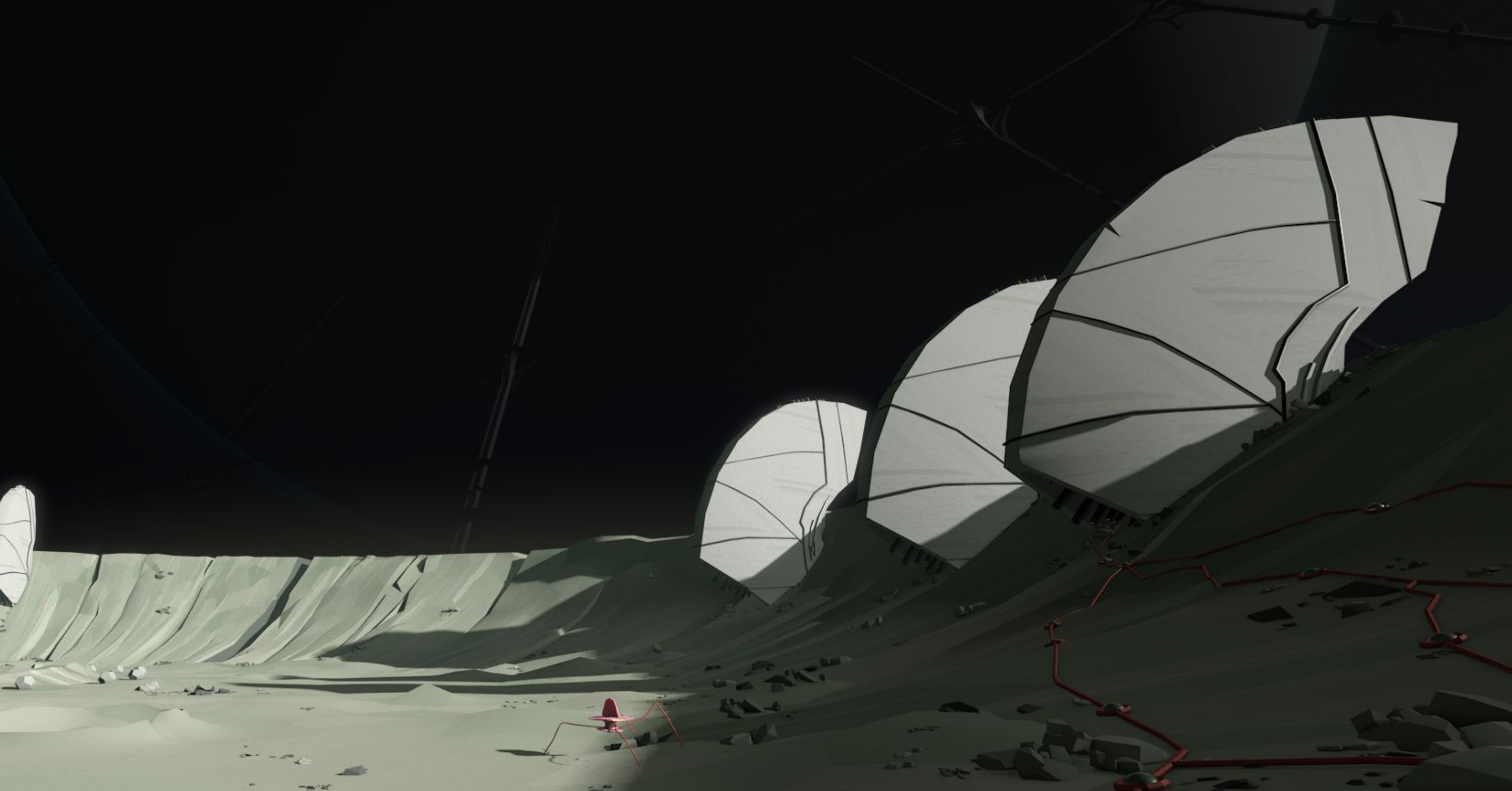
The COCOON Instruments

BOB

K88

Modnet

Weather



COCOON Instruments

BOB

Monophonic subtractive synthesizer

Arpeggiator to generate notes (because FMOD Studio doesn't have MIDI)

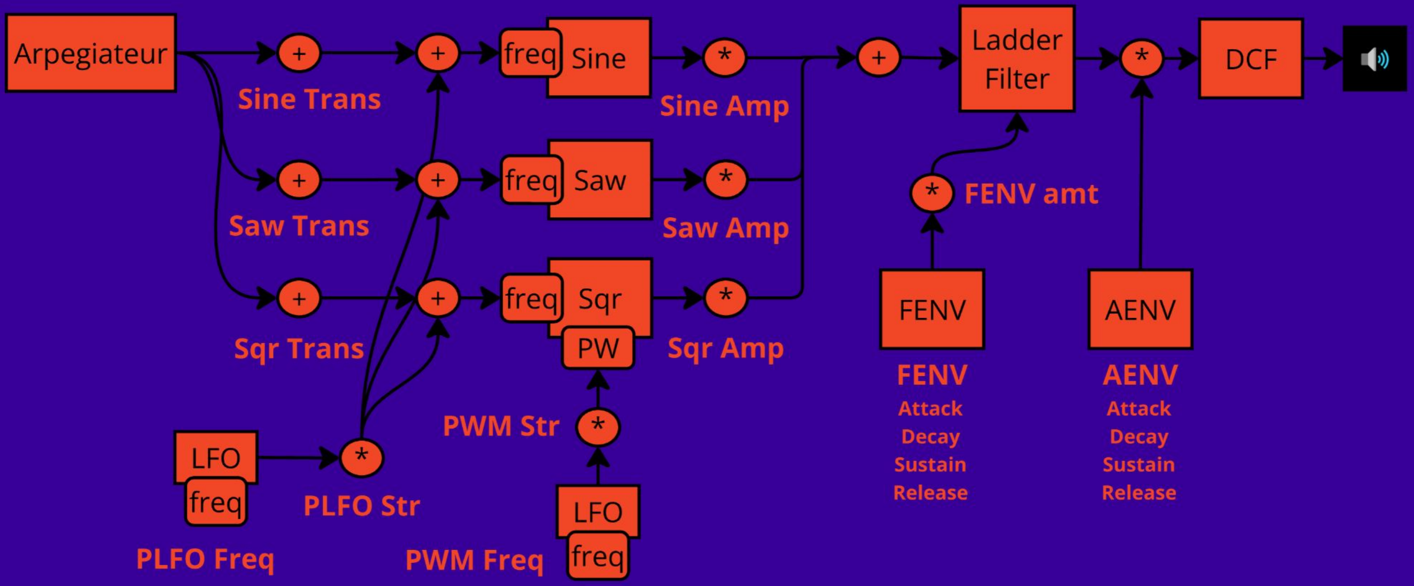
Three oscillators square, saw, and sine, individually adjustable pitch and amplitude

Pulse-width modulation of square wave (also, vibrato)

Ladder filter for resonant filtering

Amplitude and Filter Envelopes

BOB Structure

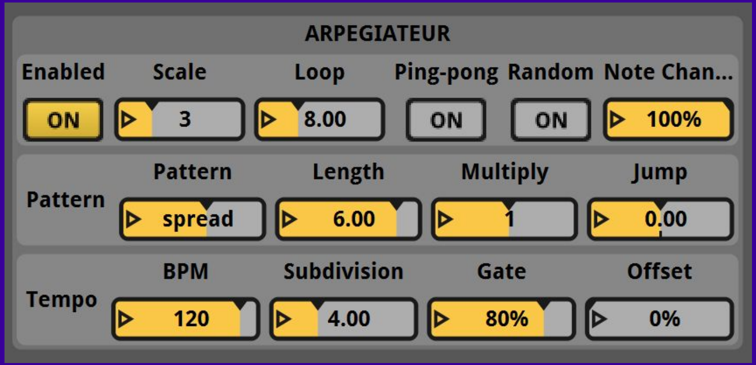


Arpeggiator

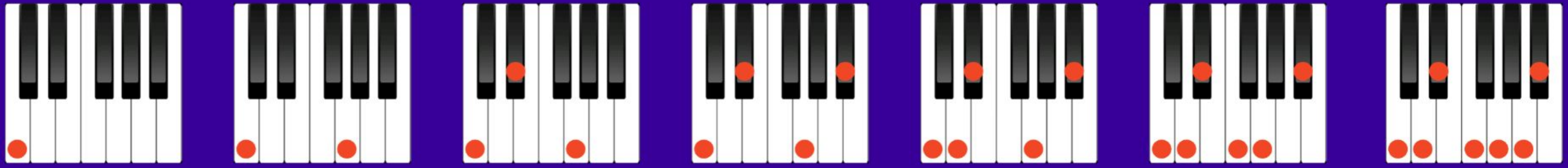
Arpeggiator generates notes the only way BOB can play anything in COCOON

More flexible than usable hard to use for anyone but me

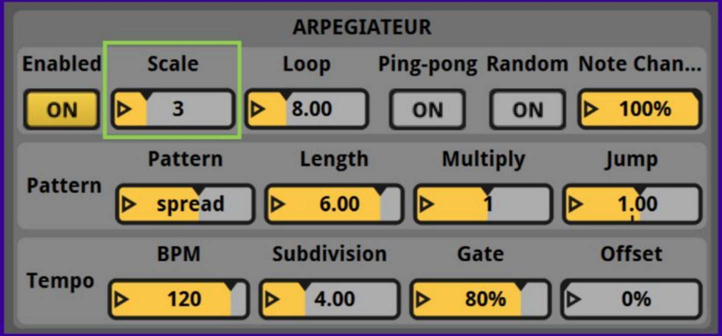
Named 'Arpegiateur' after Jean-Michel Jarre's 1982 track



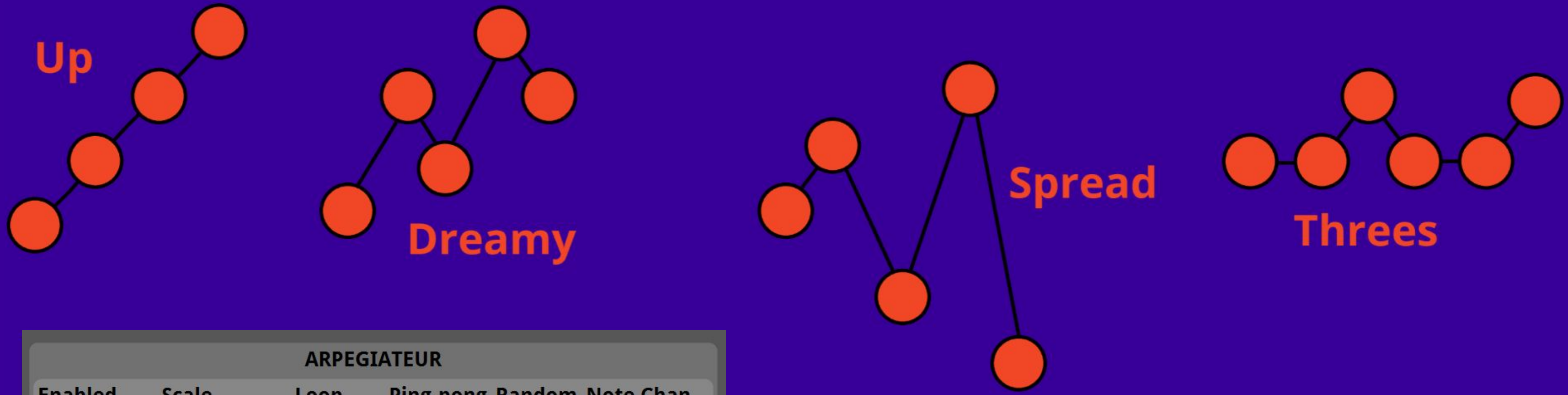
Arpeggiator: Incremental Scale Control

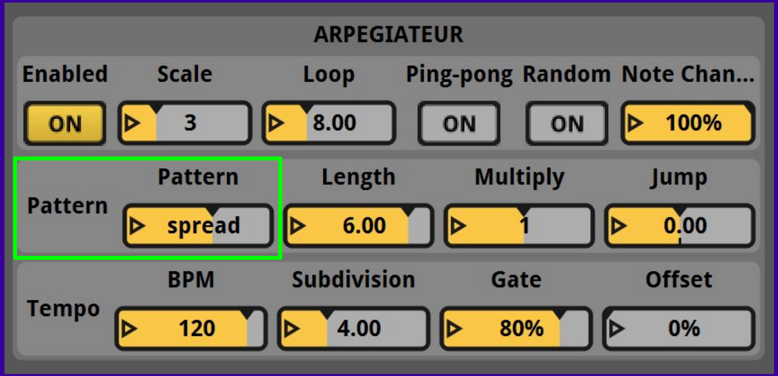


Scale 1	Scale 2	Scale 3	Scale 4	Scale 5	Scale 6	Scale 7
Prime	Fifth	Minor	Minor 7	Minor 9	Minor 11	Minor 13

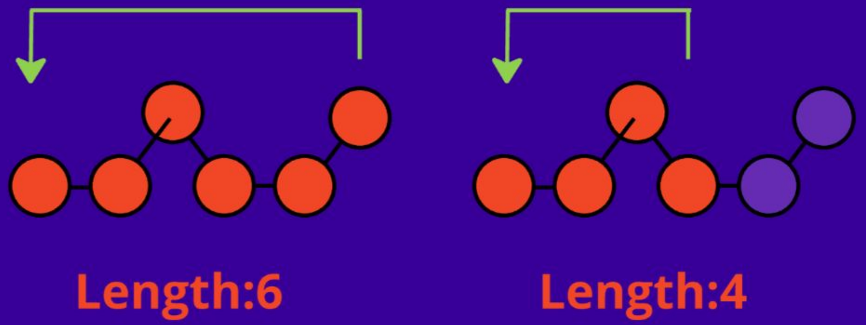


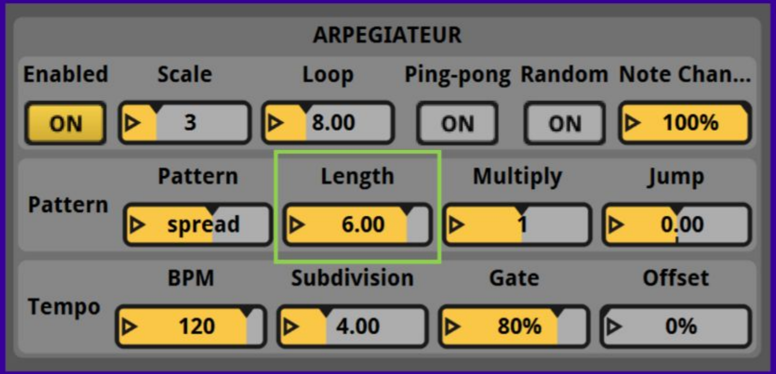
Arpeggiator: Pattern



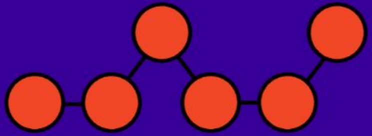


Arpeggiator: Length

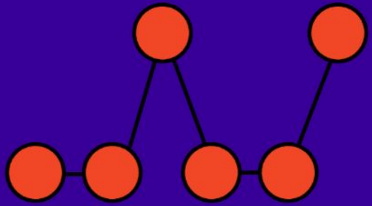




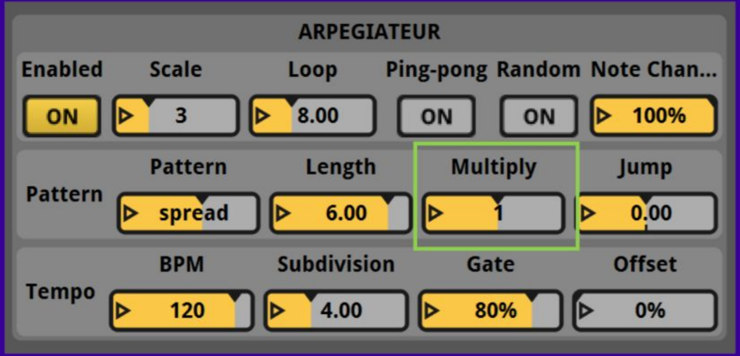
Arpeggiator: Multiply

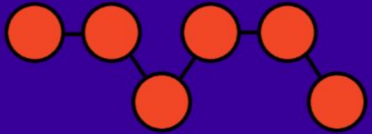


Multiply: 1



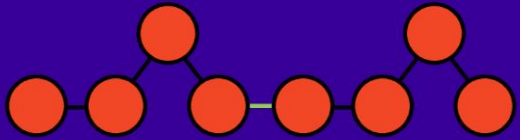
Multiply: 2



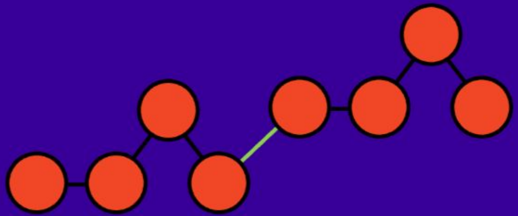


Multiply: -1

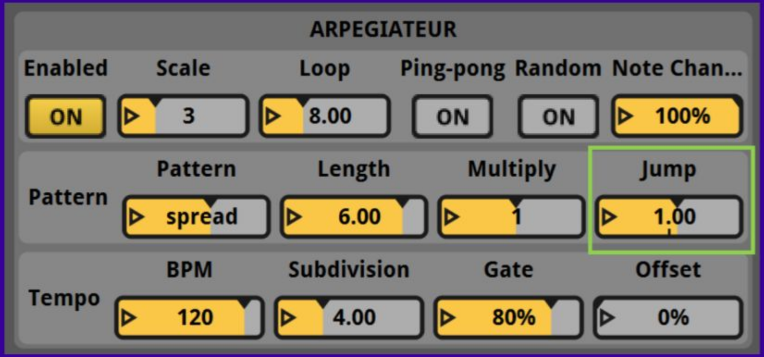
Arpeggiator: Jump



Jump:0

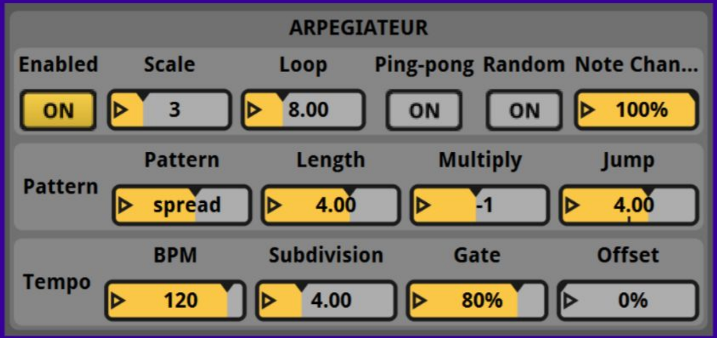


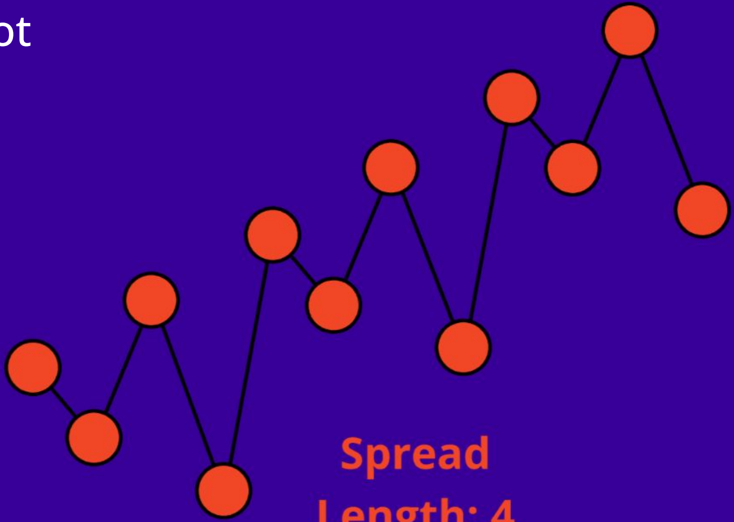
Jump:1



Arpeggiator: Combination

Combining these parameters gives a lot of flexibility





Spread
Length: 4
Multiply: -1
Jump: 4

K88

Granular synthesis (sort of)

Two modes Orchestra and Swarm

Shared sample bank 4MB built-in bank, recorded from classic synthesizers

Smearing reverb Series of 12 all-pass filters 'smears' the output to create soft pads

K88: Swarm Mode

Extracts grains from specified offset in sample bank

Scale and pitch controls Grains are tuned to scale between pitch min and max

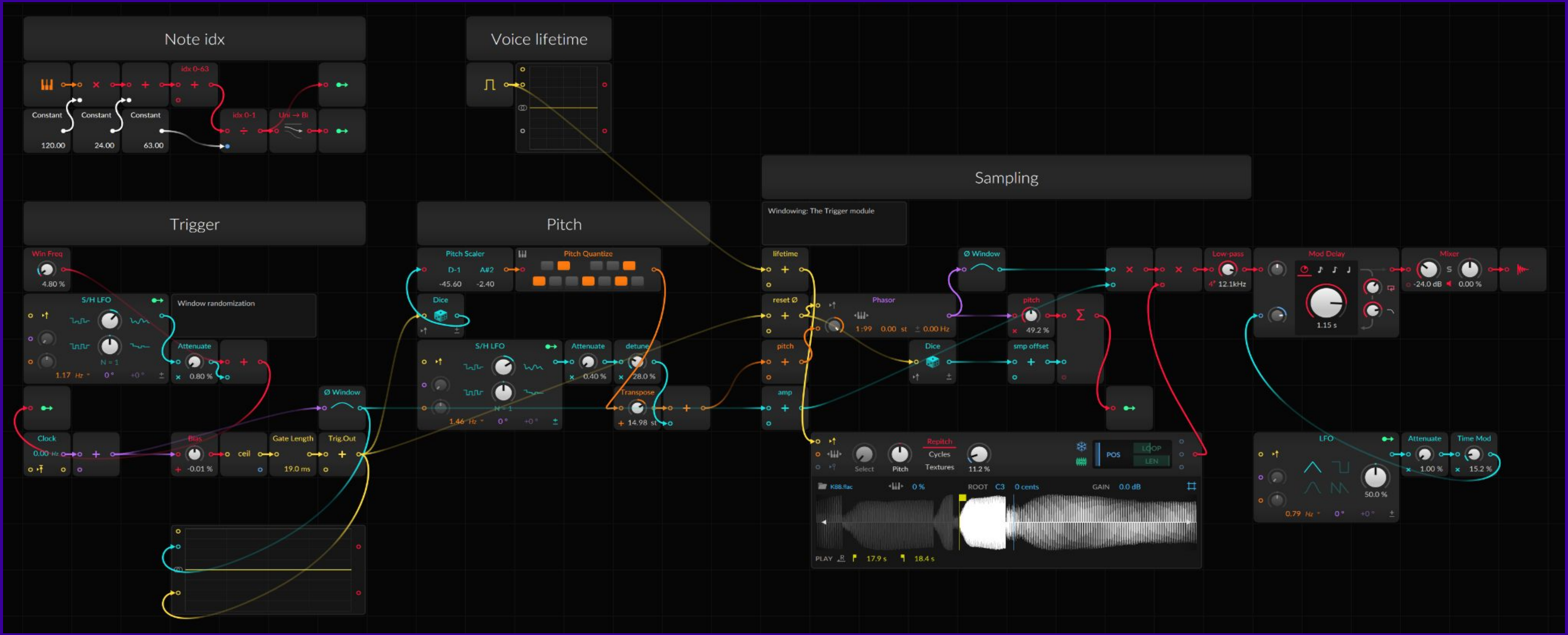
Per-voice pitch instability Each voice has random pitch modulation

Modulation delay Separate delay for each voice

- Gives each voice uniqueness, enlarging total sound

Low pass filter on output to remove unwanted high frequency artifacts

K88: Based on Bitwig Grid Patch



The K88 Swarm mode started as a Bitwig Grid patch

K88: Orchestra Mode

Slides parallel playheads across sample bank

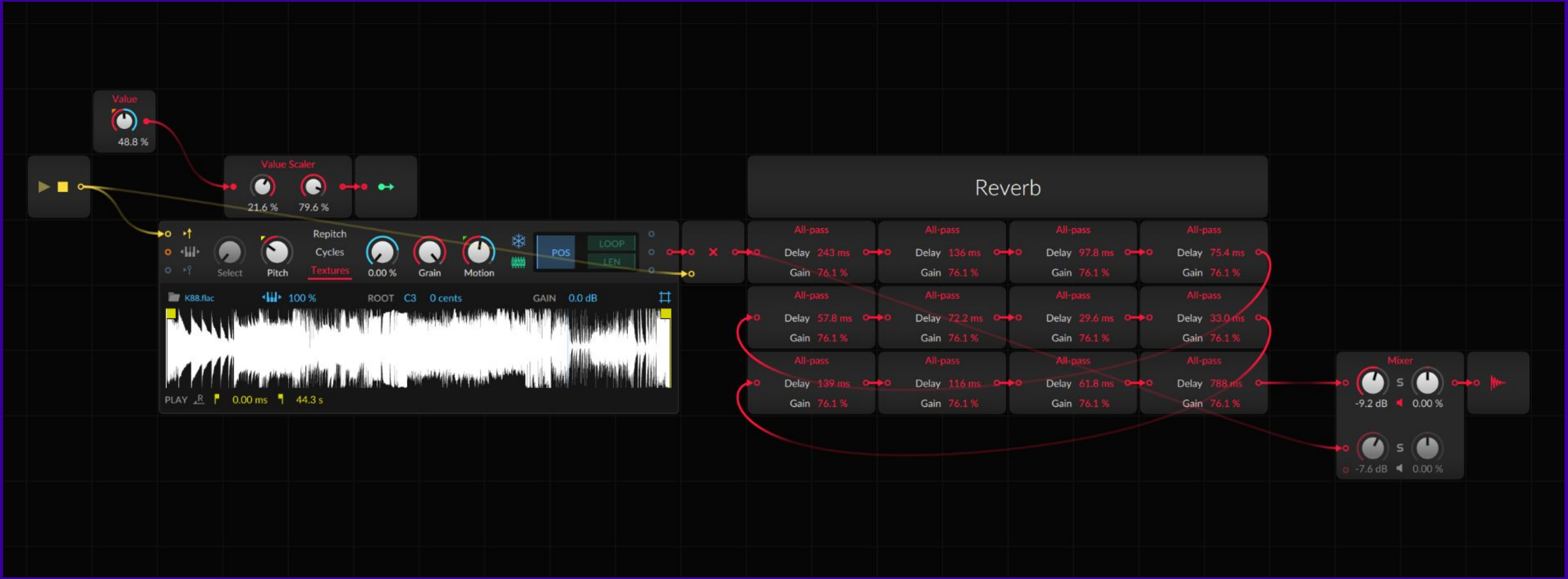
Windowed grains are extracted from bank and played

Random LFO controls playheads sliding

Random offset to each grain avoids robotic quality

Atonal orchestral sound works well for horror sequences

K88: Orchestra Mode



The K88 Orchestra mode started as a Bitwig Grid patch

Modnet

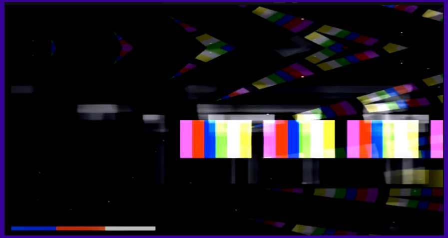
FM/AM synthesizer with 16 operators

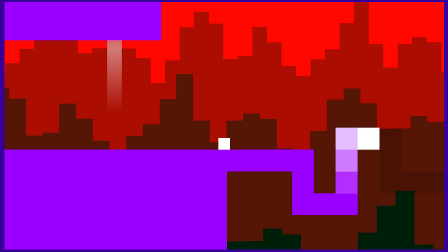
Interpolates between two configurations

Brass-like sounds used to dramatic effect in COCOON

Non-realtime version developed in 2013 for a live performance

Also used on the '140' soundtrack





Modnet

A meta-algorithm and a few parameters generates a patch (10 meta-algorithms)

Two patches are defined, A and B

Morph parameter interpolates between A and B

Morph LFO slightly varies morph to add life to sound

Interesting sounds are found close to A and B

Weather

Wind / rain simulation

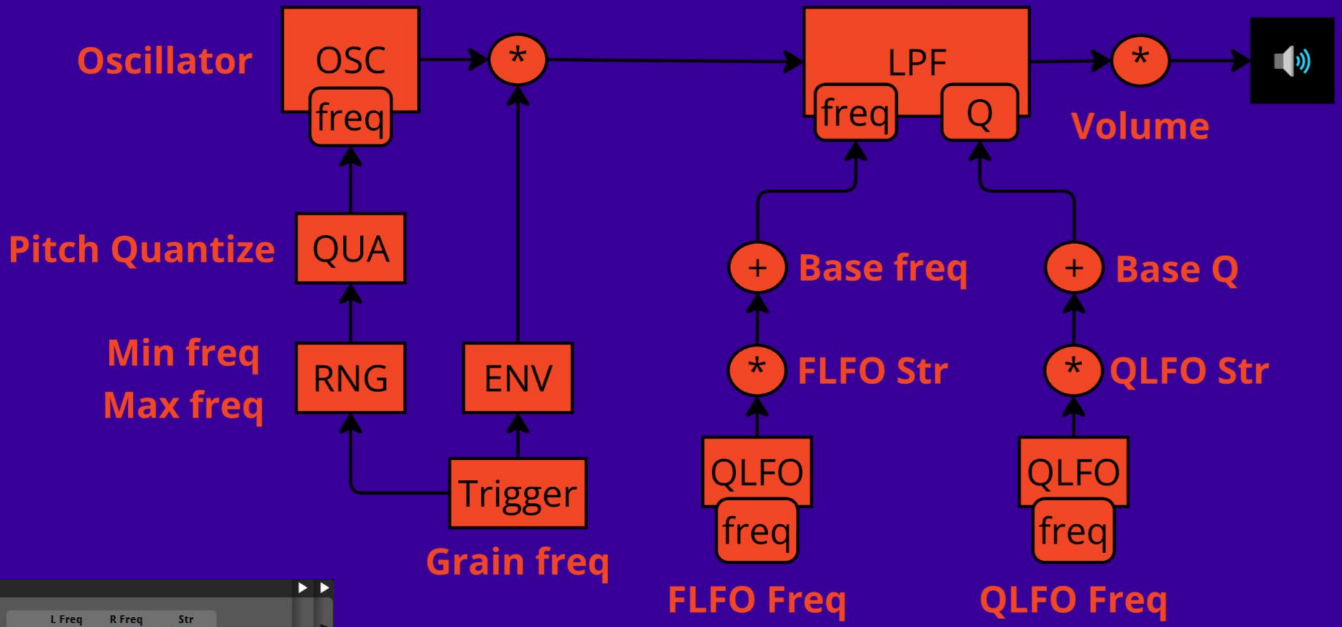
Generates grains up to 20.000 per second

Dual resonant filters left and right channel

Four LFOs controlling filter cutoff and resonance

Weather Structure

Duplicated for each stereo channel



Used for Ambience and Music

▸ develop2024-fmod-weather1 , develop2024-fmod-weather2

Composing with Plugins

Composing in FMOD Studio

Note Chance

Incremental Scale Control

Parameter-controlled Form

Mastering

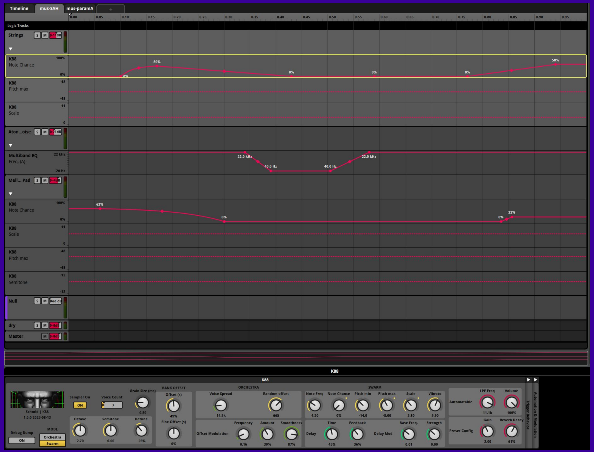
Composing in FMOD Studio

Experimentation with instruments

3 instruments for typical ambient music

Fixed effect buses reverb and delay used prominently

Some EQ required especially for Modnet



FMOD Event Structure

Useful event structure for plugin-based music

Null channel :

- Volume turned completely down
- All instrument channels Rerouted to Null channel

Dry output is a send same as the effects

Allows mixing/muting tracks while having complete control over dry/effects sends

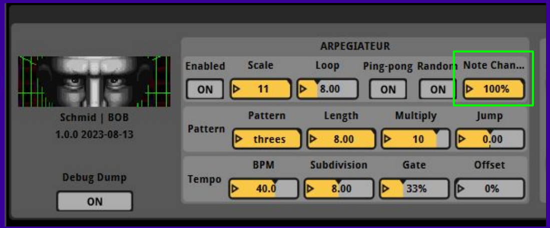
Note Chance

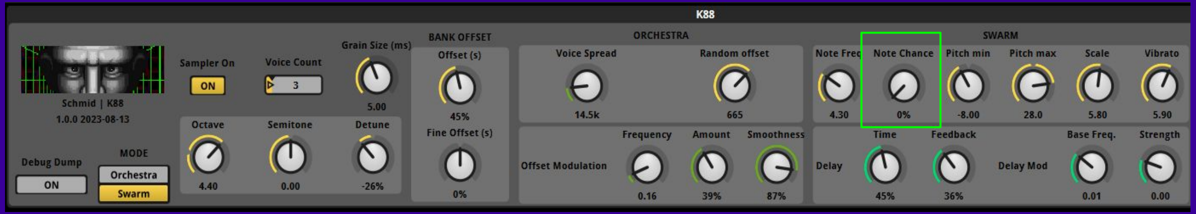
Used for BOB arpeggiator notes and K88 grains

Play chance for every note/grain triggered, roll a dice if it should play

Note-based fading automate to perform musical sounding note-based 'fades'

Note-based ducking set note chance to 0 to stop new notes during stingers





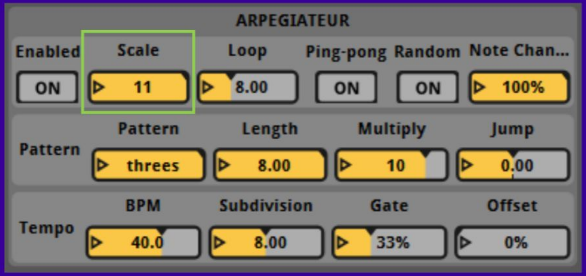
Incremental Scale Control

Used for BOB arpeggiator notes and K88 grains

Harmonic control allows music to react harmonically to game

Scale control up: More harmonic tension

Scale control down: Less harmonic tension

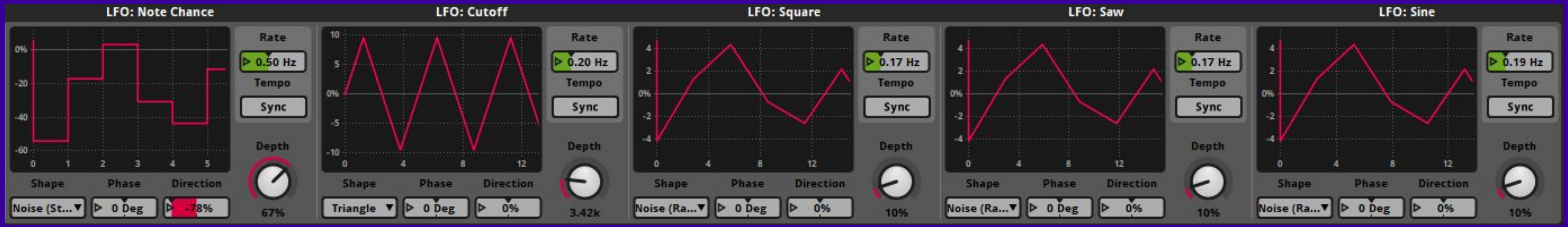


Scale 1	Scale 2	Scale 3	Scale 4	Scale 5	Scale 6	Scale 7
Prime	Fifth	Minor	Minor 7	Minor 9	Minor 11	Minor 13

Modulation Problems

Early in the project, I had unique modulations on individual parameters

Feels very dynamic and organic

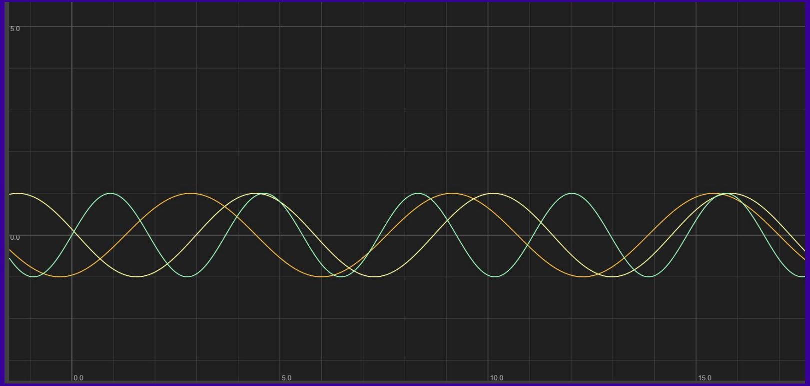


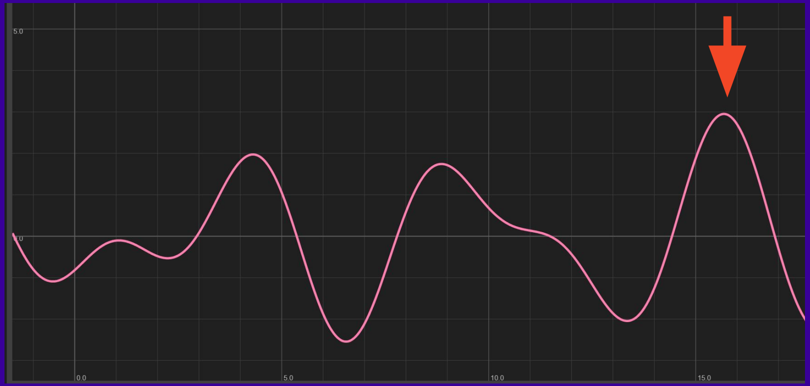
Constructive Interference

Combinations of modulators can produced unexpected results

Almost impossible to verify that a combination of modulators always play well together

Worst case, could cause clipping





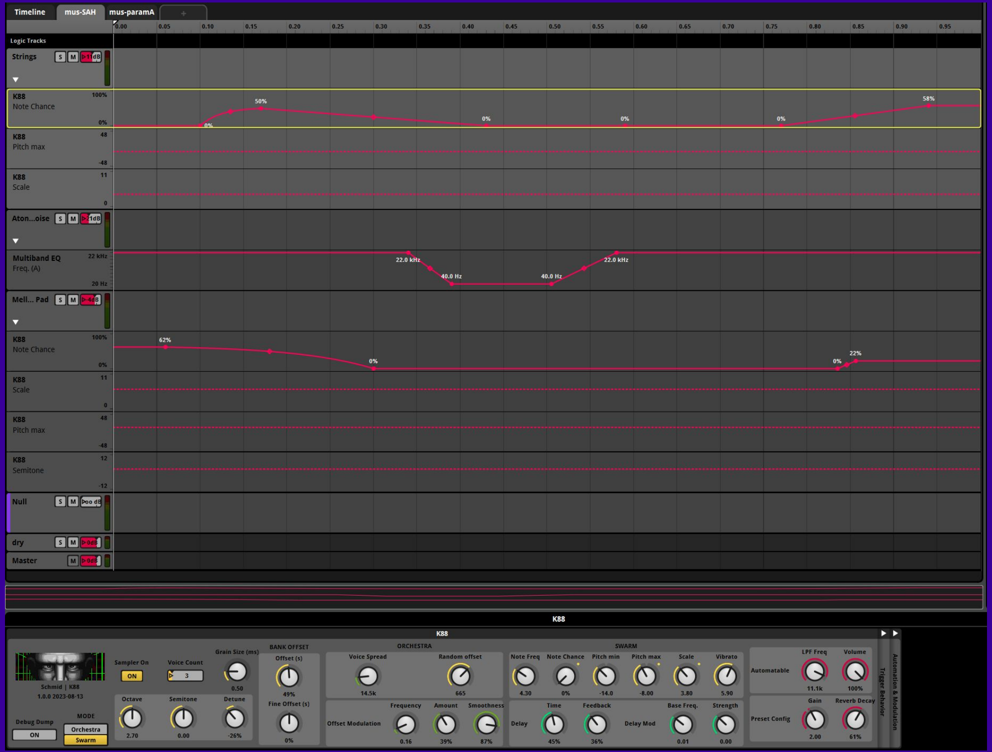
Parameter-controlled Form

Non-linear one dimensional score:

Like a linear score, but time can move back and forward arbitrarily

Define parameter sheet with all desired instrument configurations

Control using a single parameter



Parameter-controlled Form

It's a bit like the 'Hunt!' level in Braid

where you scrub through a short musical piece

But with an FMOD parameter instead of Tim

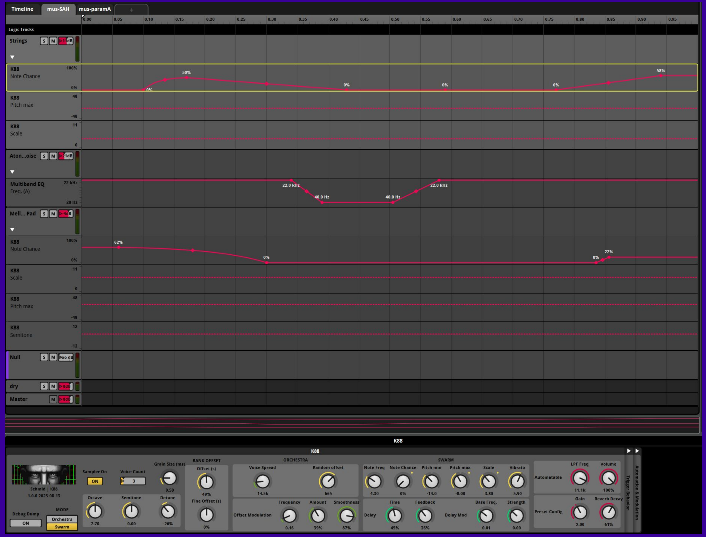
Parameter-controlled Form

Testable by manually scrubbing through the whole range

Control options Could be controlled by random LFO or game (e.g. player position on map)

Automate everything including key, scale, timbre, effects

Note chance is useful for transitions

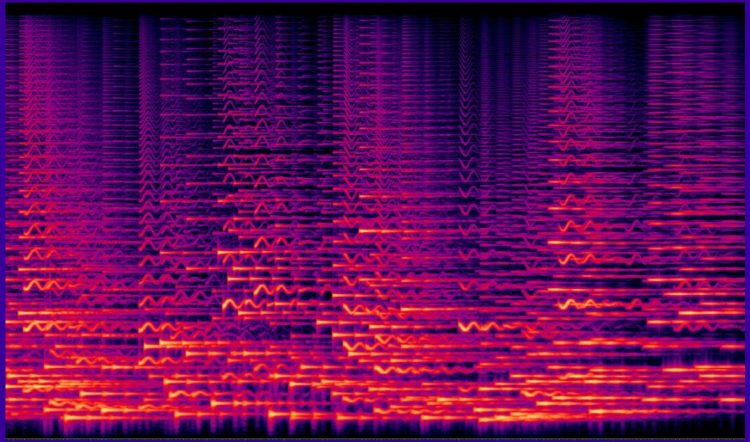


Mastering

Problem:

Production incoherence between pre-rendered vignettes and real-time synthesized ambient music

Coherence is improved by master plugin Wobble adding pitch instability to all music



Boss Fights

Mostly real-time synthesized

Music reacts to boss actions

▸ cocoon-cloak_boss.mkv

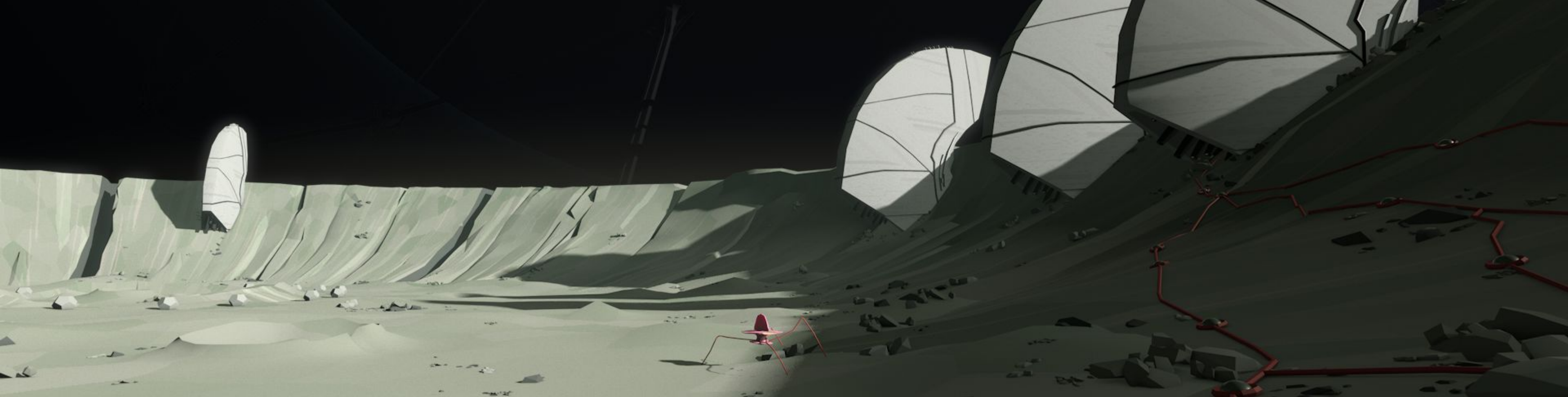
Plugin Implementation

From Bitwig Grid prototype to FMOD Studio Plugin

DSP Components and Signal Graph

K88, BOB, Modnet Components

Debugging



Disclaimer

Self-taught DSP programmer

I'm probably saying things wrong

Bear with me

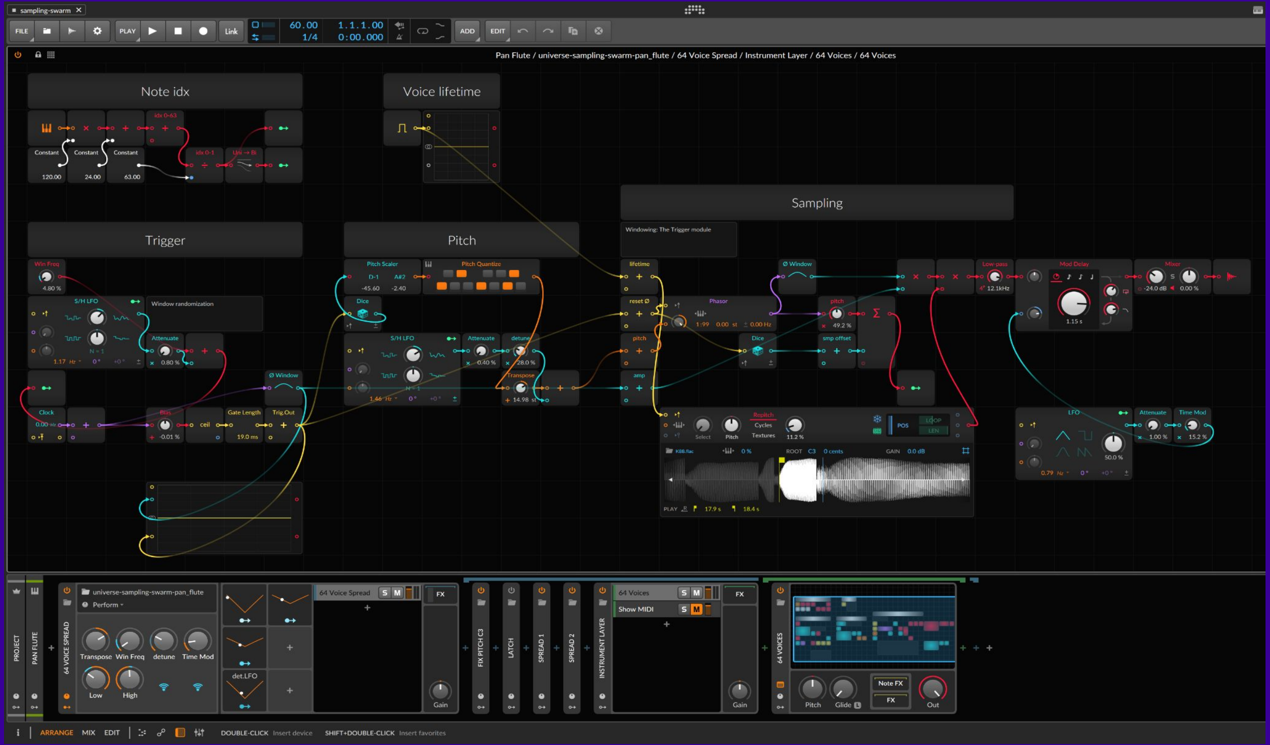
How to write an FMOD Studio Plugin

FMOD Studio plugin API is open

Plugins are normally written in C++

Start with example project and modify

From Bitwig Grid Prototype to Plugin



Bitwig Grid prototype for K88 Swarm mode

DSP Components

A Bitwig Grid patch can be expressed as a graph of DSP nodes.

It can be implemented as a set of nodes and a graph rendering algorithm.

Each node is a simple DSP component, such as:

- oscillator
- filter
- delay

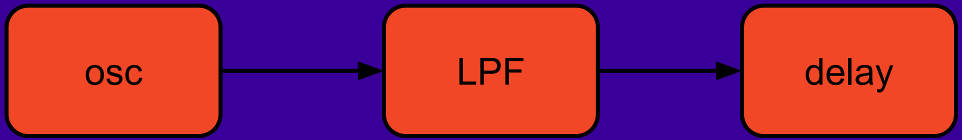
More about these later…

A selection of useful Bitwig Grid nodes

Signal Graph

The signal graph can be implemented in code as a fixed sequence of component updates.

For example,



this graph can be rendered like this:

1. render osc output, then
2. render LPF using output from osc as input
3. render delay using output from LPF as input

```
osc          = get_osc_output()
osc_filtered = lpf.process(osc)
out          = delay.render(osc_filtered)
```

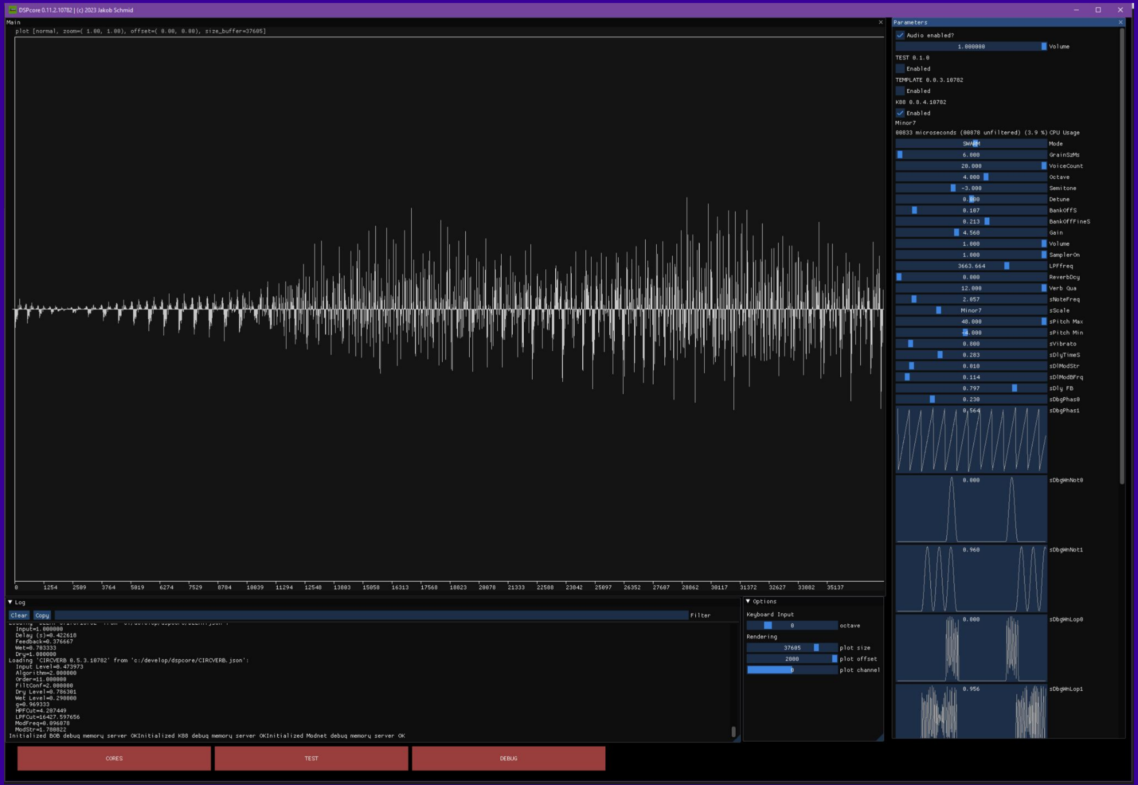
Example signal graph implementation (pseudocode)

DSPcore.exe: Test Interface

Visualizes output waveform

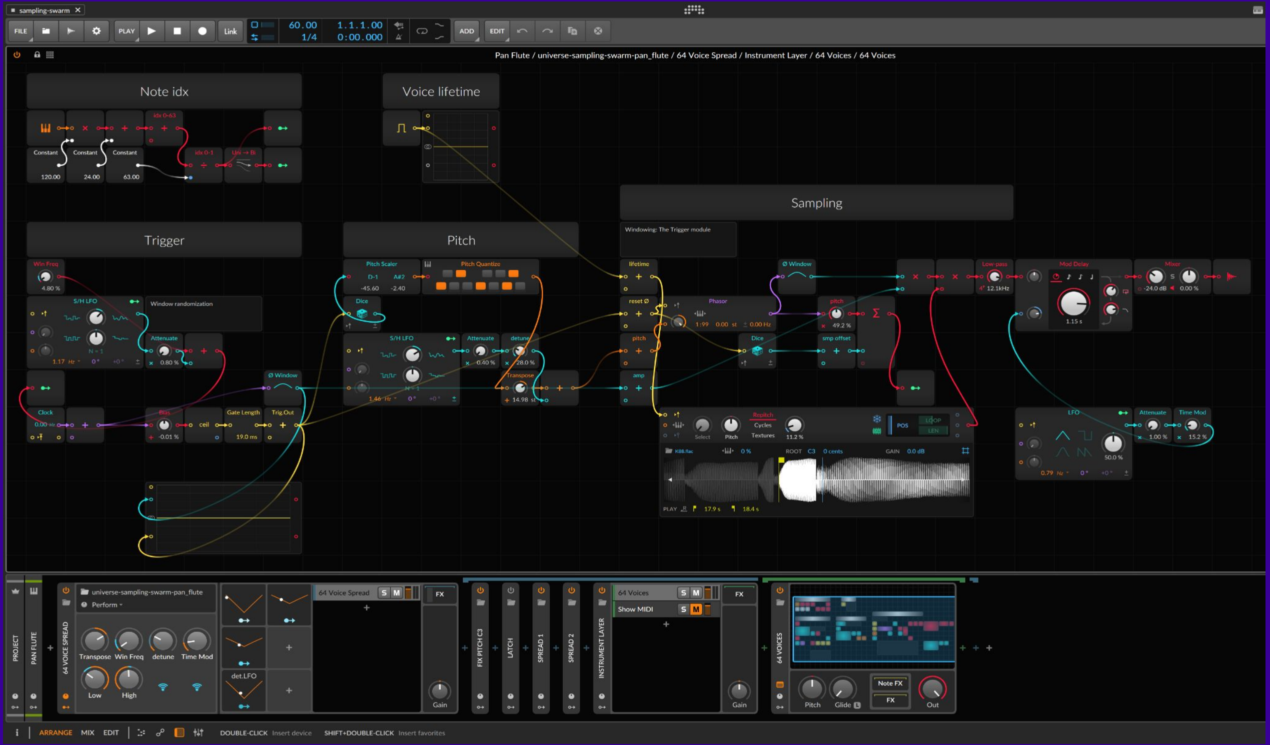
Easy to step debug

Can show debug info for plugin instances



Wrap as FMOD Plug-in Instrument

K88: Implementing Swarm Mode



Bitwig Grid patch

K88 Components

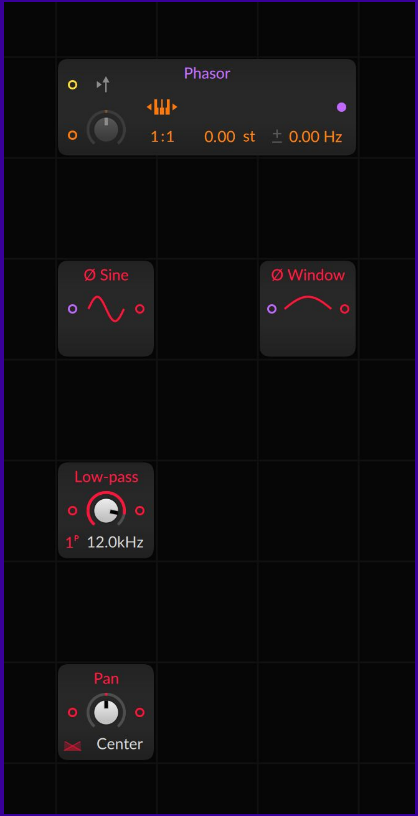
Phase generator clock component, generates a control signal 0..1

Look-up table combines with phase generator to make oscillators, Hanning windows

Low-pass filter to remove unwanted high frequency content (simple 1-pole LPF)

DC filter to avoid build-up of DC offset

Constant power panner for spreading voices in the stereo field



K88 Components

Sampler with linear interpolation

Sample and hold component with smoothing

Modulation delay based on circular buffer with linear interpolation

All-pass filter based on circular buffer

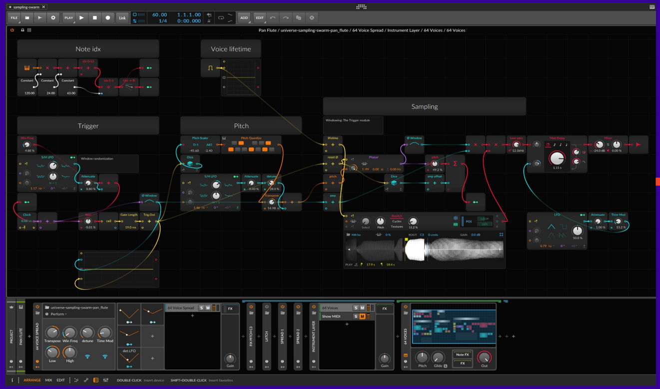
Pitch quantizer quantizes random pitch to specified scale

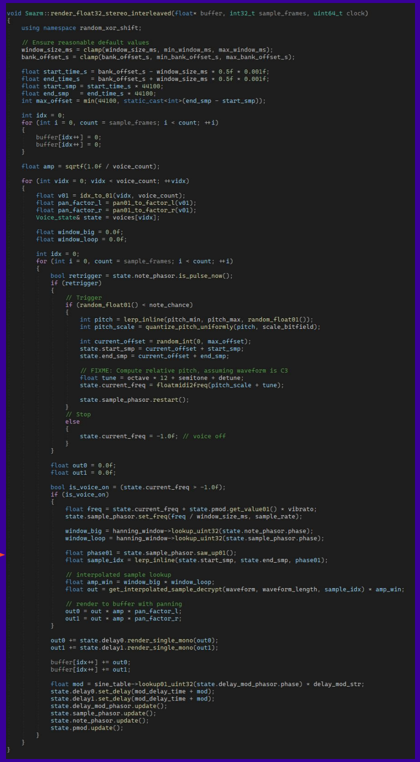
K88: Translate Signal Graph to C++

Signal graph is a tree structure

Render 'leafs' first, use as input for branches

If possible, let component render entire buffer, otherwise 1 sample at a time





```cpp
void Swarm::render_float32_stereo_interleaved(float* buffer, int32_t sample_frames, uint64_t clock)
{
    using namespace random_xor_shift;

    // Ensure reasonable default values
    window_size_ms = clamp(window_size_ms, min_window_ms, max_window_ms);
    bank_offset_s = clamp(bank_offset_s, min_bank_offset_s, max_bank_offset_s);

    float start_time_s = bank_offset_s - window_size_ms * 0.5f * 0.001f;
    float end_time_s   = bank_offset_s + window_size_ms * 0.5f * 0.001f;
    float start_smp = start_time_s * 44100;
    float end_smp   = end_time_s * 44100;
    int max_offset = min(44100, static_cast<int>(end_smp - start_smp));

    int idx = 0;
    for (int i = 0, count = sample_frames; i < count; ++i)
    {
        buffer[idx++] = 0;
        buffer[idx++] = 0;
    }

    float amp = sqrtf(1.0f / voice_count);

    for (int vidx = 0; vidx < voice_count; ++vidx)
    {
        float v01 = idx_to_01(vidx, voice_count);
        float pan_factor_l = pan01_to_factor_l(v01);
        float pan_factor_r = pan01_to_factor_r(v01);
        Voice_state& state = voices[vidx];

        float window_big = 0.0f;
        float window_loop = 0.0f;

        int idx = 0;
        for (int i = 0, count = sample_frames; i < count; ++i)
        {
            bool retrigger = state.note_phasor.is_pulse_now();
            if (retrigger)
            {
                // Trigger
                if (random_float01() < note_chance)
                {
                    int pitch = lerp_inline(pitch_min, pitch_max, random_float01());
                    int pitch_scale = quantize_pitch_uniformly(pitch, scale_bitfield);

                    int current_offset = random_int(0, max_offset);
                    state.start_smp = current_offset + start_smp;
                    state.end_smp = current_offset + end_smp;

                    // FIXME: Compute relative pitch, assuming waveform is C3
                    float tune = octave * 12 + semitone + detune;
                    state.current_freq = floatmidi2freq(pitch_scale + tune);

                    state.sample_phasor.restart();
                }
                // Stop
                else
                {
                    state.current_freq = -1.0f; // voice off
                }
            }

            float out0 = 0.0f;
            float out1 = 0.0f;

            bool is_voice_on = (state.current_freq > -1.0f);
            if (is_voice_on)
            {
                float freq = state.current_freq + state.pmod.get_value01() * vibrato;
                state.sample_phasor.set_freq(freq / window_size_ms, sample_rate);

                window_big = hanning_window->lookup_uint32(state.note_phasor.phase);
                window_loop = hanning_window->lookup_uint32(state.sample_phasor.phase);

                float phase01 = state.sample_phasor.saw_up01();
                float sample_idx = lerp_inline(state.start_smp, state.end_smp, phase01);

                // interpolated sample lookup
                float amp_win = window_big * window_loop;
                float out = get_interpolated_sample_decrypt(waveform, waveform_length, sample_idx) * amp_win;

                // render to buffer with panning
                out0 = out * amp * pan_factor_l;
                out1 = out * amp * pan_factor_r;
            }

            out0 += state.delay0.render_single_mono(out0);
            out1 += state.delay1.render_single_mono(out1);

            buffer[idx++] += out0;
            buffer[idx++] += out1;

            float mod = sine_table->lookup01_uint32(state.delay_mod_phasor.phase) * delay_mod_str;
            state.delay0.set_delay(mod_delay_time + mod);
            state.delay1.set_delay(mod_delay_time + mod);
            state.delay_mod_phasor.update();
            state.sample_phasor.update();
            state.note_phasor.update();
            state.pmod.update();
        }
    }
}
```

BOB Components

Band-limited oscillators to avoid aliasing of sawtooth and square waves

Ladder filter for resonant filtering

DC filter removes DC offset that can be introduced in signal chains

BOB: Band-limited Oscillators

Saw

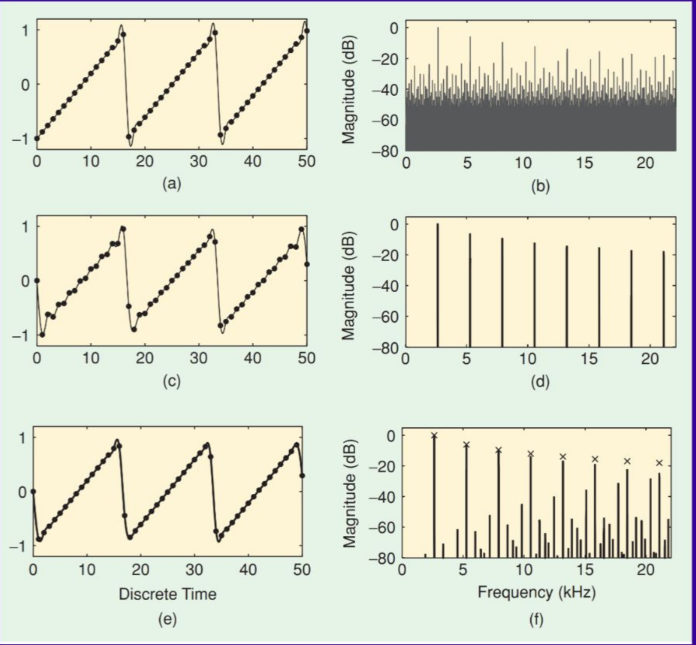
Sqr

Trivial sawtooth

Ideal band-limited sawtooth: sum of sines

PolyBLEP approximation

$$p_{\text{PolyBLEP}}(t) = \begin{cases} \frac{t^2}{2} + t + \frac{1}{2}, & \text{when } -1 \leq t \leq 0 \\ t - \frac{t^2}{2} - \frac{1}{2}, & \text{when } 0 < t \leq 1. \end{cases} \quad (7)$$



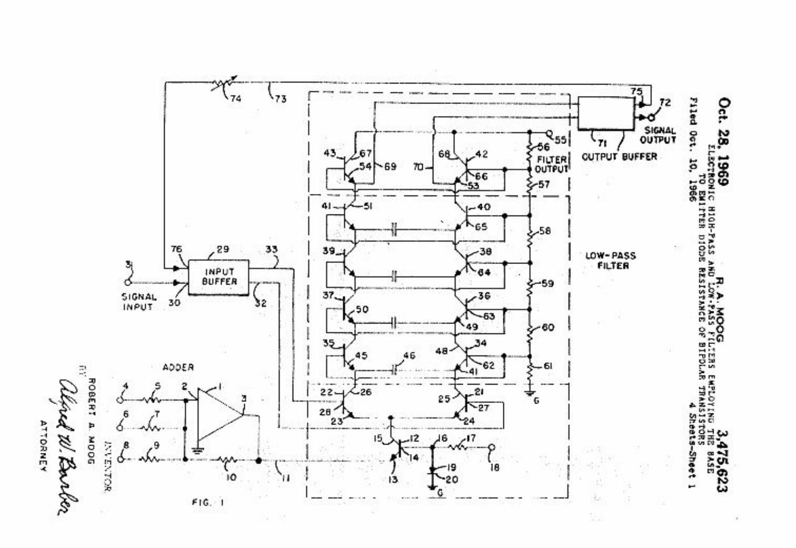
Antialiasing Oscillators in Subtractive Synthesis

Article *in* IEEE Signal Processing Magazine · April 2007

DOI: 10.1109/MSP.2007.323276 · Source: IEEE Xplore

BOB: Ladder Filter

Ladder Filter



Proc. of the 7th Int. Conference on Digital Audio Effects (DAFx'04), Naples, Italy, October 5-8, 2004

NON-LINEAR DIGITAL IMPLEMENTATION OF THE MOOG LADDER FILTER

Antti Huovilainen

Laboratory of Acoustics and Audio Signal Processing
Helsinki University of Technology, P.O. Box 3000, FIN-02015 HUT, Espoo, Finland
ajhuovil@acoustics.hut.fi

Difference equations can now be written for the full ladder filter.

$$y_a(n) = y_a(n-1) + \frac{I_{ctl}}{CF_s}\left(\tanh\left(\frac{x(n) - 4ry_d(n-1)}{2V_t}\right) - W_a(n-1)\right) \quad (13)$$

$$y_b(n) = y_b(n-1) + \frac{I_{ctl}}{CF_s}(W_a(n) - W_b(n-1)) \quad (14)$$

$$y_c(n) = y_c(n-1) + \frac{I_{ctl}}{CF_s}(W_b(n) - W_c(n-1)) \quad (15)$$

$$y_d(n) = y_d(n-1) + \frac{I_{ctl}}{CF_s}\left(W_c(n) - \tanh\left(\frac{y_d(n-1)}{2V_t}\right)\right) \quad (16)$$

where $x(n)$ is the input, $y_a(n)$, $y_b(n)$, $y_c(n)$ and $y_d(n)$ are the outputs of individual filter stages, r is the resonance amount ($0 < r \leq 1$) and

$$W_{\{a,b,c\}}(n) = \tanh\left(\frac{y_{\{a,b,c\}}(n)}{2V_t}\right) \quad (17)$$

Modnet Components

16 interconnected sine oscillators with FM and AM

Quality parameter controls number of round-robin updates

Patch generation based on 10 meta-algorithms and parameters

Morphing between two patches

Automatic morph modulation to give life to the sound

Modnet: Morphing Between Patches

Modulation is represented using matrices

One modulation matrix for FM, one for AM

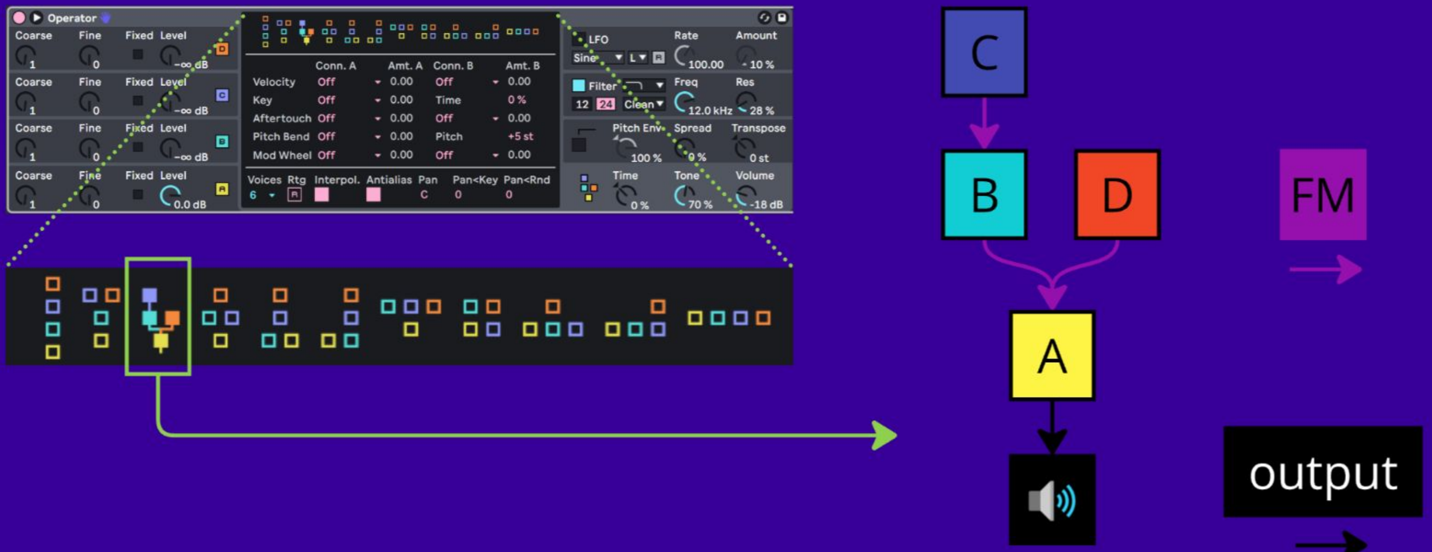
Frequency Modulation Matrix

FM	A	B	C	D	🔊
A	0	0	0	0	0
B	1	0	0	0	0
C	0	1	0	0	0
D	1	0	0	0	0
🔊	0	0	0	0	0

Amplitude Modulation / Output Matrix

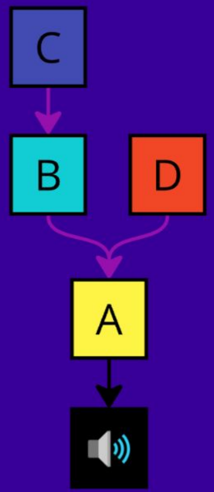
AM	A	B	C	D	🔊
A	0	0	0	0	1
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
🔊	0	0	0	0	0

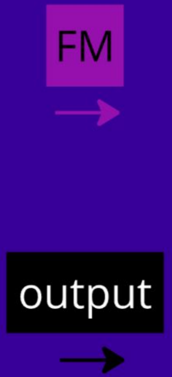
Modnet: Ableton Operator Algorithm 3



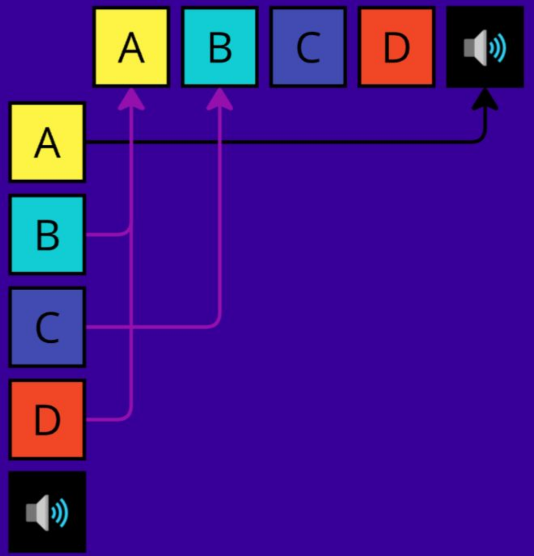
Modnet Normalized Form

Operator Algorithm

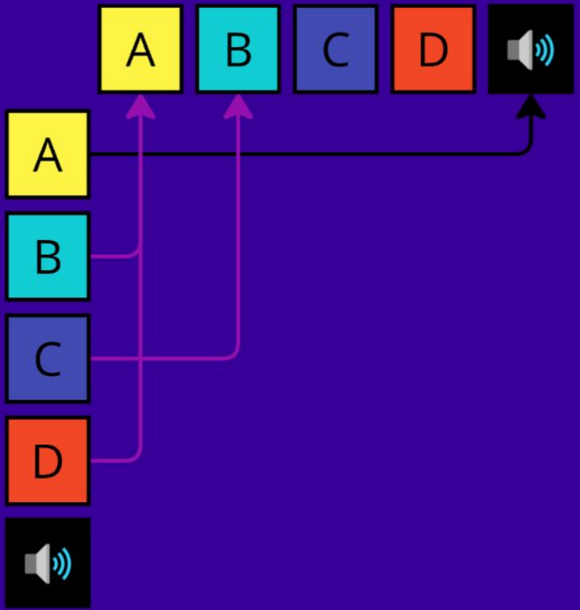




Equivalent
vertical ➡ horizontal
routing representation



Modnet: FM and AM/output Matrices



Frequency Modulation Matrix

FM	A	B	C	D	
A	0	0	0	0	0
B	1	0	0	0	0
C	0	1	0	0	0
D	1	0	0	0	0
	0	0	0	0	0

Amplitude Modulation / Output Matrix

AM	A	B	C	D	
A	0	0	0	0	1
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
	0	0	0	0	0

Modnet: Interpolation

Frequency Modulation Matrix 1

FM	A	B	C	D	
A	0	0	0	0	0
B	1	0	0	0	0
C	0	1	0	0	0
D	1	0	0	0	0
	0	0	0	0	0

Frequency Modulation Matrix 2

FM	A	B	C	D	
A	1	0	0	0	0
B	1	0	0	0	0
C	0	0.5	0	0	0
D	0	1	0	0	0
	0	0	0	0	0

Modnet in COCOON: 16 Operators

FM	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	🔊
A																	
B																	
C																	
D																	
E																	
F																	
G																	
H																	
I																	
J																	
K																	
L																	
M																	
N																	
O																	
P																	
🔊																	

AM	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	🔊
A																	
B																	
C																	
D																	
E																	
F																	
G																	
H																	
I																	
J																	
K																	
L																	
M																	
N																	
O																	
P																	
🔊																	

Debugging in DSPcore.exe

Ideally, we wanted to debug running instances of plugins

both in FMOD Studio and in the running game

Debugging in DSPcore.exe


```
instance_count: 1

MODNET 1.0.0.21254
C:\Program Files (x86)\Steam\steamapps\common\Universe\universe.exe
      OpCount:    16.000        Quality:     3.000        Alg A: Waving Ch      Alg A.P0:     1.000
     Alg A.P1:     1.000          Alg B:     Noise     Alg B.P0:     0.350      Alg B.P1:     0.470
     Octave A:     2.000     Semitone A:     4.000     Detune A:     0.000      Octave B:     2.000
   Semitone B:     0.000       Detune B:     0.000        Amp A:     0.670         Amp B:     0.700
        Morph:     0.215   Morph easing:     3.000 Morph mod str:     0.040 Morph mod freq:     0.009
     LPF freq: 12000.000       HPF freq:    65.000

instance_count: 1

BOB 1.0.0.21254
C:\Program Files (x86)\Steam\steamapps\common\Universe\universe.exe
   ARP:enabled:      1.00      ARP:scale:    Minor9 ARP:loop length:     30.00   ARP:ping pong:      0.00
    ARP:random:      1.00    ARP:pattern:      3.00 ARP:pat.length:       8.00   ARP:multiply:       5.00
ARP:rest.delta:      0.00      ARP:tempo:     60.00 ARP:subdivision:      1.00       ARP:gate:       0.94
    ARP:offset:      0.00     ARP:octave:      4.00   ARP:semitone:      -5.00 ARP:notechance:       0.20
        Cutoff:  -3400.00   Key tracking:      0.00    FENV:amount:    5800.00      Resonance:       0.20
    Square amp:      0.38       Saw amp:       0.65       Sine amp:       0.65     Tr. Square:       7.00
      Tr. Saw:       0.00      Tr. Sine:     -12.00        PWM str:       0.55       PWM freq:       0.19
    PLFO str:        0.29     PLFO freq:       6.00   AENV:attack:        0.02    AENV:decay:        0.01
  AENV:sustain:      0.77  AENV:release:      7.60   FENV:attack:        0.55    FENV:decay:        4.60
  FENV:sustain:      0.59  FENV:release:      7.40   PNOTE:pitch:      -24.00     PNOTE:amp:        0.00


instance_count: 3

K88 1.0.0.21254
C:\Program Files (x86)\Steam\steamapps\common\Universe\universe.exe
        Mode:      SWARM     GrainSzMs:      0.000    VoiceCount:      3.000         Octave:      2.00
    Semitone:      0.000        Detune:     -0.260      BankOffS:      0.533    BankOffFineS:      0.00
        Gain:      2.000        Volume:      1.000     SamplerOn:      1.000         LPFfreq: 11100.00
   ReverbDcy:      0.610    oVocOffSmp:  14500.000    oOffModAmt:      0.385      oOffModFrq:      0.16
  oOffModSmo:      0.865     oVibraStr:    665.000     oGrStFrst:    0000000       oGrStLast:    000000
   oGrPsFrst:    0000000     oGrPsLast:    0000000     sNoteFreq:      4.300       sNoteChnc:      4.00
      sScale:        N/A    sPitch Max:    -14.000    sPitch Min:      5.900      sPitch Atn:      0.45
    sVibrato:      0.005     sDlyTimeS:      0.011     sDlModStr:      0.360      sDlModBFrq:      0.00
     sDly FB:      0.988     sDbgPhas1:      0.000    sDbgWnNot0:      0.000      sDbgWnNot1:      0.00
  sDbgWnLop0:      0.000    sDbgWnLop1:      0.000

K88 1.0.0.21254
C:\Program Files\FMOD SoundSystem\FMOD Studio 2.02.14\FMOD Studio.exe
        Mode:      SWARM     GrainSzMs:    280.000    VoiceCount:      3.000         Octave:      3.00
    Semitone:      6.000        Detune:      0.020      BankOffS:      0.500    BankOffFineS:     -0.22
        Gain:      4.000        Volume:      0.515     SamplerOn:      0.000         LPFfreq: 10100.00
   ReverbDcy:      0.625    oVocOffSmp:      0.000    oOffModAmt:      0.000      oOffModFrq:      0.00
  oOffModSmo:      0.000     oVibraStr:      0.000     oGrStFrst:    0000000       oGrStLast:    000000
   oGrPsFrst:    0000000     oGrPsLast:    0000000     sNoteFreq:      1.460       sNoteChnc:      6.00
      sScale:        N/A    sPitch Max:    -17.000    sPitch Min:      5.300      sPitch Atn:      0.67
    sVibrato:      0.003     sDlyTimeS:      0.060     sDlModStr:      0.740      sDlModBFrq:      1.00
     sDly FB:      0.294     sDbgPhas1:      0.058    sDbgWnNot0:      0.637      sDbgWnNot1:      0.44
  sDbgWnLop0:      0.280    sDbgWnLop1:      0.000
```

Shared Memory for Debugging

Shared memory between DSPcore.exe and plugin instances (regardless of host app)

Each instance copies its internal state to shared memory

DSPcore.exe visualizes the internal state of each plugin

Works regardless of plugin API (FMOD, VST, Unity NAP, standalone)

```
class Shared_memory
{
    struct Buffer
    {
        struct Instance
        {
            bool active;
            char process_name[PROCESS_NAME_SIZE];
            char plugin_name[PLUGIN_NAME_SIZE];
            float params[PARAMS_MAX];
        };

        uint32_t instance_count;
        Instance instances[INSTANCES_MAX];
    };
};
```

Shared Memory using FileMappings

DSPcore.exe creates a local FileMapping using `CreateFileMapping`

If it exists, plugin instances open it using `OpenFileMapping`

File is mapped to a memory buffer using `MapViewOfFile`

Now that the memory is shared, plugins can write, and DSPcore.exe can read

```
const int buf_size = sizeof(Buffer);
Buffer* buf;
HANDLE map_file;
const char* map_name = "Local\\MyApp";

void init_server()
{
    map_file = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, buf_size, map_name);
    buf = (Buffer*)MapViewOfFile(map_file, FILE_MAP_ALL_ACCESS, 0, 0, buf_size);
}

void init_client()
{
    map_file = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, map_name);
    buf = (Buffer*)MapViewOfFile(map_file, FILE_MAP_ALL_ACCESS, 0, 0, buf_size);
}
```

Closing Thoughts

Other Wrappers

Platforms

Other Wrappers

The DSPcore synths can easily be wrapped as other plugin formats:

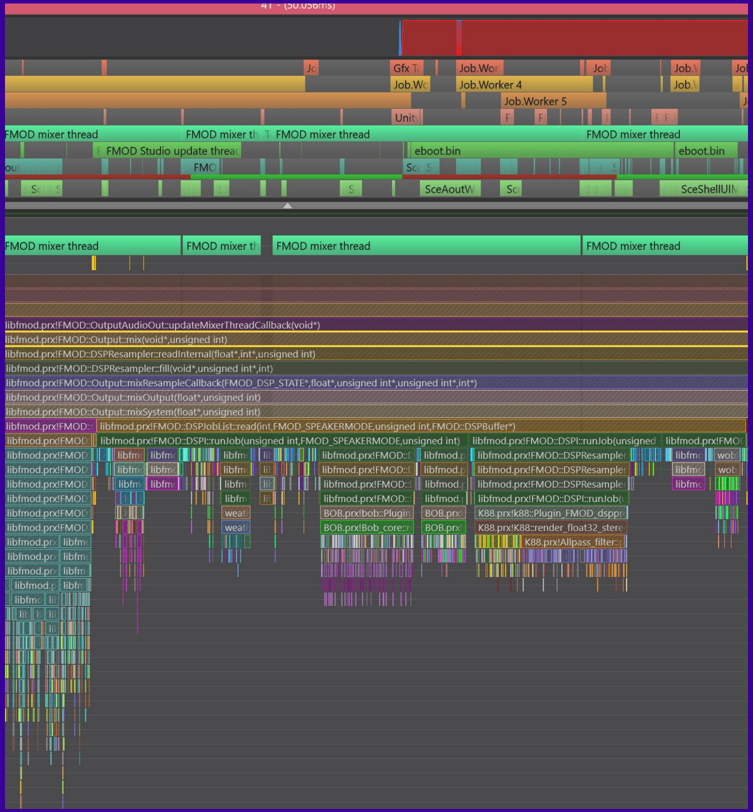
- Steinberg VST for music software
- Unity Native Audio Plugin for the built-in Unity audio system

All DSP code is reused, only plugin interface is different

All Platforms

The COCOON plugins run on these platforms

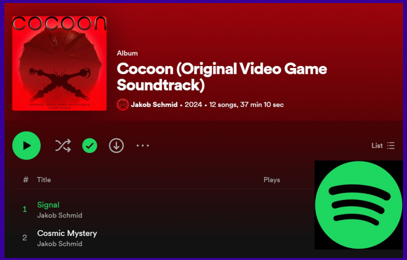
- Windows
- Xbox Series S|X, Xbox One
- PlayStation 5, PlayStation 4
- Nintendo Switch



Game and Soundtrack

cocoongame.com

Twitter: @playcocoon



AVAILABLE NOW ON

STEAM XBOX NINTENDO SWITCH PS5 | PS4

Questions?

Contact me on

E-mail: jakob@schmid.dk

Twitter: @jakobschmid

Slides will be available here: schmid.dk/talks

Please rate my session!

Bonus Slides

K88: Phase Generator

Clock component that generates a control signal 0..1

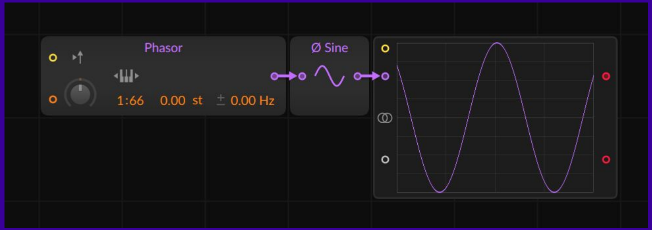
Oscillator use as input for a function or table to generate any periodic signal

Example:

```
ph01 = ph_gen.get_phase()

sin_osc = sin(ph01 * 2 * PI)
```

Internally uses unsigned 32-bit integer as counter

Bitwig Grid phase generator with oscilloscope



Generating sine wave using phase generator

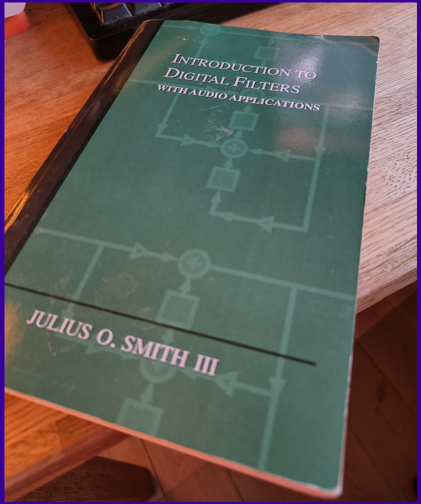
Core Component: DC Filter

DCF

```
y[n] = x[n] - x[n-1] + R * y[n-1]
   R = 1 - (2*PI * cutoff_freq / sample_rate)
```

- Difference equation from 'Introduction to Digital Filters: with Audio Applications' (JOS 2007)

- R calculation by hc.niweulb@lossor.ydna at musicdsp.org



Phase Generator Implementation

Effective implementation using 32-bit unsigned integer as clock counter

Modular arithmetic using the 'wrapping' type uint32_t

(don't use signed int, it doesn't support this)

Update method is a single line:

```
phase += freq;
```

Frequency is represented as phase increment per update

Phase Generator Implementation

32-bit fixed point representation of a fractional number in the range [0;1)

```
Binary: 0.00000000000000000000000000000000 represents 0.0
Binary: 0.11111111111111111111111111111111 represents 0.999985
```

Phase Generator Implementation

```cpp
class Phaser
{
  const float PHASE_MAX = 4294967296; // = 0x100000000

  uint32_t phase, freq;
  bool is_active;

  void set_freq(float freq__hz, int update_rate)
  {
    // freq_float: periods / update
    float freq_f = freq__hz / update_rate;

    // freq: periods / update, scaled to full uint32_t range
    freq = static_cast<uint32_t>(freq_f * PHASE_MAX);
  }

  void update()          { phase += freq; }
  uint32_t get_phase() { return phase; }
};
```

Excerpt of phase generator implementation

Core Component: Fast_table

Table with fast lookup using uint32_t phase as input.

Useful for sine tables and the like with predictable performance across platforms.

Combines with Phaser to form an oscillator.


```
Fast_table<20> sine_table;

void init()
{
    sine_table.init_sine(); // generates 1M float sine table
}
void update()
{
    float sine_osc = sine_table.lookup_uint32(phasor_sine.phase);
}
```

Fast_table Implementation

Inspired by MC68000 assembly code...

Table size is power of two for fast lookup.

Bit_size	size()
8	256
20	~1M

Uses bit shifted phase as index

Can be resized for enhanced accuracy without modifying lookup code.

```
Fast_table<20> sine_table;

void init()
{
    sine_table.init_sine(); // generates 1M float sine table
}
void update()
{
    float sine_osc = sine_table.lookup_uint32(phasor_sine.phase);
}
```

```
template<int Bit_size> class Fast_table
{
    std::vector<float> table;

    void init_sine();     // f(x) = sin(2*PI*x), x in [0;1]
    void init_hanning(); // f(x) = sin(PI*x)^2, x in [0;1]
    constexpr uint32_t size();
    float lookup(uint32_t phase_32bit);
};
```

Fast_table Lookup Code

```
template<int Bit_size> class Fast_table
{
    std::vector<float> table;

    void init_sine();     // f(x) = sin(2*PI*x), x in [0;1]
    void init_hanning();  // f(x) = sin(PI*x)^2, x in [0;1]
    constexpr uint32_t size();
    float lookup(uint32_t phase_32bit);
};

template<int Bit_size>
constexpr uint32_t Fast_table<Bit_size>::size()
{
    return 1 << Bit_size;
}

template<int Bit_size>
float Fast_table<Bit_size>::lookup(uint32_t phase_32bit)
{
    constexpr int shift = 32 - Bit_size;
    uint32_t idx = phase_32bit >> shift;
    return table[idx];
}
```

Further Reading

Band-limited Step Functions (BLEP) (Brandt 2001, Leary & Bright 2009)

Non-linear Digital Implementation of the Moog Ladder Filter (Huovilainen 2004)

Natural Sounding Artificial Reverberations (Schroeder 1962)

Introduction to Digital Filters with Audio Applications (JOS 2007)

Steinberg VST Plugin Wrapper

```
void VstXSynth::setParameter (VstInt32 index, float value01)
{
    float min, max, exp;
    value01 = clamp01(value01);
    Plugin_info::get_parameter_range(index, min, max, exp);
    synth.set_parameter(index, lerp_inline(min, max, value01), -1);
}

float VstXSynth::getParameter (VstInt32 index) {
    float min, max, exp;
    Plugin_info::get_parameter_range(index, min, max, exp);
    float value = synth.get_parameter(index);
    float value01 = inverse_lerp(value, min, max);
    return value01;
}
```

```
void VstXSynth::processReplacing(
    float** inputs, float** outputs, VstInt32 sample_frames )
{
    float* out1 = outputs[0]; // out1 = left channel
    float* out2 = outputs[1]; // out2 = right channel

    interleave_buffer(out1, out2, buf_tmp, sample_frames);
    synth.render_float32_stereo_interleaved(buf_tmp, sample_frames, 0u);
    deinterleave_buffer(buf_tmp, out1, out2, sample_frames);
}
```

Unity Native Audio Plugin Wrapper

```cpp
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ProcessCallback(UnityAudioEffectState* state,
    float* inbuffer, float* outbuffer, unsigned int length, int inchannels, int outchannels)
{
    EffectData::Data* data = &state->GetEffectData<EffectData>()->data;

    // ...

    bool isPlaying = true;
    bool isMuted = ((state->flags & UnityAudioEffectStateFlags::UnityAudioEffectStateFlags_IsMuted) != 0);

    if (isPlaying && (!isMuted))
    {
        uint64_t clock_smp = state->currdsptick;
        data->synth.render_float32_stereo_interleaved(outbuffer, length, clock_smp);
    }
    else
    {
        // Silence
        memset(outbuffer, 0, sizeof(float) * 2 * length);
    }

    return UNITY_AUDIODSP_OK;
}
```

From Bitwig Prototype to FMOD Plugin

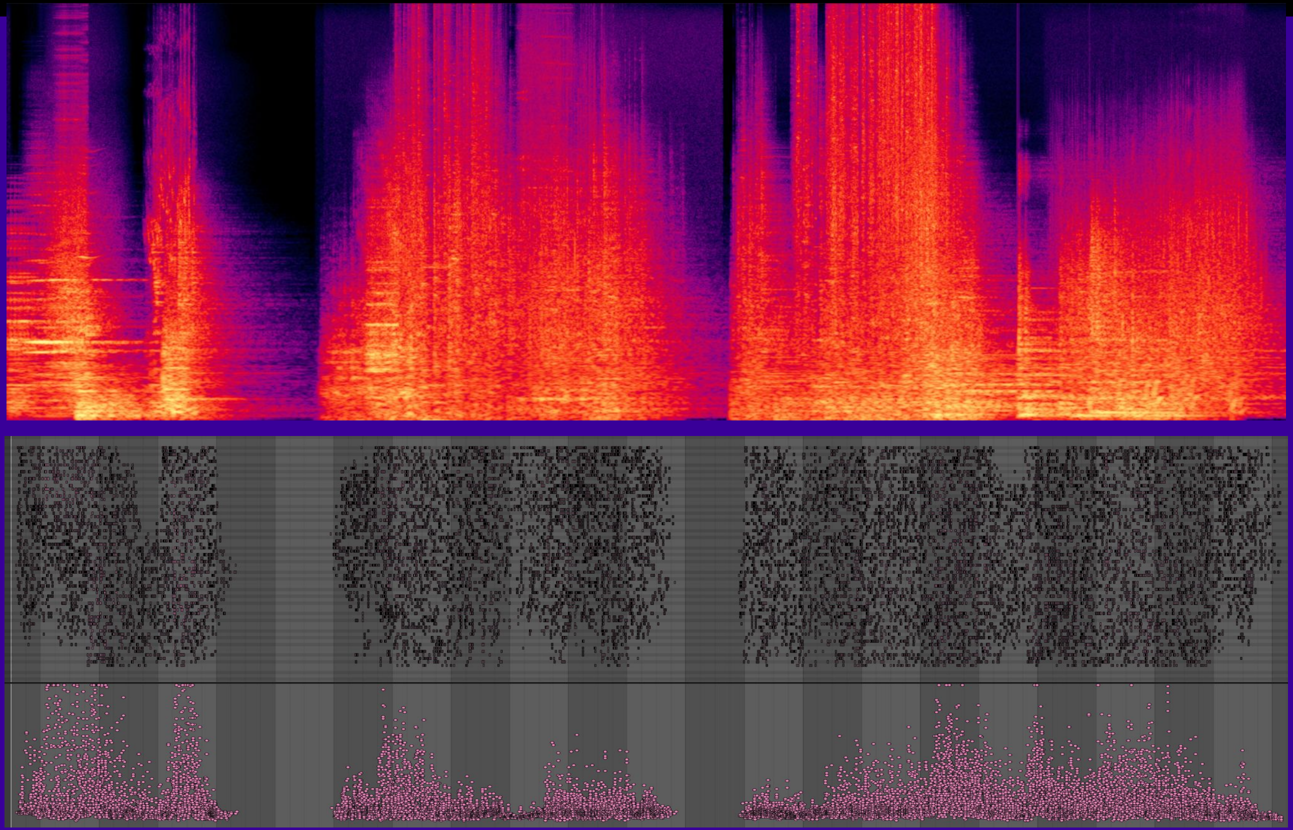
```
FMOD_DSP_DESCRIPTION Plugin_FMOD_Desc =
{
    FMOD_PLUGIN_SDK_VERSION,
    "",                             // name (32 chars) (filled in by FMODGetDSPDescription)
    Plugin_info::get_version(),     // plug-in version
    0,                              // Number of input buffers to process
    1,                              // Number of output buffers to process
    Plugin_FMOD_dspcreate,
    Plugin_FMOD_dsprelease,
    Plugin_FMOD_dspreset,
    0,                              // read callback
    Plugin_FMOD_dspprocess,
    0,                              // set position callback
    -1,                             // param count, set in FMODGetDSPDescription
    Plugin_FMOD_dspparam_ptrs,      // param descriptions
    Plugin_FMOD_dspsetparamfloat,
    Plugin_FMOD_dspsetparamint,
    Plugin_FMOD_dspsetparambool,
    Plugin_FMOD_dspsetparamdata,
    Plugin_FMOD_dspgetparamfloat,
    Plugin_FMOD_dspgetparamint,
    Plugin_FMOD_dspgetparambool,
    Plugin_FMOD_dspgetparamdata,
    0,
    0,                              // userdata
    0,                              // Register
    0,                              // Deregister
    0                               // Mix
};
```

```
FMOD_RESULT F_CALLBACK Plugin_FMOD_dspprocess(
    FMOD_DSP_STATE *dsp, unsigned int length,
    const FMOD_DSP_BUFFER_ARRAY * inbufferarray, FMOD_DSP_BUFFER_ARRAY *outbufferarray,
    FMOD_BOOL inputsidle, FMOD_DSP_PROCESS_OPERATION op)
{
    PluginFMODState *state = (PluginFMODState *)dsp->plugindata;

    // ...

    if (op == FMOD_DSP_PROCESS_PERFORM)
    {
        // Get clock from FMOD.
        unsigned long long clock; // event clock (smp)
        unsigned int offset;      // where does event start in input buffer?
        unsigned int length;      // when does event stop in input buffer?
        FMOD_DSP_GETCLOCK(dsp, &clock, &offset, &length);

        // Render
        state->synth.render_float32_stereo_interleaved(outbufferarray->buffers[0], length, clock);
    }

    return FMOD_OK;
}
```

Translate More Components to C++


```cpp
class Mod_delay
{
private:
    Circbuf buf0, buf1;
    float max_delay__s;
    float current_delay__s = 0;
    float target_delay__s = 0;
    float current_input_scale = 0;
    float target_input_scale = 0;
    float smoothness__s_p_smp = 0.01f;
    float feedback = 0.0f;
    float current_dry = 0;
    float current_wet = 0;
    int sample_rate;

public:
    void reallocate(float max_delay__s, int sample_rate);
    void clear_state();
    void set_feedback(float feedback01) { this->feedback = feedback01; }
    float get_feedback() { return feedback; }
    // smoothness is measured in delay time (s) per second
    void set_smoothness(float smoothness);
    void set_delay(float delay__s);
    void set_delay_instantaneous(float delay__s);
    void set_input_level(float input_level01);
    void set_input_level_instantaneous(float input_level01);
    float get_delay() const;
    float render_single_mono(float input);
    void render_float32_mono(float* buffer, int32_t sample_frames);
    void render_float32_stereo_interleaved(float* buffer, int32_t sample_frames);
    void render_float32_stereo_interleaved_additive(float* buffer, int32_t sample_frames,
        float gain_dry, float gain_wet);
};
```

MIDI Vocoder: Dyson Gate

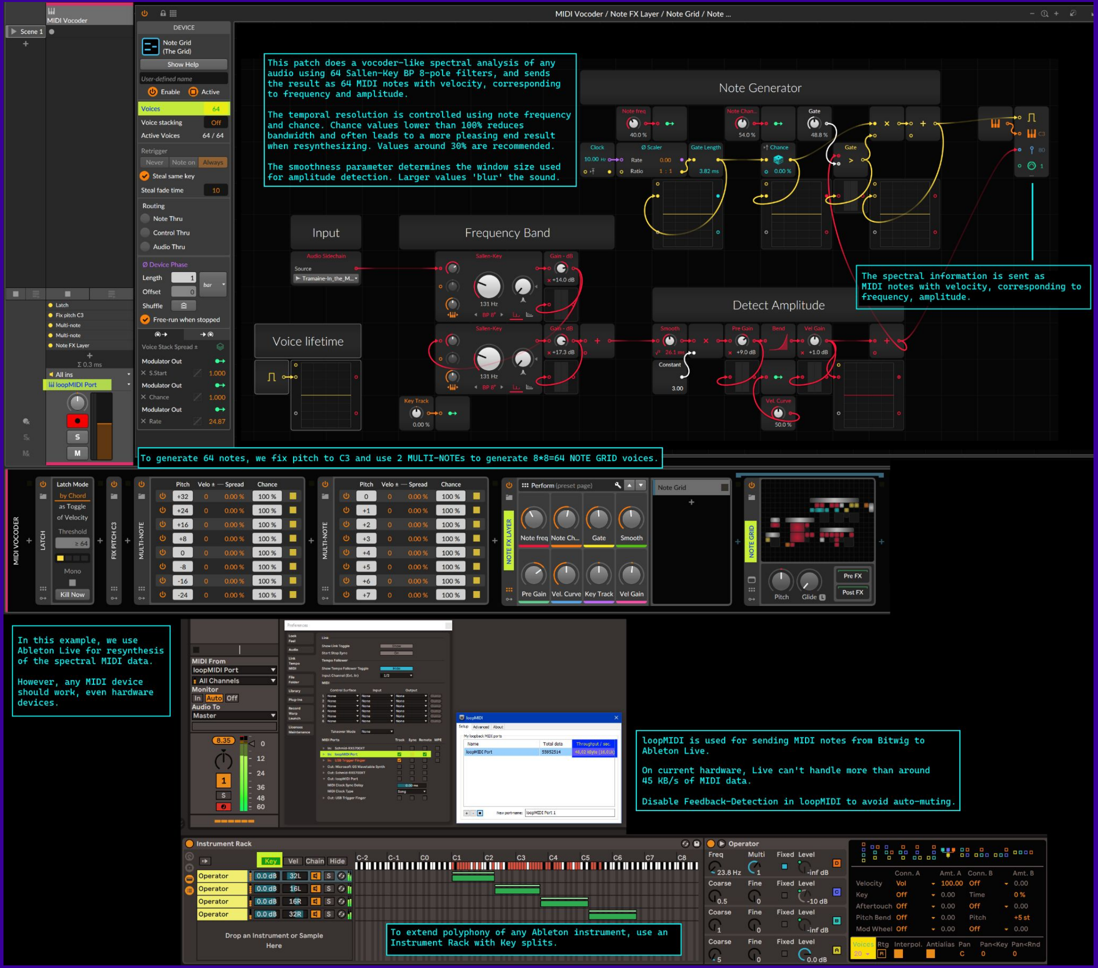


▸ midi_vocoder-bitwig, midi_vocoder-ableton, cocoon-gate

MIDI Vocoder

Home-made vocoder

- Bitwig audio analysis
- MIDI sent via loopMIDI
- Record MIDI in Ableton Live



Puzzle Feedback Music

▸ cocoon-puzzfeed

MIDI Vocoder: Puzzle Feedback

Inspiration

Quite & Orange: CDAK (2010)

Music by Lassi Nikko

4K demo