

COCCON

Develop:Brighton 2024  
Jakob Schmid  
Geometric Interactive

# Who am I?

Computer scientist from Aalborg University

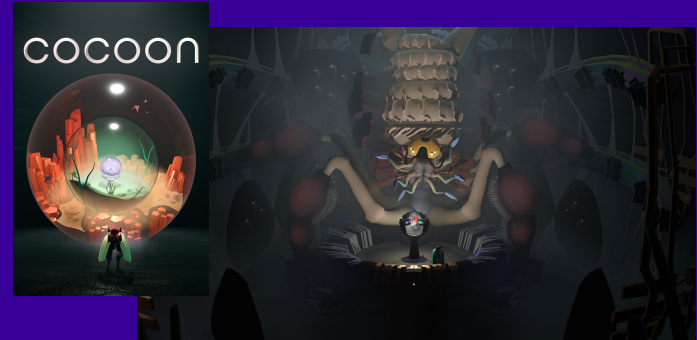
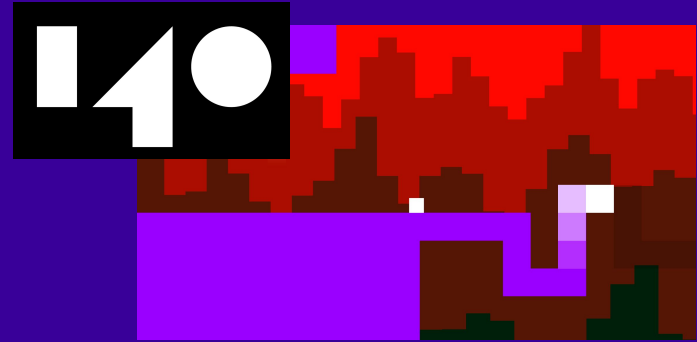
16 years experience in game development

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

# COCOON

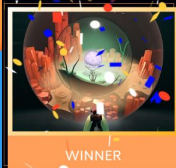


# Some People Seem to Like It



## BEST DEBUT INDIE GAME

For the best debut game created by a new independent studio.



**COCOON**  
Geometric Interactive/Annapurna Interactive

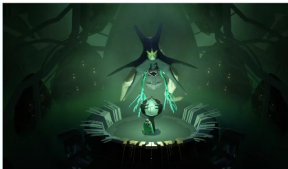


## OUTSTANDING ACHIEVEMENT FOR AN INDEPENDENT GAME



## Cocoon is Eurogamer's game of 2023

What is a great game made of?



# COCOON Audio Team

Audio direction / music:

Jakob Schmid

Sound design:

Julian Lentz

Mikkel Anttila



# Topics

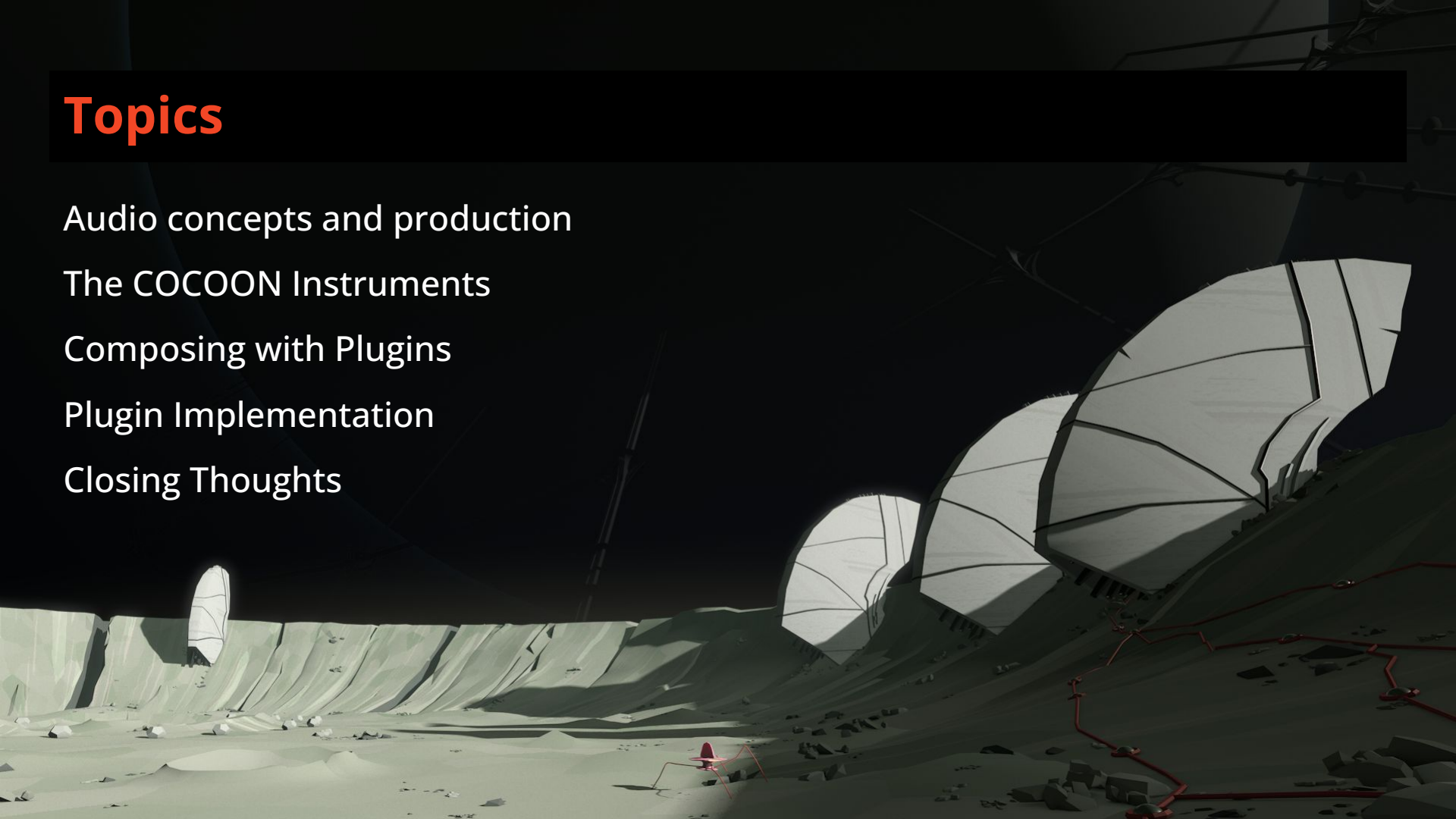
Audio concepts and production

The COCOON Instruments

Composing with Plugins

Plugin Implementation

Closing Thoughts



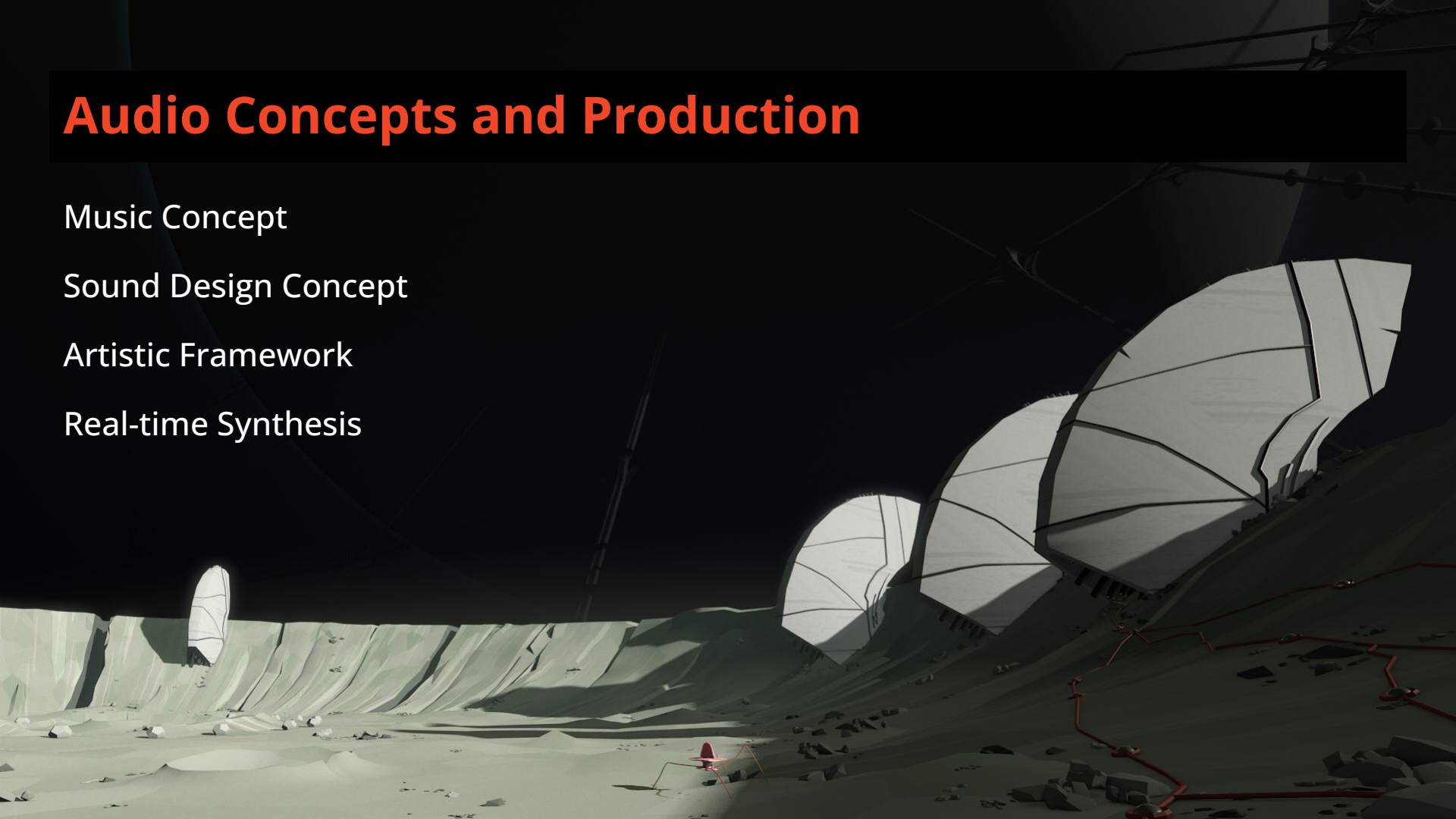
# Audio Concepts and Production

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*



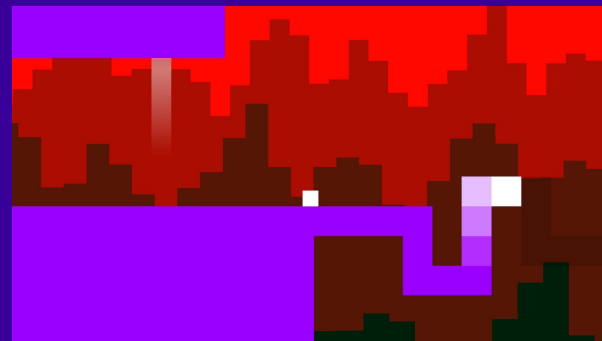
*Puzzle gameplay: synthesized ambient music*



# 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'

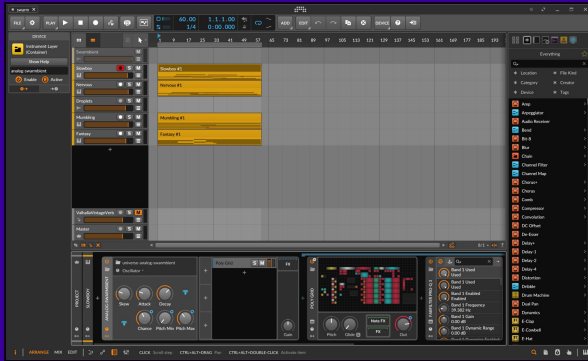


# Music Software

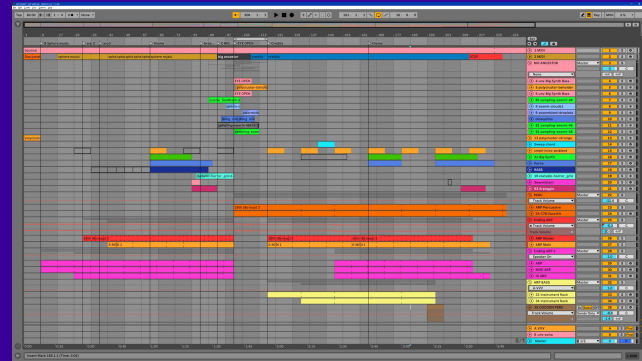
Bitwig Studio and Ableton Live was used for music production and sound design

**Ableton Live** designing sounds, especially the unique Corpus plugin

**Bitwig Studio** developing new synthesizers, sophisticated automation



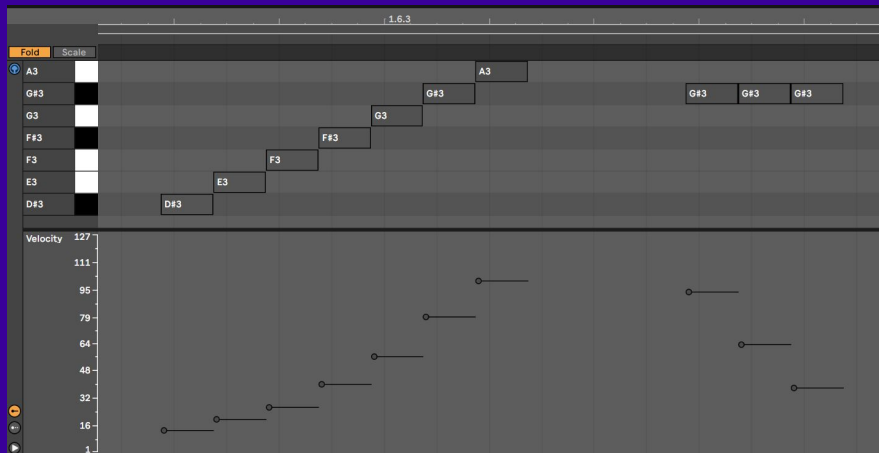
*Bitwig Studio 5*



*Ableton Live 11*

# Synthetic Sound Design Experiments

Frogs, footsteps, portals



The screenshot shows the Ableton Live software interface with several audio processing modules. The Operator module is active, displaying a sine wave envelope and oscillator settings. The EQ Eight module is also active, showing a frequency response curve. Other modules like Auto Pan, Delay, and Ping Pong are visible.

# 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** -

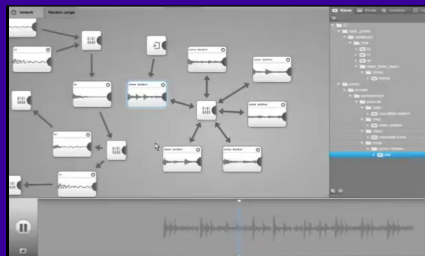
Graph-based game music middleware

**140** -

Music systems for Jeppe Carlsen's music platformer

**Rytmos** -

DSP plugins for Floppy Club's puzzle game



*Audioflow (2010)*



*Lost Empire: Immortals (2008)*



*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 Sega Genesis music routine (WIP)

Defender Defender arcade machine sound board emulator



Acorn Electron



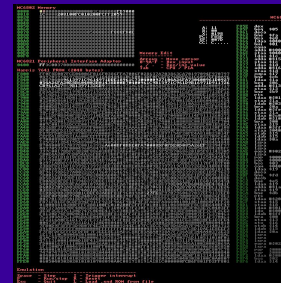
ProTracker (Amiga)



AlgoTracker (Pico-8)



AlgoTracker (Mega Drive)



DefendEmu

# The COCOON Instruments

BOB

K88

Modnet

Weather



# COCOON Instruments

**K88**

Schmid | K88  
1.0.0 2023-08-13

Sampler On  Voice Count 3

Grain Size (ms) 5.00

BANK OFFSET  
Offset (s) 45%  
Fine Offset (s) 0%

ORCHESTRA  
Voice Spread 14.5k  
Random offset 665

SWARM  
Note Freq 4.30  
Note Chance 0%  
Pitch min -8.00  
Pitch max 28.0  
Scale 5.80  
Vibrato 5.90

Automatable LPF Freq 11.1k Volume 100%

Gain 2.00 Reverb Decay 61%

Debug Dump  MODE  
Orchestra  
Swarm

Octave 4.40 Semitone 0.00 Detune -26%

Offset Modulation Frequency 0.16 Amount 39% Smoothness 87%

Delay 45% Time 36% Feedback  
Base Freq. 0.01 Strength 0.00

Delay Mod

**Modnet**

Schmid | Modnet  
1.0.0 2023-08-13

Operator Count 16  
Quality 3

Debug Dump

Alg A Waving Chord 100% 100% 2.60 4.40 0% 0.67

Alg B Noise 35% 47% 2.40 0.00 0% 0.70

Param 0 Param 1 Octave Semitone Detune Amp

Morph Morph Mod Morph Easing LPF freq HPF freq

21% 0.01 str 12.0k 65.0

3 0.04

**Weather**

Schmid | Weather  
1.0.0 2023-08-13

Debug Dump

Oscillator Sine 4.00 100% 380

Grain freq Spread Base freq L Freq R Freq Str

FLFO 0.00 0.02 0.42

Pitch Quantize  Min freq Max freq Base-Q L Freq R Freq Str

50.0 4.80k 0.86 0.02 0.03 0.00

QLFQ

Volume 20%

**BOB**

Schmid | BOB  
1.0.0 2023-08-13

Debug Dump

ARPEGGIATEUR  
Enabled  Scale 11 Loop 8.00 Ping-pong  Random  Note Chan... 100%

Pattern Length Multiply Jump

Pattern threes 8.00 10 0.00

Tempo BPM 40.0 Subdivision 8.00 Gate 33% Offset 0%

Pitch -24.0 Amp 0%

Transpose Octave 1.00 Semitone -2.00 Square 0.00 Saw 19.0 Sine 0.00 Freq 1.80 Str 0%

PLFO

Filter Cutoff -2.58k Key Track 0.70 FENV amt 2.30k Resonance 0.62 FENV Attack 0.02 Decay 3.00 Sustain 66% Release 0.58

OSC amp Square 52% Saw 0% Sine 52% PWM Freq 0.02 Str 49%

AENV Attack 0.02 Decay 2.40 Sustain 0% Release 0.58

# 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

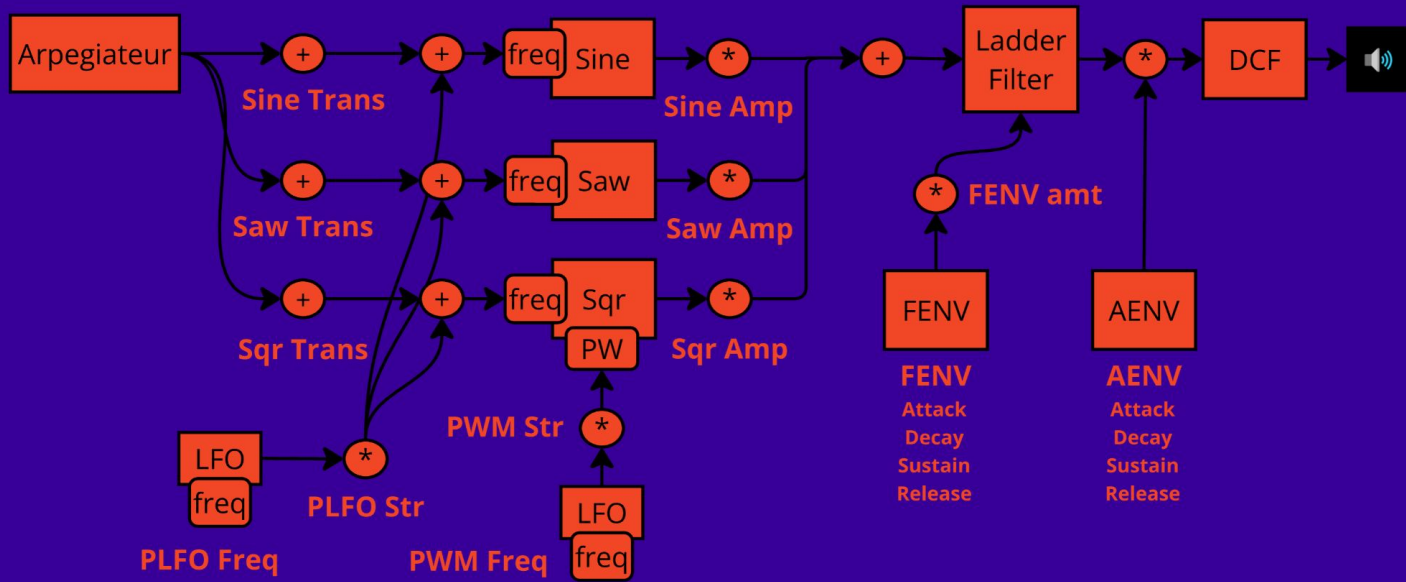
Pulse-width modulation of square wave (also, vibrato)

Ladder filter for resonant filtering

Amplitude and Filter Envelopes



# BOB Structure



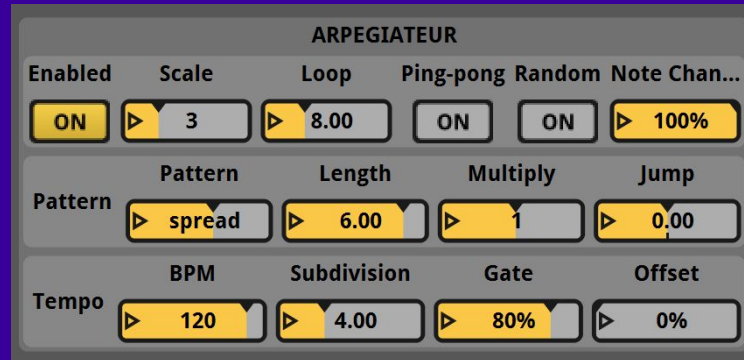
The screenshot shows the BOB synthesizer's software interface. The **ARPEGGIATEUR** section is active, showing parameters like **Scale** (11), **Loop** (8.00), **Ping-pong** (ON), **Random** (ON), and **Note Chan...** (100%). The **Pattern** is set to **threes**, **Length** is 8.00, **Multiply** is 10, and **Jump** is 0.00. The **Tempo** is 40.0, **BPM** is 8.00, **Subdivision** is 33%, and **Gate** is 0%. The **Pitch** section shows **Transpose** at -24.0 and **Amp** at 0%. The **Oscillator** section shows **Square** at 52%, **Saw** at 0%, **Sine** at 52%, **Freq** at 0.02, and **Str** at 49%. The **Filter** section shows **Cutoff** at -2.58k, **Key Track** at 0.70, **FENV amt** at 2.30k, **Resonance** at 0.62, **Attack** at 0.02, **Decay** at 3.00, **Sustain** at 66%, and **Release** at 0.58. The **Automation & Modulation** section shows **FENV** parameters (Attack, Decay, Sustain, Release) and **AENV** parameters (Attack, Decay, Sustain, Release).

# 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  
Prime



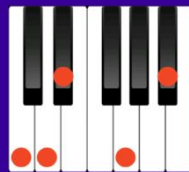
Scale 2  
Fifth



Scale 3  
Minor



Scale 4  
Minor 7



Scale 5  
Minor 9



Scale 6  
Minor 11

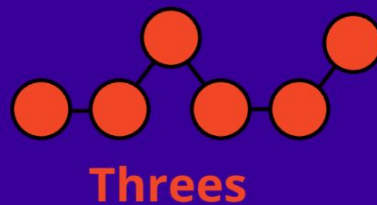
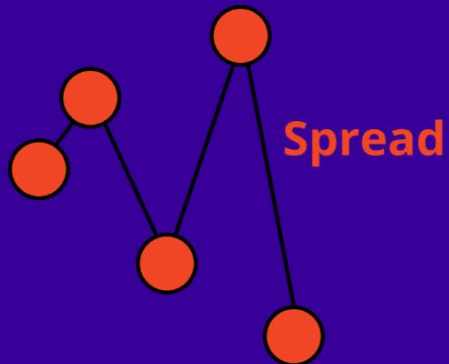
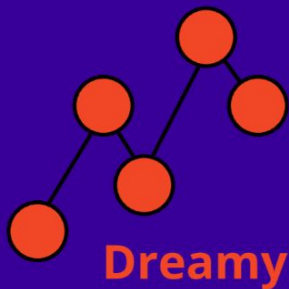
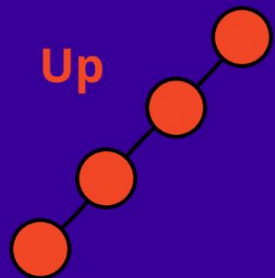


Scale 7  
Minor 13

ARPEGGIATEUR

Enabled	Scale	Loop	Ping-pong	Random	Note Chan...
ON	3	8.00	ON	ON	100%
Pattern	Pattern	Length	Multiply	Jump	
	spread	6.00	1	1.00	
Tempo	BPM	Subdivision	Gate	Offset	
	120	4.00	80%	0%	

# Arpeggiator: Pattern

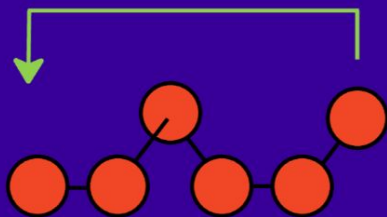


ARPEGGIATEUR

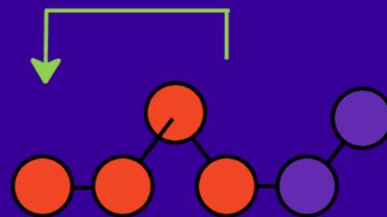
Enabled	Scale	Loop	Ping-pong	Random	Note Chan...
<input checked="" type="checkbox"/>	3	8.00	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	100%
Pattern	Length	Multiply	Jump		
spread	6.00	1	0.00		
Tempo	BPM	Subdivision	Gate	Offset	
	120	4.00	80%	0%	



# Arpeggiator: Length



Length:6

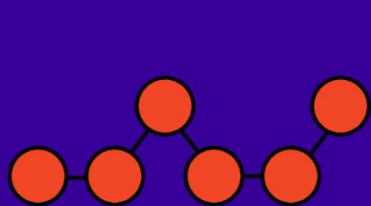


Length:4

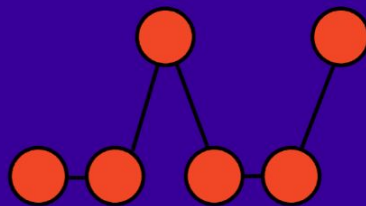
ARPEGGIATEUR

Enabled	Scale	Loop	Ping-pong	Random	Note Chan...
<input checked="" type="checkbox"/>	3	8.00	<input type="checkbox"/>	<input type="checkbox"/>	100%
Pattern	Pattern	Length	Multiply	Jump	
	spread	6.00	1	0.00	
Tempo	BPM	Subdivision	Gate	Offset	
	120	4.00	80%	0%	

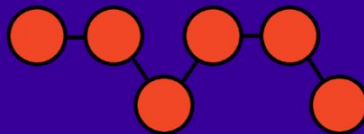
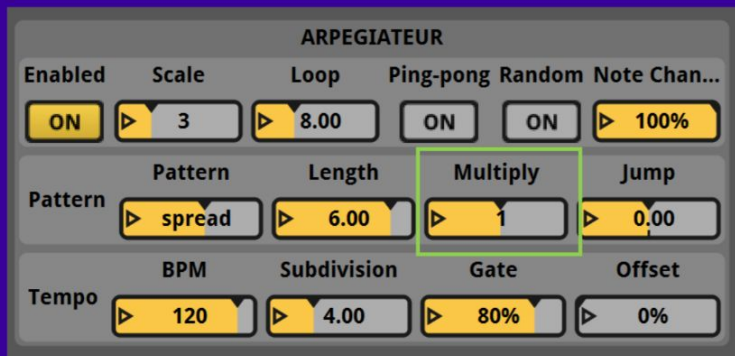
# Arpeggiator: Multiply



Multiply: 1

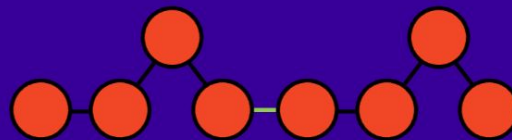


Multiply: 2

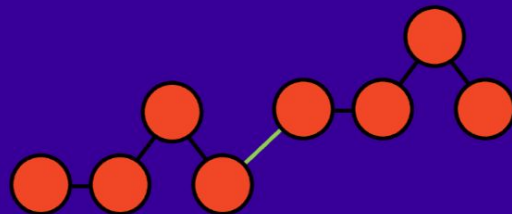


Multiply: -1

# Arpeggiator: Jump



Jump:0



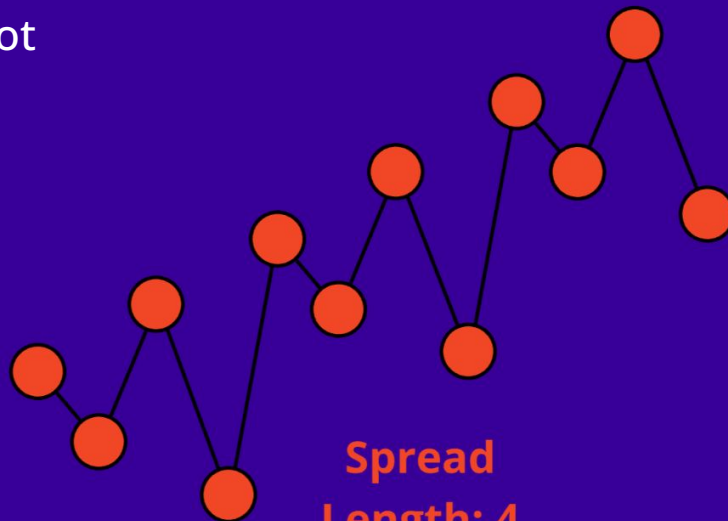
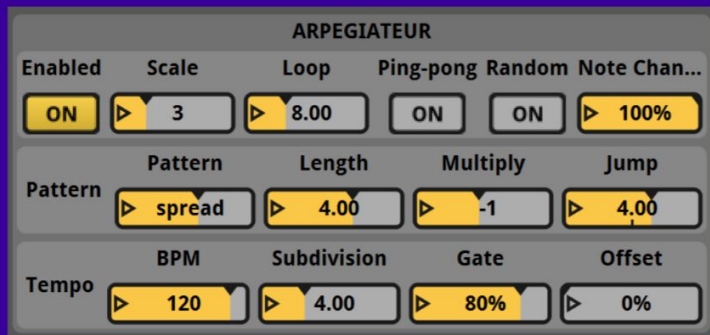
Jump:1

ARPEGGIATEUR

Enabled	Scale	Loop	Ping-pong	Random	Note Chan...
<input checked="" type="checkbox"/>	3	8.00	<input type="checkbox"/>	<input type="checkbox"/>	100%
Pattern	Pattern	Length	Multiply	Jump	
	spread	6.00	1	1.00	
Tempo	BPM	Subdivision	Gate	Offset	
	120	4.00	80%	0%	

# 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

Bitwig Studio Prototype developed in Bitwig Studio first, then reimplemented in C++



# 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

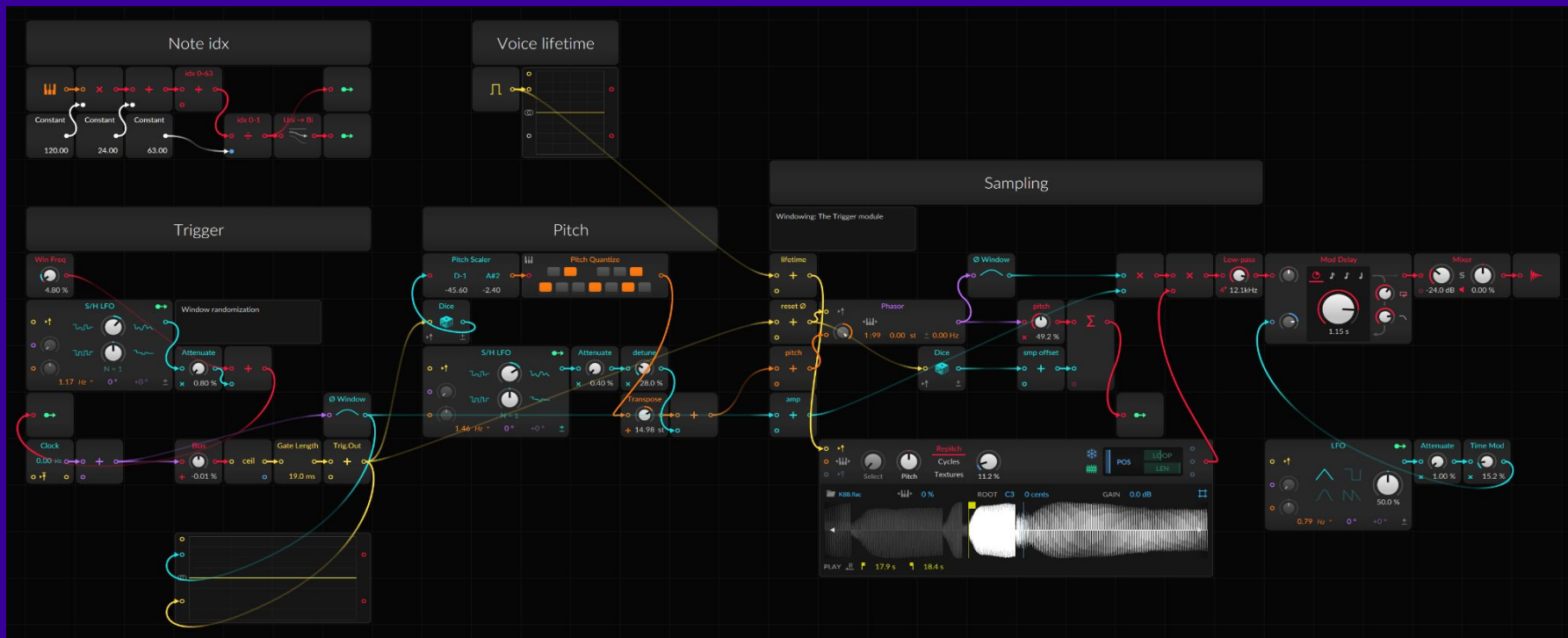
Modulation delay Separate delay for each voice

- Gives each voice uniqueness, enlarging total sound

Low pass filter on output to remove unwanted high frequency artifacts



# K88: Swarm Mode



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

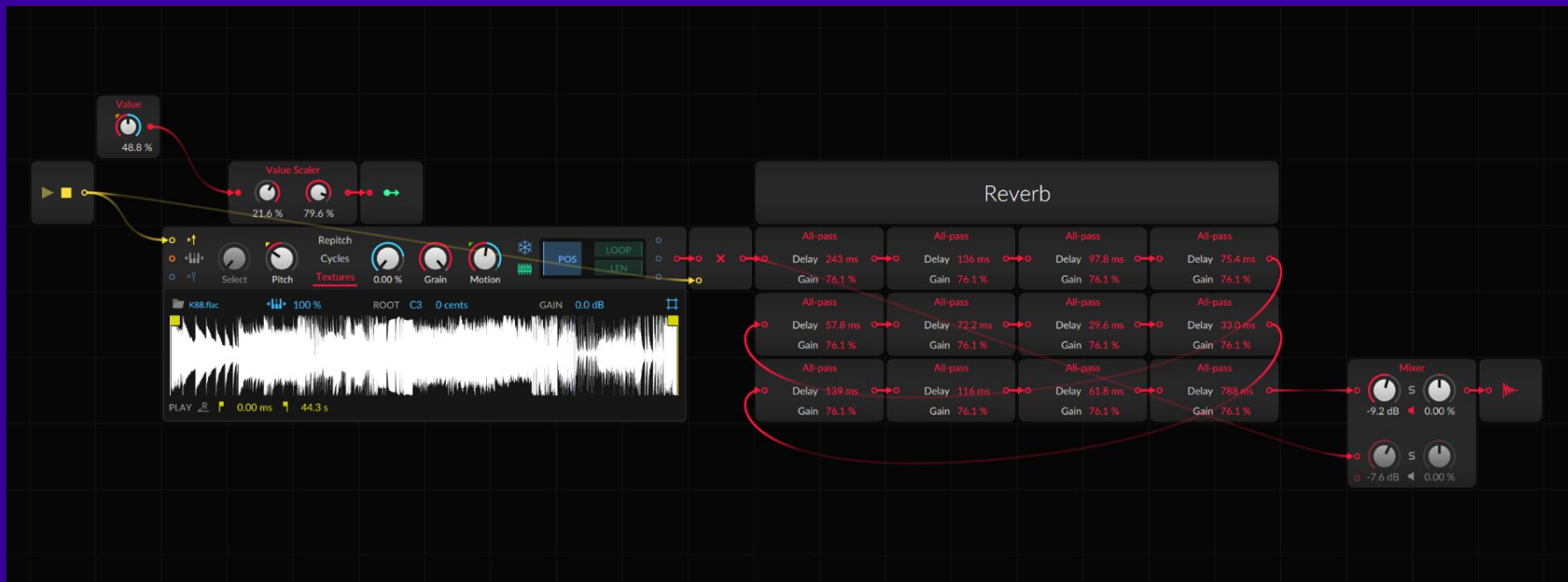
Random offset to each grain avoids robotic quality

Atonal orchestral sound works well for horror sequences





# K88: Orchestra Mode



Bitwig Grid patch



# Modnet

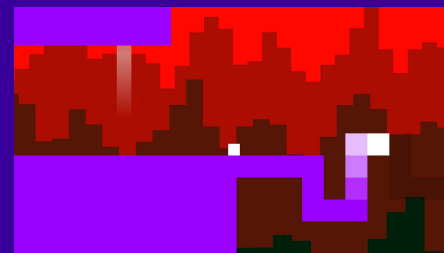
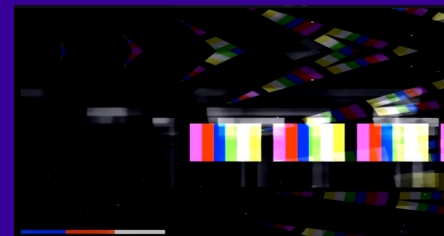
FM/AM synthesizer with 16 operators

Interpolates between two configurations

Brass-like sounds used to dramatic effect in COCOON

Developed in 2013 for a live performance, but couldn't run realtime back then.

Is also used on the '140' soundtrack (0x8c)



# Modnet

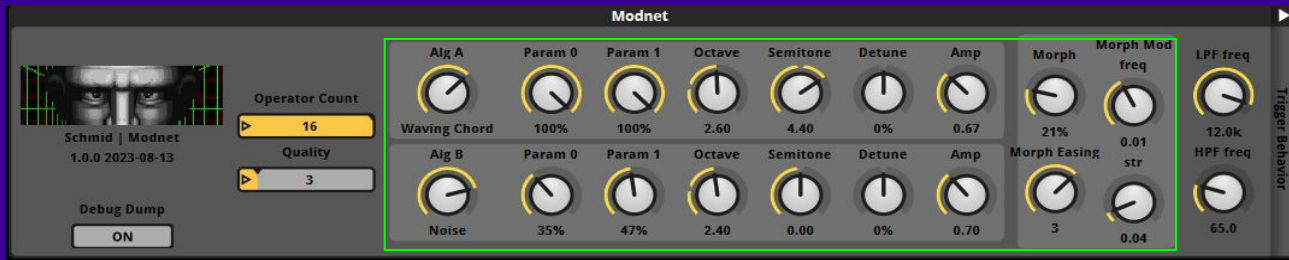
A meta-algorithm and a few parameters generates a patch (10 meta-algorithms)

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

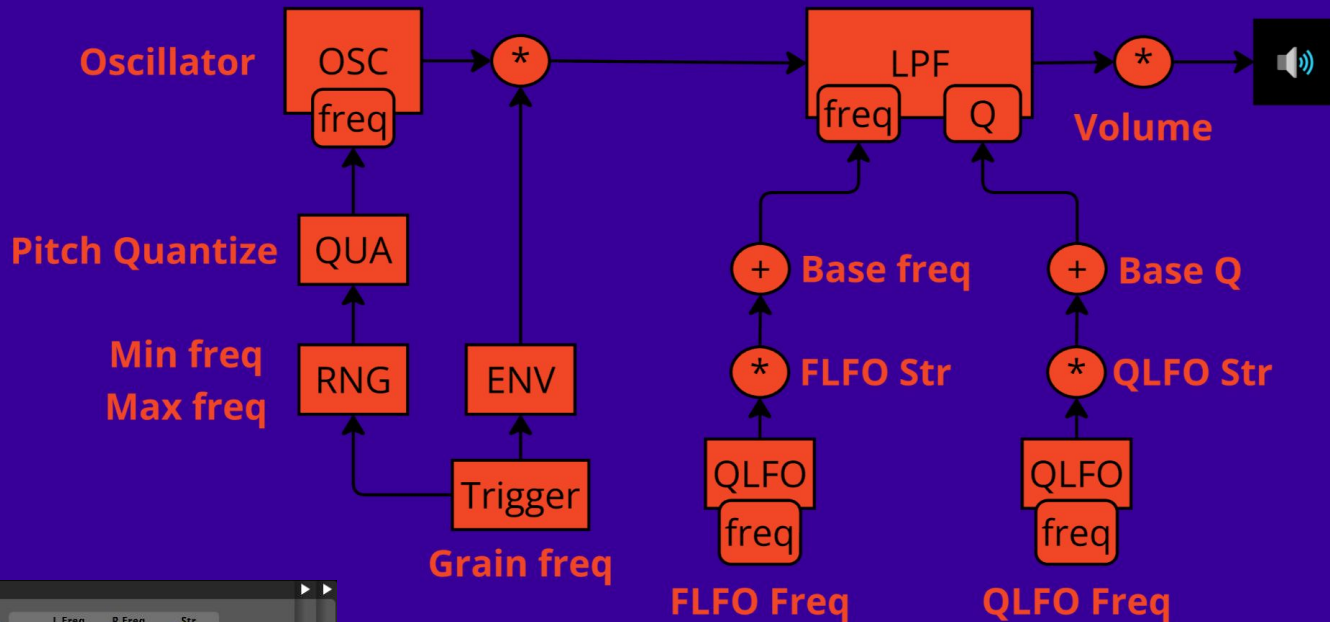
Dual resonant filters left and right channel

Four LFOs controlling filter cutoff and resonance



# Weather Structure

Duplicated for each stereo channel



# Used for Ambience and Music



# Composing with Plugins

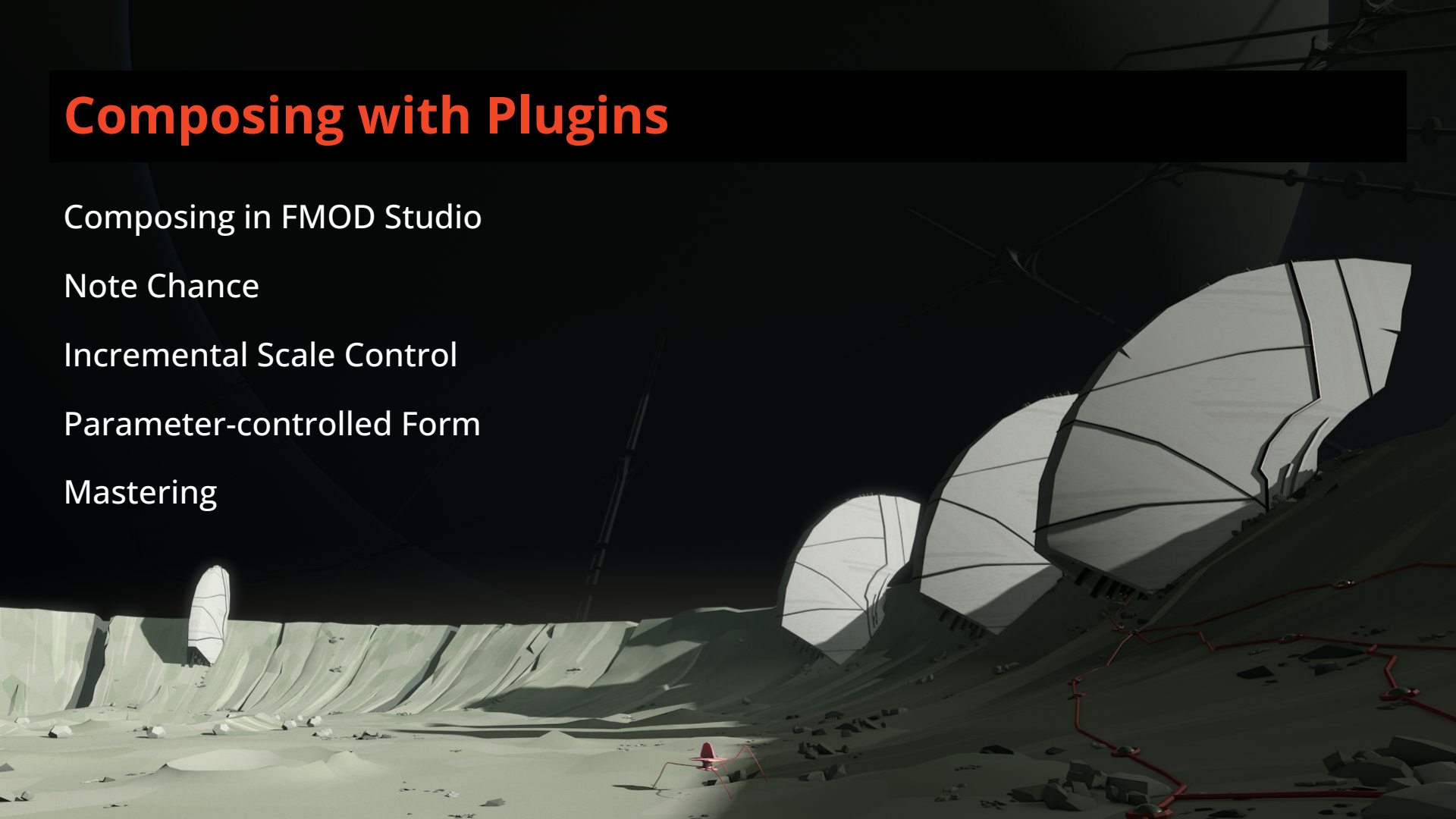
Composing in FMOD Studio

Note Chance

Incremental Scale Control

Parameter-controlled Form

Mastering



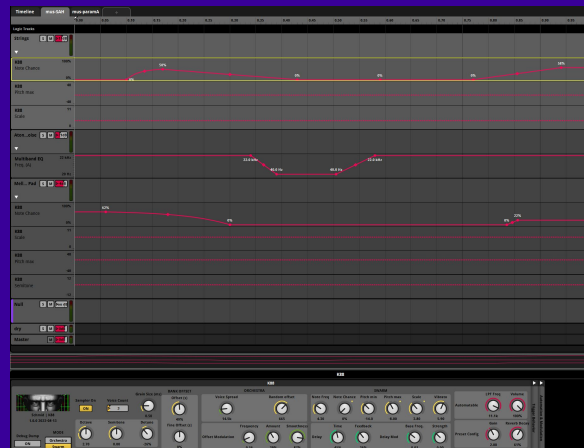
# Composing in FMOD Studio

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



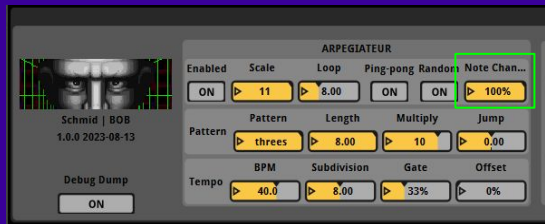
# Note Chance

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



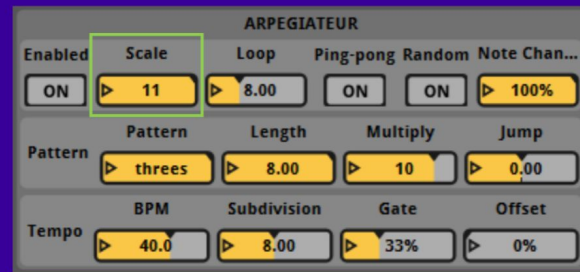
# Incremental Scale Control

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  
Prime



Scale 2  
Fifth



Scale 3  
Minor



Scale 4  
Minor 7



Scale 5  
Minor 9



Scale 6  
Minor 11



Scale 7  
Minor 13

# Modulation Problems

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

Feels very dynamic and organic

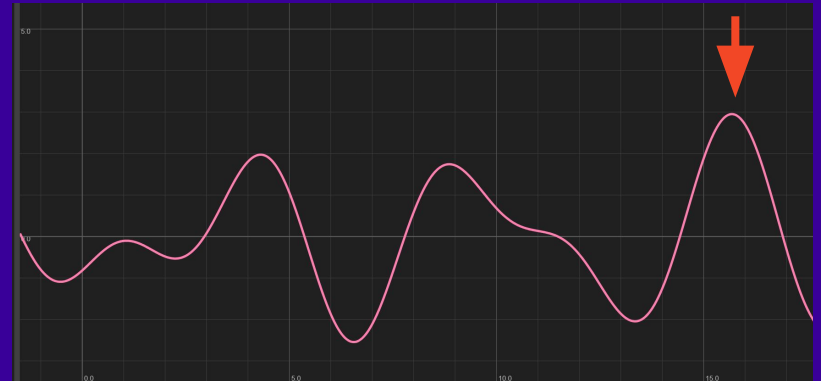


# Constructive Interference

Combinations of modulators can produce unexpected results

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

Worst case, could cause clipping



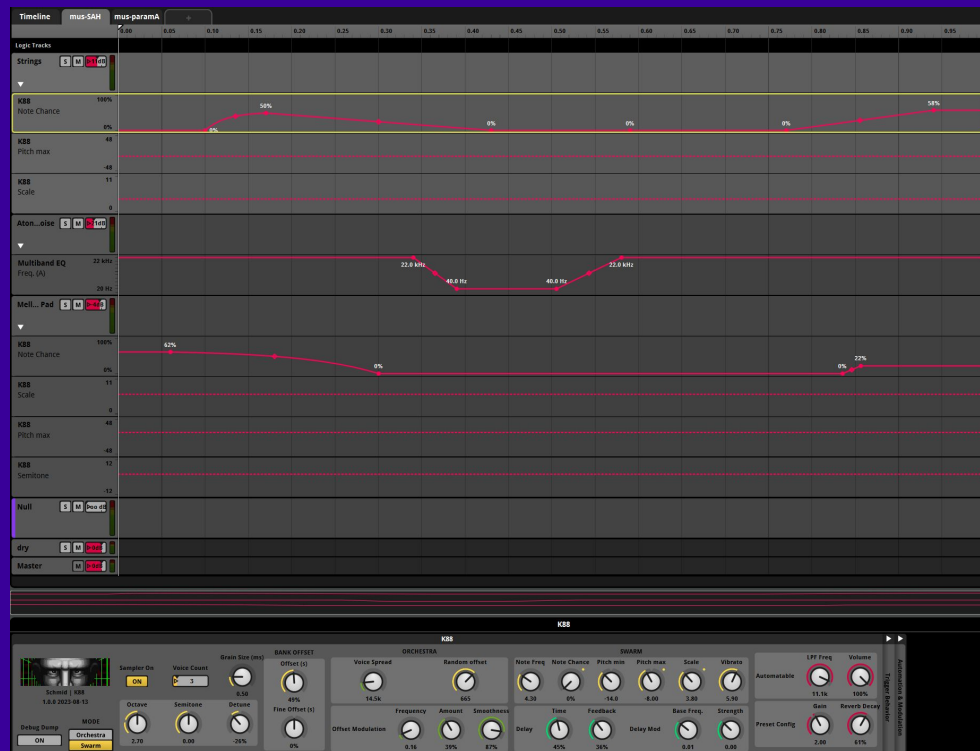
# Parameter-controlled Form

## 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

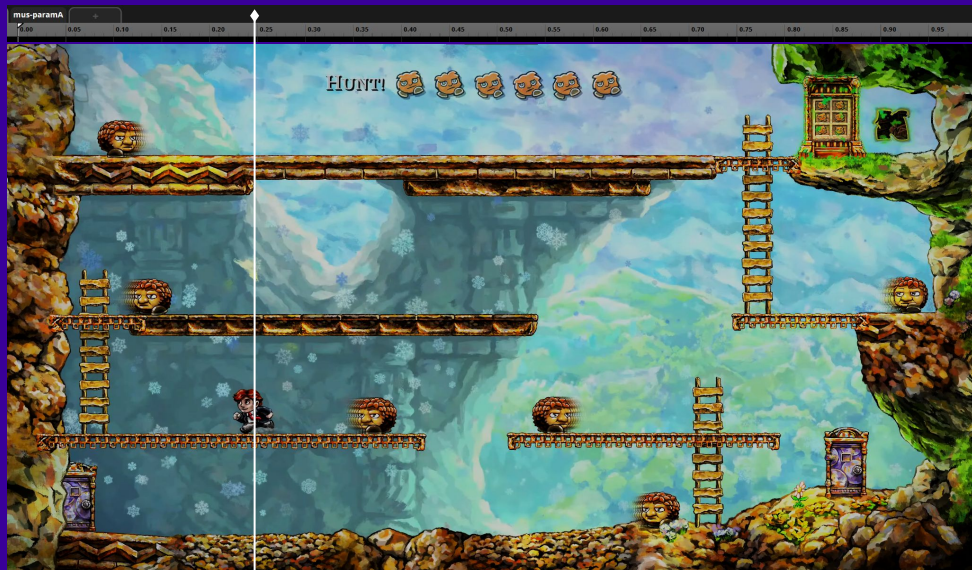
Control using a single parameter



# 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



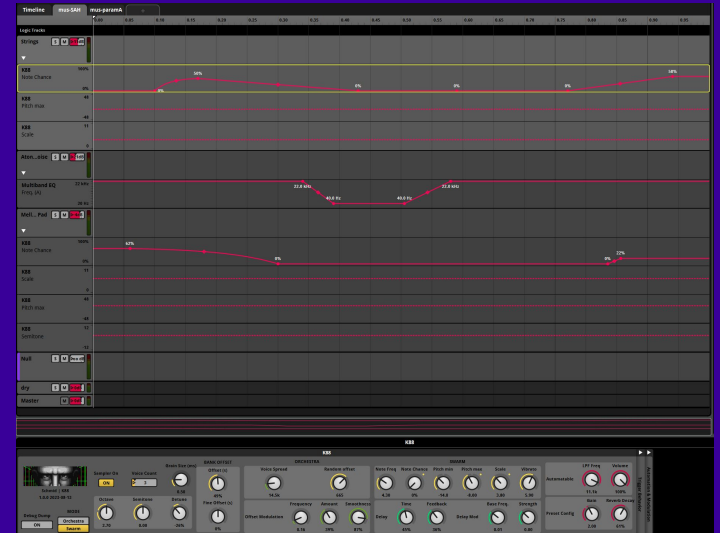
# Parameter-controlled Form

**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



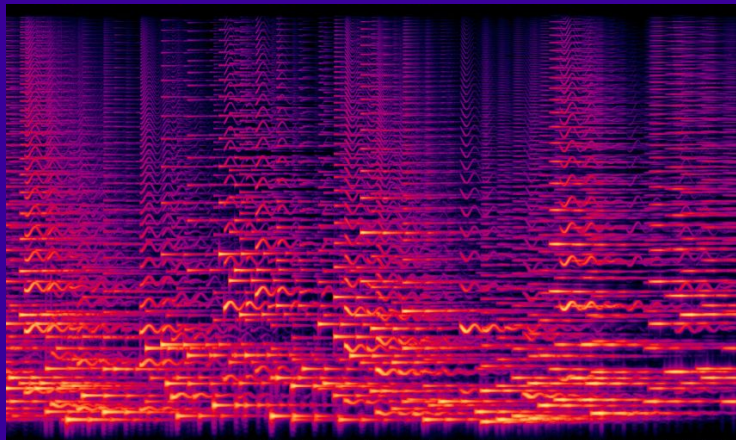


# Mastering

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



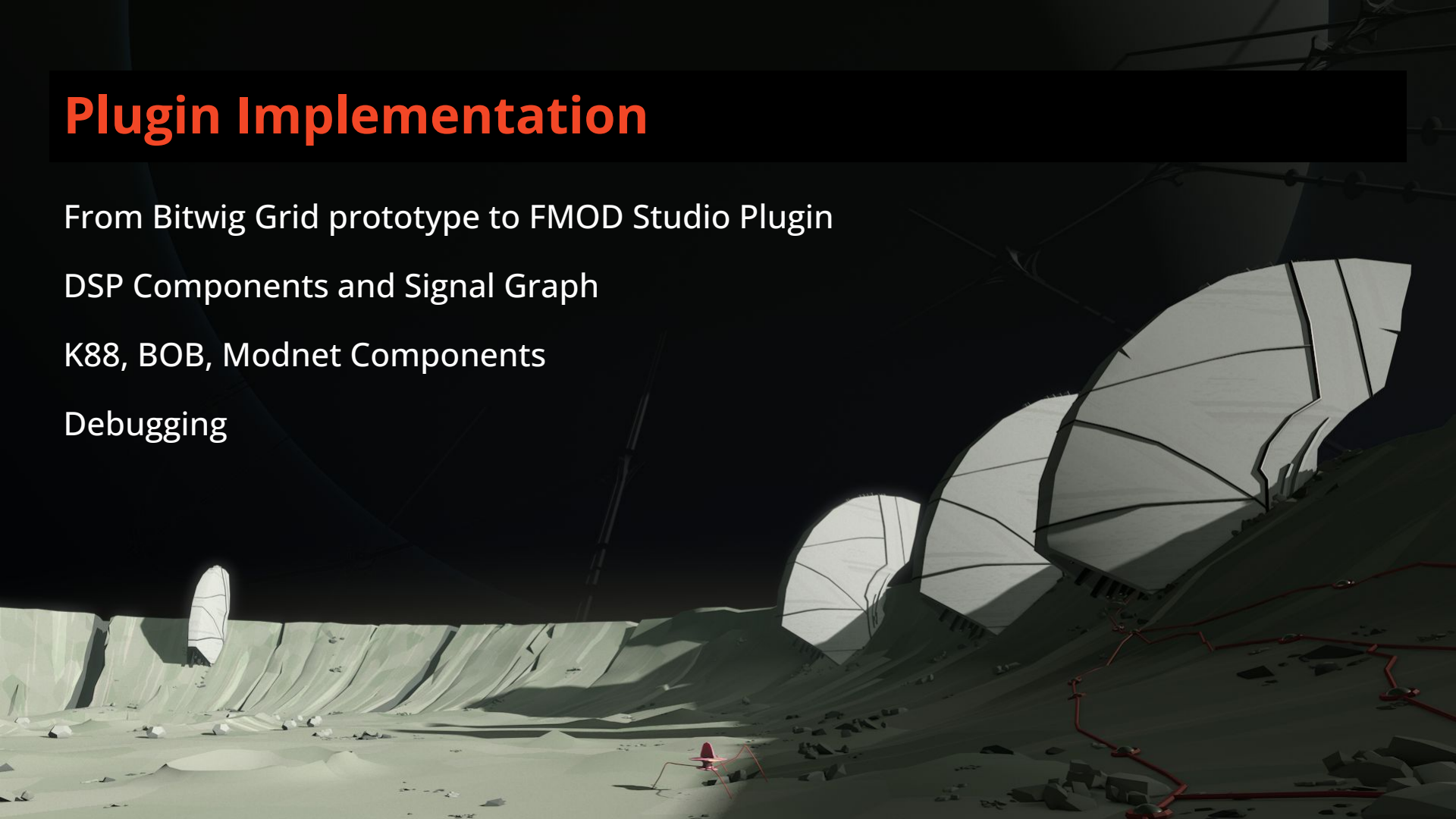
# Plugin Implementation

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

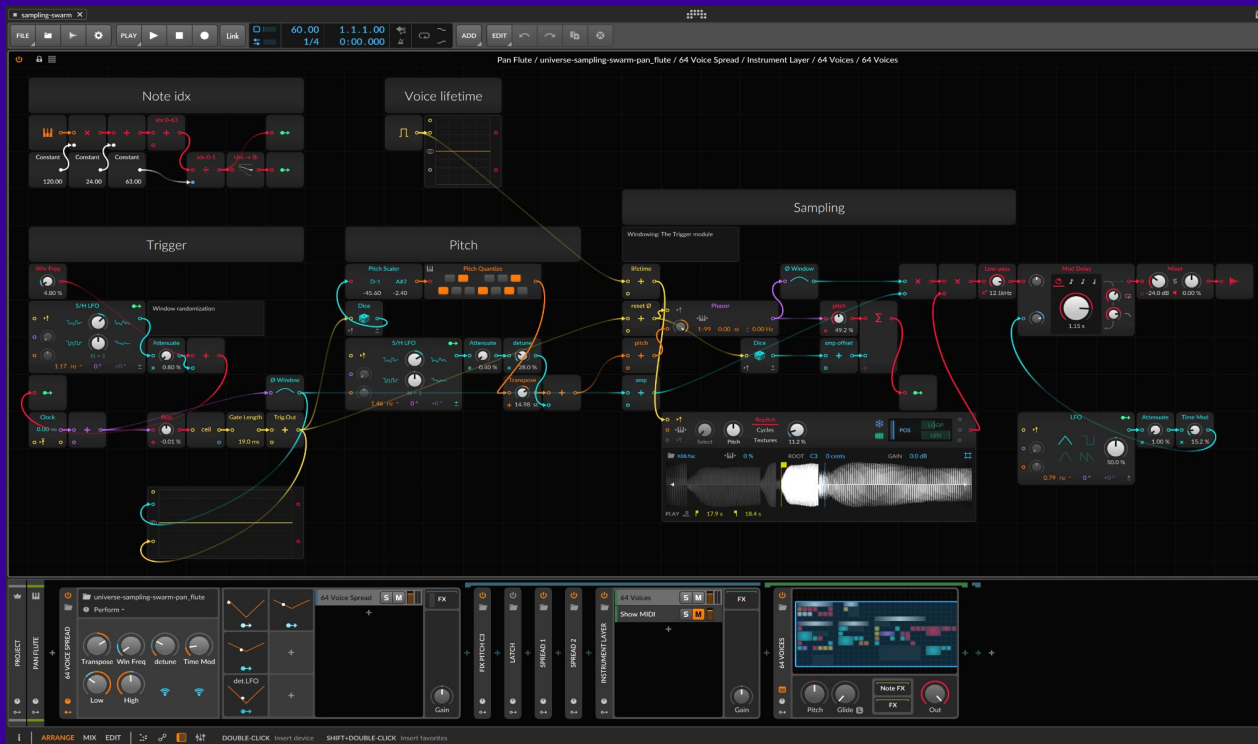
FMOD Studio plugin API is open

Plugins are normally written in C++

Start with example project and modify



# From Bitwig Grid Prototype to Plugin



Bitwig Grid patch

# DSP Components

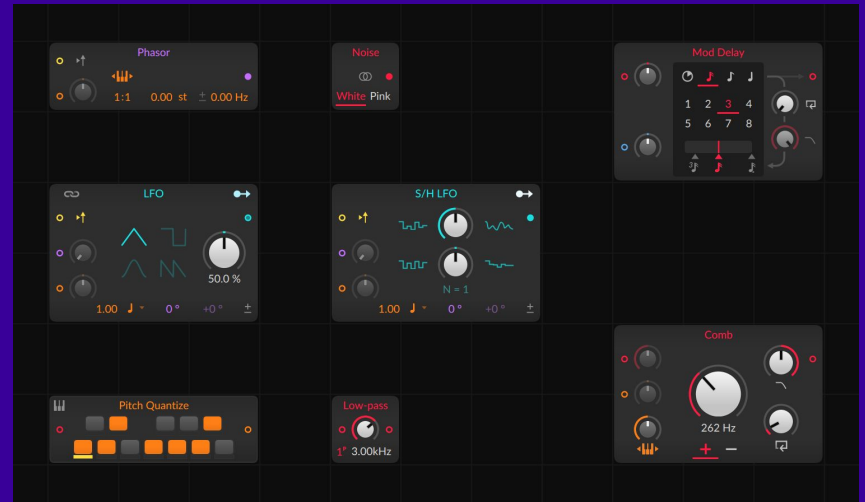
A Bitwig Grid patch can be expressed as a graph of DSP nodes.

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)*

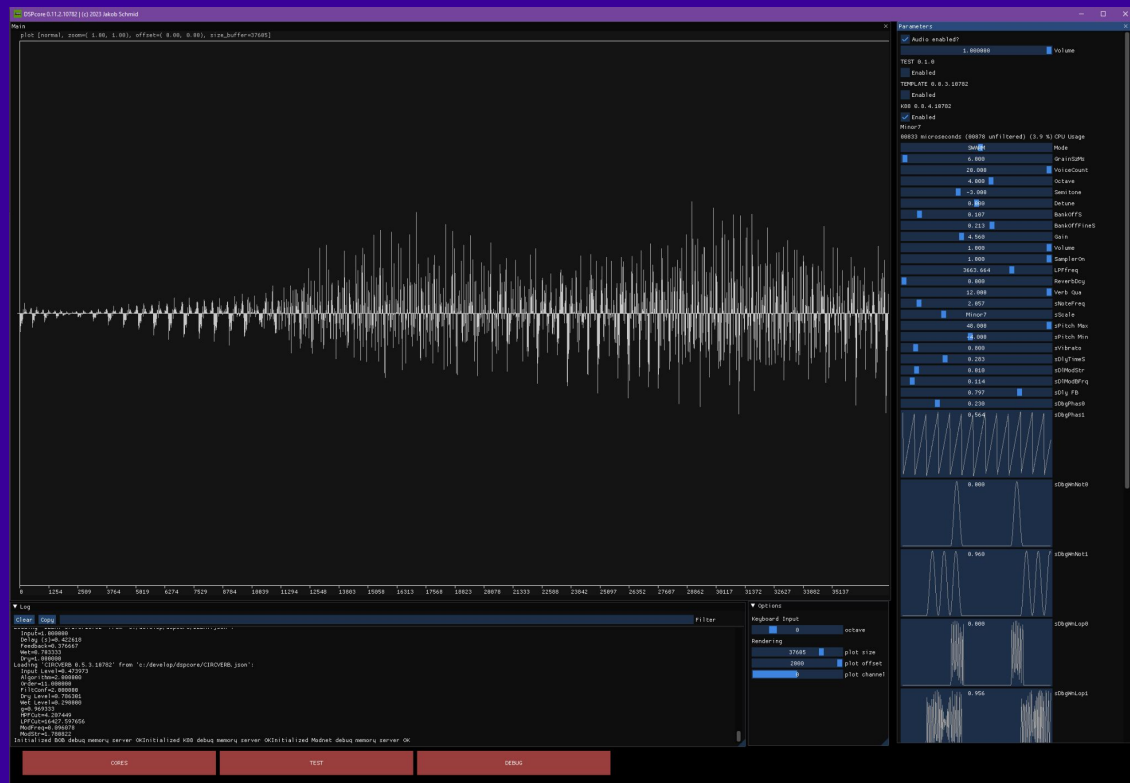


# DSPcore.exe: Test Interface

Visualizes output waveform

Easy to step debug

Can show debug info for plugin instances



# Wrap as FMOD Plug-in Instrument

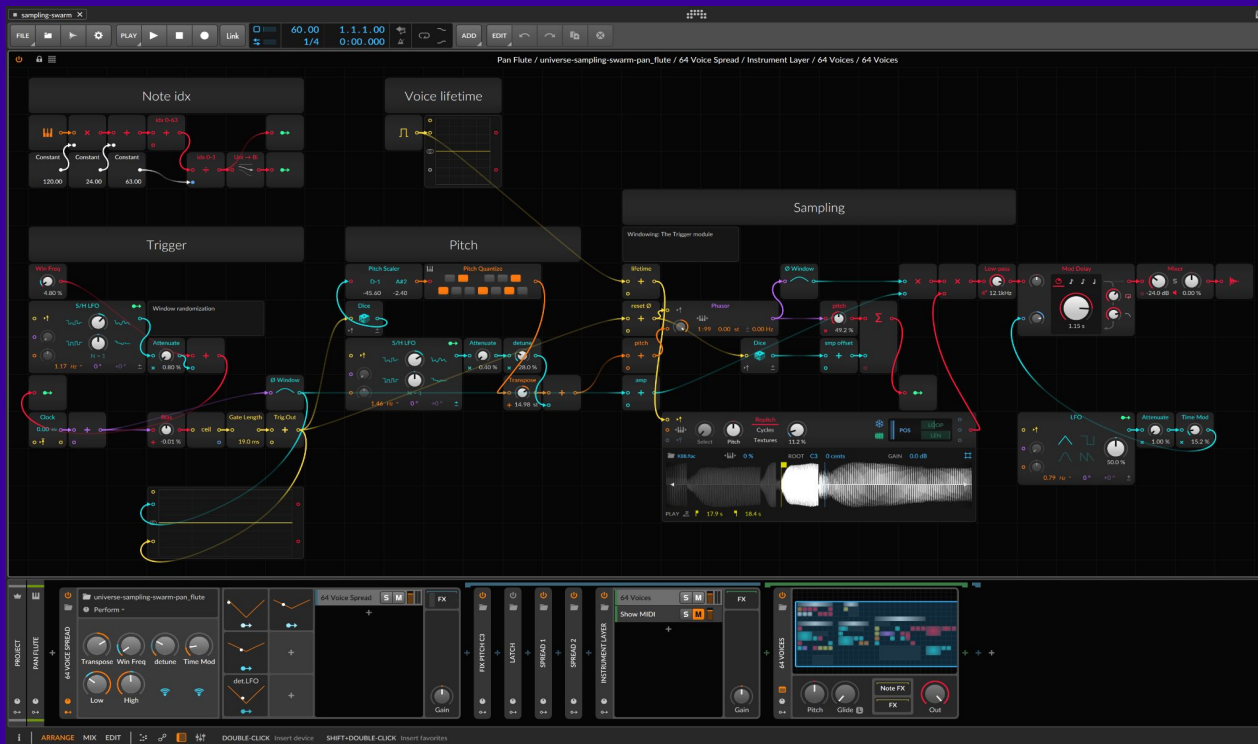
The screenshot displays the FMOD Studio interface. At the top, the Timeline shows a track for 'mus-paramA' with a 'K88' instrument track highlighted in pink. Below the timeline, the Logic Tracks section shows 'Strings', 'Null', 'dry', and 'Master' tracks. The right side of the interface shows the 'Local' and 'Global' parameter settings for 'mus-paramA', with values of 0.15 and 0.87 respectively.

The main parameter control panel for the 'K88' instrument is shown below. It includes a 'Schmid | K88' logo and a 'Debug Dump' section with 'Orchestra' and 'Swarm' modes. The 'Swarm' mode is selected. The panel is divided into several sections:

- BANK OFFSET:** Grain Size (ms) at 5.00, Offset (s) at 45%, Fine Offset (s) at 0%.
- ORCHESTRA:** Voice Spread at 14.5k, Random offset at 665.
- SWARM:** Note Freq at 4.30, Note Chance at 0%, Pitch min at -8.00, Pitch max at 28.0, Scale at 5.80, Vibrato at 5.90.
- Automatable:** LPF Freq at 11.1k, Volume at 100%.
- Preset Config:** Gain at 2.00, Reverb Decay at 61%.
- Other parameters:** Sampler On (ON), Voice Count at 3, Octave at 4.40, Semitone at 0.00, Detune at -26%, Frequency at 0.16, Amount at 39%, Smoothness at 87%, Delay at 45%, Feedback at 36%, Base Freq. at 0.01, Strength at 0.00.



# K88: Implementing Swarm Mode



Bitwig Grid patch

# K88 Components

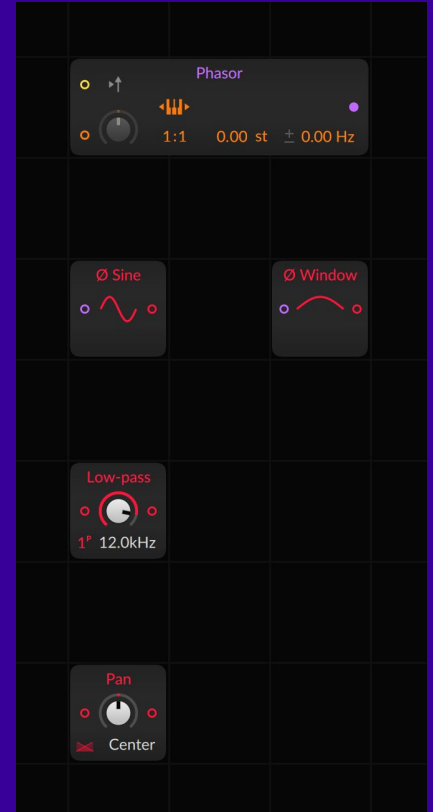
**Phase generator** clock component, generates a control signal 0..1

**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: Phase Generator

**Clock** component that generates a control signal 0..1

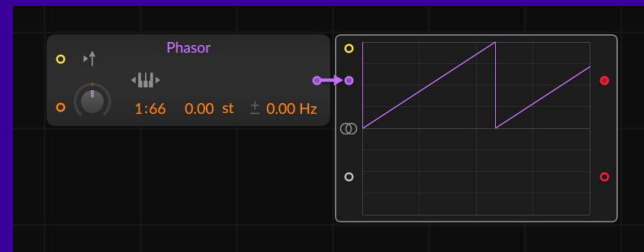
**Oscillator** use as input for a function or table to generate any periodic signal

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*

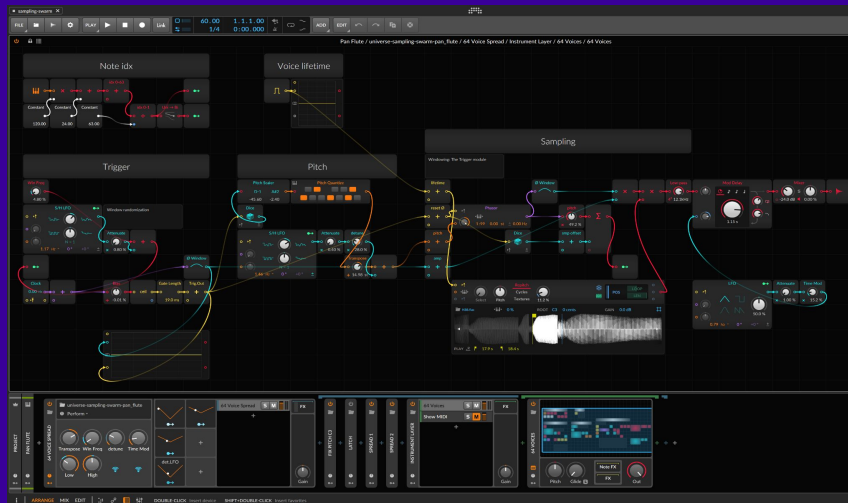


*Generating sine wave using phase generator*

# K88: Translate Signal Graph to C++

Signal graph is a tree structure

Render 'leaves' first, use as input for branches



```
void Stereo::render(float32_stereo_interleaved(float* buffer, int32_t sample_frames, unsigned_t clock) {
    using namespace random_spr_shuff;

    // Process reasonable default values
    window_size_n = clamp(window_size_n, min_window_size, max_window_size);
    bank_offset_n = clamp(bank_offset_n, min_bank_offset_n, max_bank_offset_n);

    float start_time_n = bank_offset_n - window_size_n * 0.5f + 0.5f;
    float end_time_n = bank_offset_n + window_size_n * 0.5f + 0.5f;
    float start_sep = start_time_n * buffer;
    float end_sep = end_time_n * buffer;
    int max_offset = min(window_size, static_cast<int>(end_sep - start_sep));

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

    float amp = sqrtf(1.f / voice_count);

    for (int voice = 0; voice = voice_count; ++voice) {
        float wbt = idx % 4 == voice * 2;
        float pan_factor_1 = pan_factor_factor * wbt;
        float pan_factor_2 = pan_factor_factor * (1 - wbt);
        voice_channel_state = voice_channel_state;

        float window_big = 0.5f;
        float window_loop = 0.5f;

        int idx = 0;
        for (int i = 0; count = sample_frames; i < count; ++i) {
            bool retrigger = state.note_phaser_is_pulse_on();
            if (retrigger) {
                // Trigger
                if (random_float() < note_chance) {
                    int pitch = lerp<int>(pitch_min, pitch_max, random_float());
                    int pitch_scale = quantize_pitch_informally(pitch, scale_bitfield);
                    int current_offset = pan_pos * max_offset;
                    state.start_sep = current_offset + start_sep;
                    state.end_sep = current_offset + end_sep;

                    // FROM: Compute relative pitch, assuming waveform is c3
                    float temp = octave * 12 + state.note = detune;
                    state.current_freq = floatid2freq(pitch_scale + temp);

                    state.sample_phaser_restart();
                }
            } else {
                state.current_freq = -1.f; // voice off
            }

            float wsb = 0.5f;
            float wml = 0.5f;

            bool is_voice_on = (state.current_freq != -1.f);
            if (!is_voice_on) {
                float freq = state.current_freq;
                state.pend_get_val(id) += freq;
                state.sample_phaser_set_freq(freq * window_size_n * sample_rate);
                window_big = handling_window_loop(id) * (state.note_phaser_phase);
                window_loop = handling_window_loop(id) * (state.sample_phaser_phase);
                float phase0 = state.sample_phaser_phase;
                float sample_id = lerp<int>(state.start_sep, state.end_sep, phase0);

                // Interpolated sample loop
                float amp_n = window_size * window_loop;
                float out = get_interpolated_sample_decrypt(waveform, max_offset_length, sample_id + amp_n * wsb);
                wml = amp * pan_factor_1;
                wml = wml * amp * pan_factor_2;

                mult += state.delay_render_single_annotated();
                mult = state.delay_render_single_annotated();

                buffer[idx++] += mult;
                buffer[idx++] += mult;
            }

            state.delay_idt_delay_idt(mod_delay_time + mod);
            state.delay_idt_delay_idt(mod_delay_time + mod);
            state.delay_and_phase_update();
            state.sample_phaser_update();
            state.note_phaser_update();
            state.pend_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

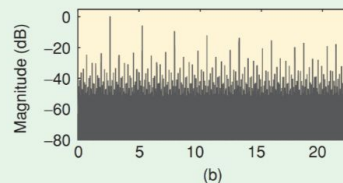
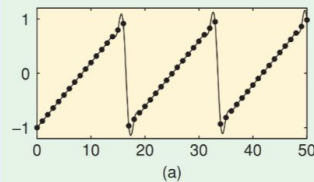




# BOB: Band-limited Oscillators

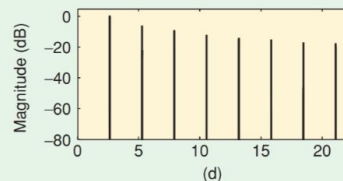
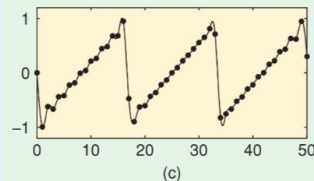
Saw

Trivial sawtooth

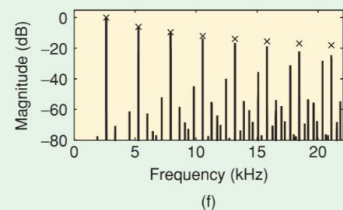
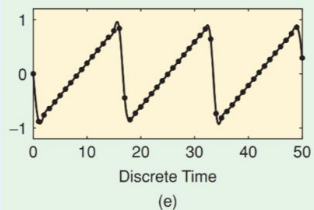


Sqr

Ideal band-limited  
sawtooth: sum of  
sines



PolyBLEP  
approximation



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

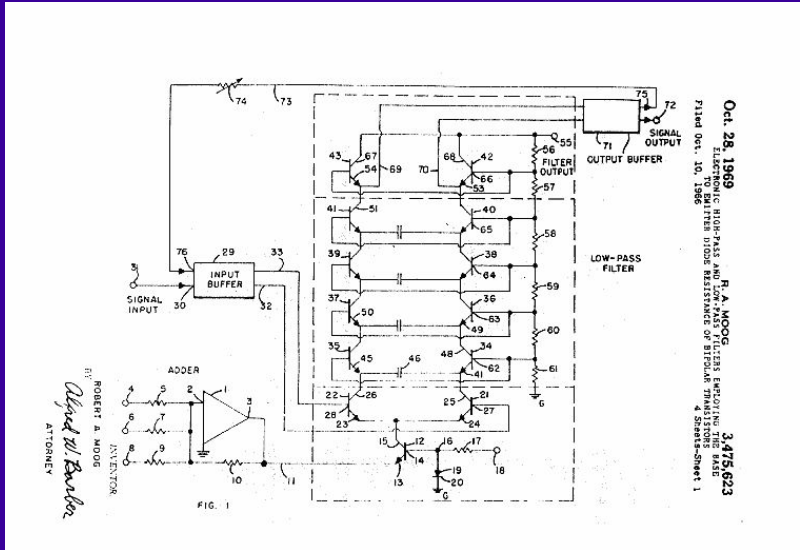
Antialiasing Oscillators in Subtractive Synthesis

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 7<sup>th</sup> 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.

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

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

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

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

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

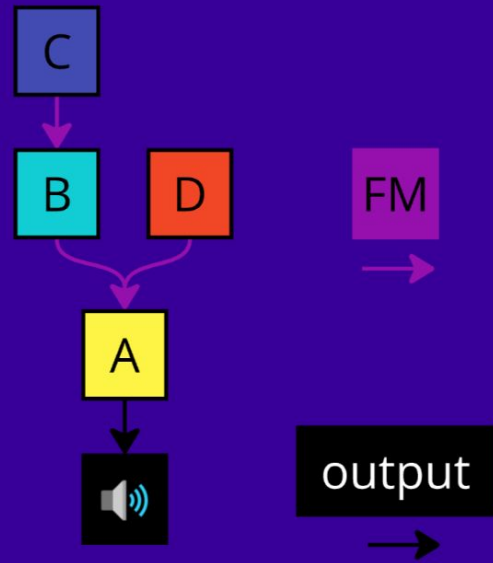
Frequency Modulation Matrix

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

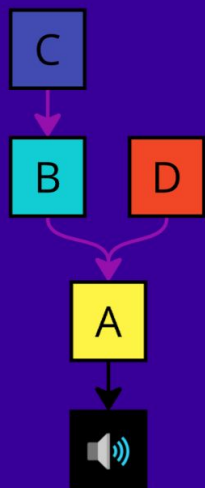
AM	A	B	C	D	
A	0	0	0	0	1
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
	0	0	0	0	0

# Modnet: Ableton Operator Algorithm 3



# Modnet Normalized Form

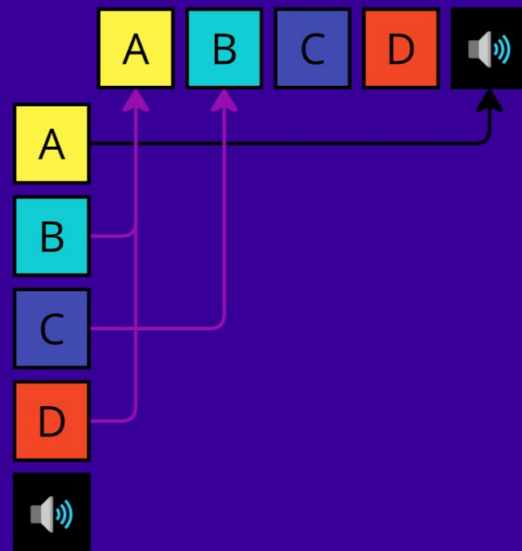
Operator Algorithm



FM

output

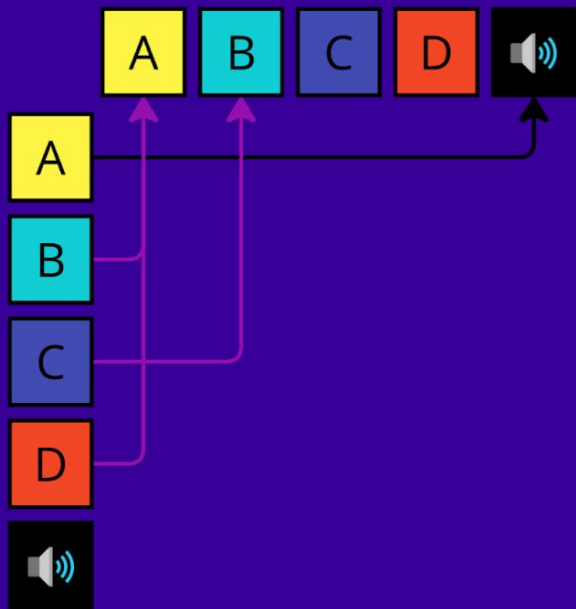
Equivalent  
vertical  $\rightarrow$  horizontal  
routing representation



# Modnet: FM and AM/output Matrices

Frequency Modulation Matrix

Amplitude Modulation / Output Matrix



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

AM	A	B	C	D	Speaker
A	0	0	0	0	1
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
Speaker	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 essings: 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: 0.00 ARP:multiply: 5.00
ARP:rest.delc: 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
PLFO str: 0.00 Tr. Sine: -12.00 PWM str: 0.55 PWM freq: 0.19
PLFO str: 0.29 PLFO freq: 6.00 AENV:attack: 0.82 AENV:decay: 0.81
AENV:sustain: 0.77 AENV:release: 7.60 FENV:attack: 0.55 FENV:decay: 4.68
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: 2.000
Semi tone: 0.000 Detune: -0.250 BankOffFS: 0.533 BankOffFineS: 0.000
Gain: 2.000 Volume: 1.000 SamplerOn: 1.000 LPFFreq: 11100.000
ReverbDcy: 0.610 oVocofFSmp: 14500.000 ooffModAmt: 0.385 ooffModFrq: 0.161
ooffModSmo: 0.865 oVibraStr: 665.000 oGrStFrst: 00000000 oGrStLast: 00000000
oGrPsFrst: 00000000 oGrPsLast: 00000000 sNoteFreq: 4.300 sNoteChnc: 4.000
sScale: N/A sPitch Max: -14.000 sPitch Min: 5.900 sPitch Atn: 0.451
sVibrato: 0.005 sDlTimeS: 0.011 sDlModStr: 0.360 sDlModFrq: 0.000
sDly FB: 0.988 sDbgPhas1: 0.000 sDbgWnNot0: 0.000 sDbgWnNot1: 0.000
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: 200.000 VoiceCount: 3.000 Octave: 3.000
Semi tone: 6.000 Detune: 0.020 BankOffFS: 0.500 BankOffFineS: -0.221
Gain: 4.000 Volume: 0.515 SamplerOn: 0.000 LPFFreq: 10100.000
ReverbDcy: 0.625 oVocofFSmp: 0.000 ooffModAmt: 0.000 ooffModFrq: 0.000
ooffModSmo: 0.000 oVibraStr: 0.000 oGrStFrst: 00000000 oGrStLast: 00000000
oGrPsFrst: 00000000 oGrPsLast: 00000000 sNoteFreq: 1.460 sNoteChnc: 6.000
sScale: N/A sPitch Max: -17.000 sPitch Min: 5.300 sPitch Atn: 0.671
sVibrato: 0.003 sDlTimeS: 0.060 sDlModStr: 0.740 sDlModFrq: 1.000
sDly FB: 0.294 sDbgPhas1: 0.058 sDbgWnNot0: 0.637 sDbgWnNot1: 0.441
sDbgWnLop0: 0.200 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:

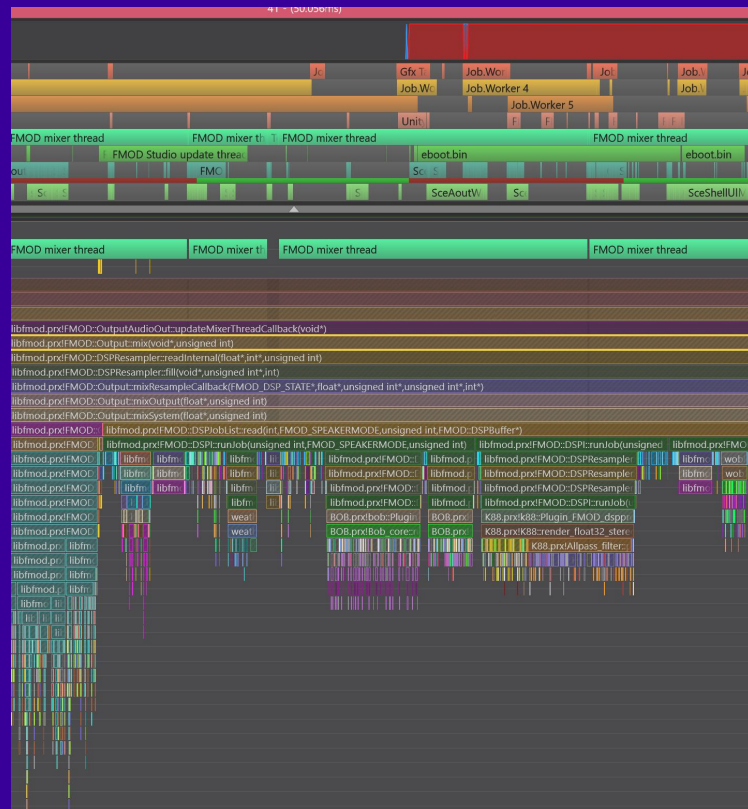
- Steinberg VST for music software
- Unity Native Audio Plugin for the built-in Unity audio system

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



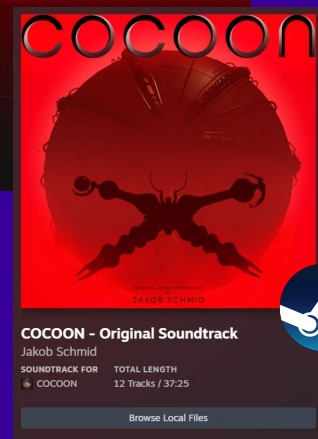
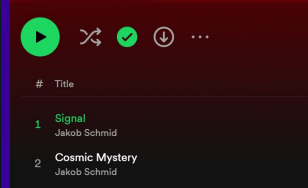


# Questions?

Contact me on

E-mail: [jakob@schmid.dk](mailto:jakob@schmid.dk)

Twitter: [@jakobschmid](https://twitter.com/jakobschmid)



AVAILABLE NOW ON















# Bonus Slides



# Core Component: DC Filter

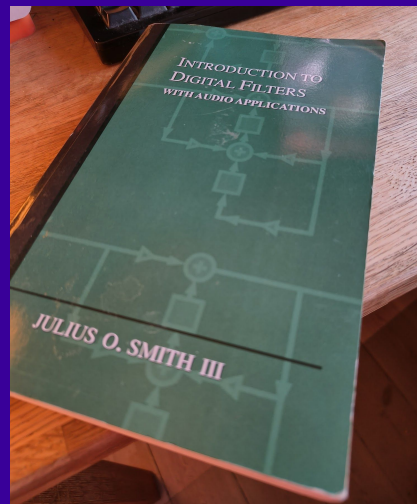
DCF

$$y[n] = x[n] - x[n-1] + R * y[n-1]$$

$$R = 1 - (2 * \text{PI} * \text{cutoff\_freq} / \text{sample\_rate})$$

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

- *R calculation by [hc.niweulb@lossor.ydna](mailto:hc.niweulb@lossor.ydna) at [musicdsp.org](http://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

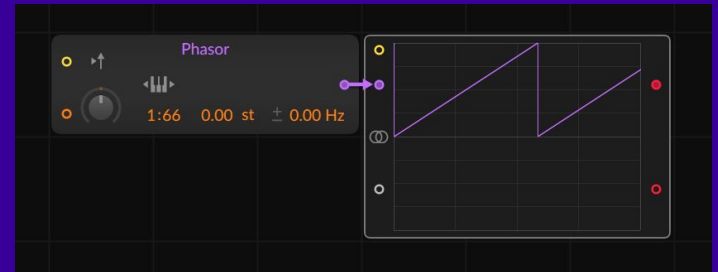


# Phase Generator Implementation

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

```
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

```
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, // set position callback
    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;
}
```

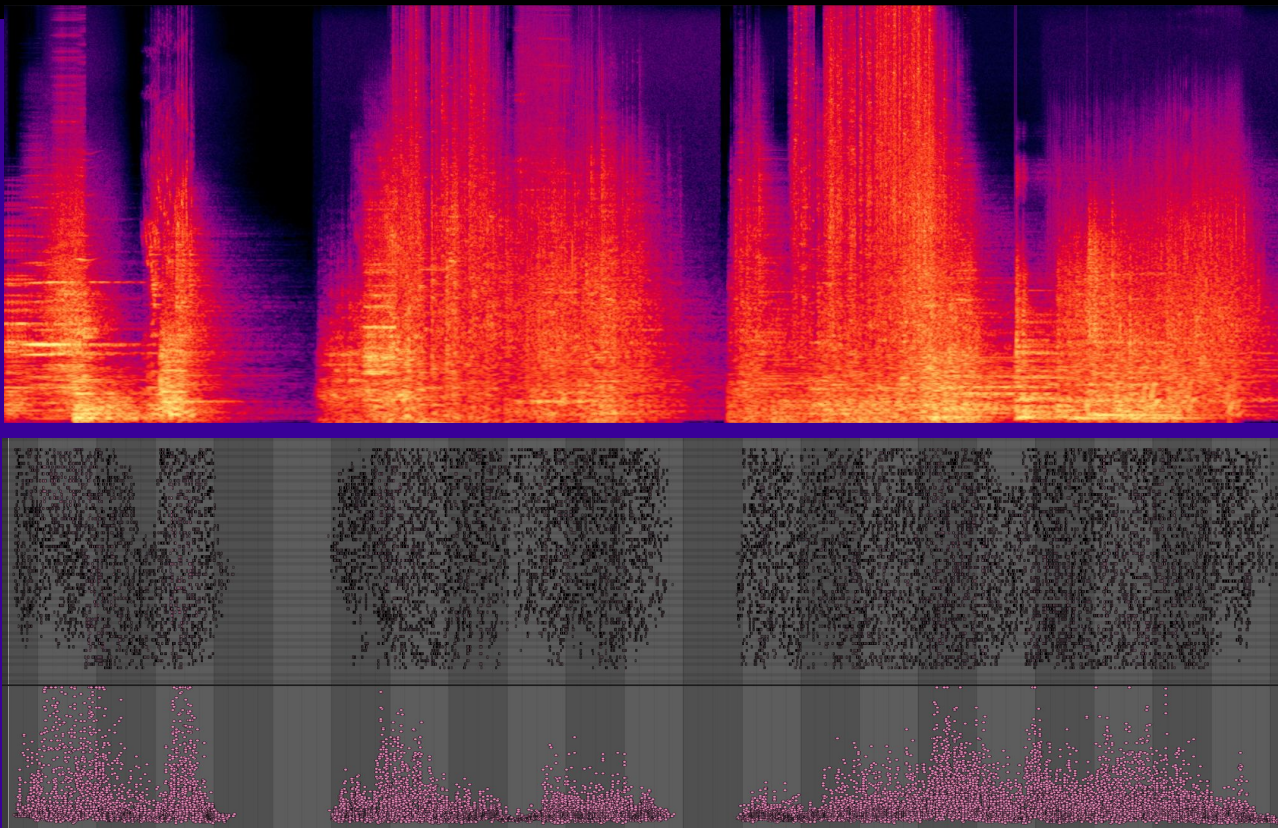
# Translate More Components to C++



```
class Mod_delay
{
private:
    Cincbuf 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

This patch does a vocoder-like spectral analysis of any audio using 64 Sallen-Key BP 8-pole filters, and sends the result as 64 MIDI notes with velocity, corresponding to frequency and amplitude.

The temporal resolution is controlled using note frequency and chance. Chance values lower than 100% reduces bandwidth and often leads to a more pleasing end result when resynthesizing. Values around 50% are recommended.

The smoothness parameter determines the window size used for amplitude detection. Larger values 'blur' the sound.

Note Generator

Detect Amplitude

To generate 64 notes, we fix pitch to C3 and use 2 MULTI-NOTES to generate 8\*8=64 NOTE GRID voices.

Pitch	Velo	Spread	Chance
C3	0	0.00%	100%
C3	12	0.00%	100%
C3	24	0.00%	100%
C3	36	0.00%	100%
C3	48	0.00%	100%
C3	60	0.00%	100%

In this example, we use Ableton Live for resynthesis of the spectral MIDI data.

However, any MIDI device should work, even hardware devices.

loopMIDI is used for sending MIDI notes from Bitwig to Ableton Live.

On current hardware, Live can't handle more than around 45 MB/s of MIDI data.

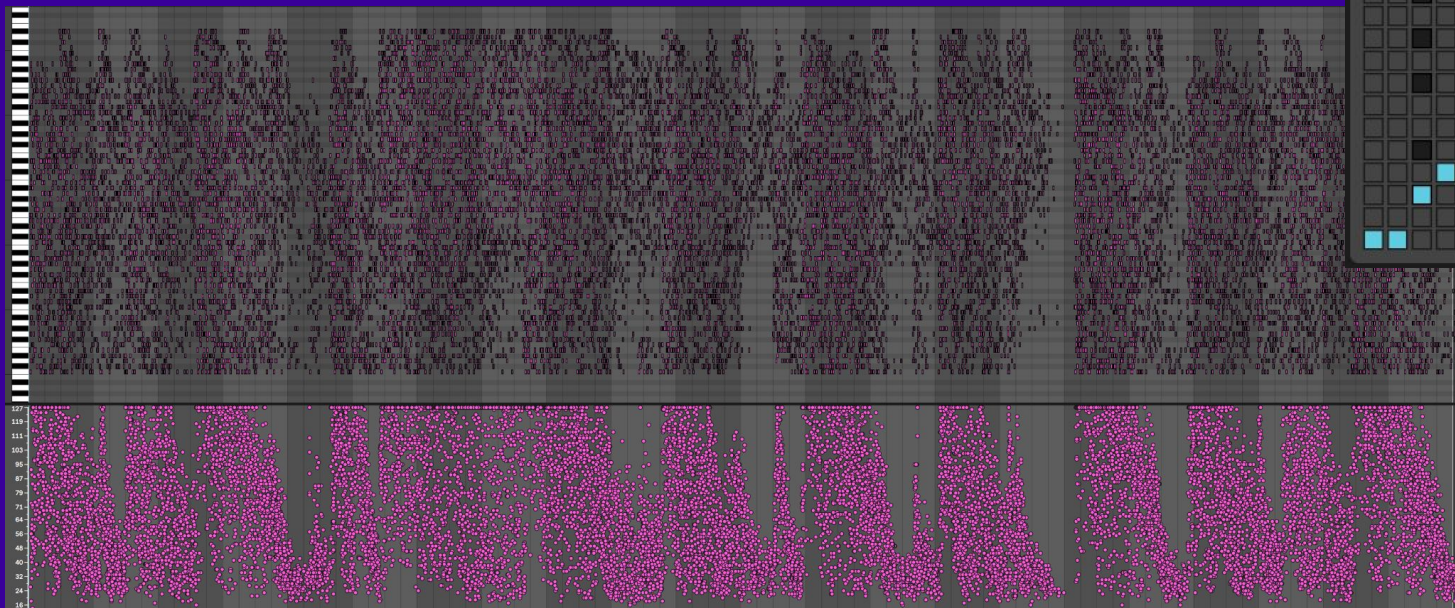
Disable Feedback-Detection in loopMIDI to avoid auto-muting.

To extend polyphony of any Ableton Instrument, use an Instrument Rack with Key splits.

# Puzzle Feedback Music



# MIDI Vocoder: Puzzle Feedback



**Ambigorian**

Base  
B

Transpose  
0 st

Fold

Range  
+128 st  
Lowest  
C-2





# Inspiration

Quite & Orange: CDAK (2010)

Music by Lassi Nikko

4K demo

