COCOON

Game Audio Playthrough Nov. 2023

What is COCOON?

A puzzle adventure game

Geometric Interactive

Director:

Jeppe Carlsen

Art director:

Erwin Kho

Production time: 6.5 years

▸ cocoon-intro

COCOON Audio Team

Audio direction / music:

Jakob Schmid

Sound design:

Julian Lentz
Mikkel Anttila

Music Concept

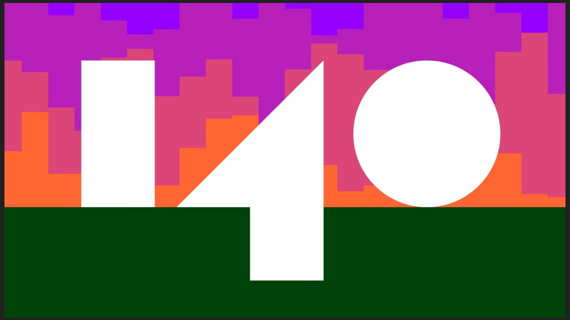
Generative music using real-time synthesis

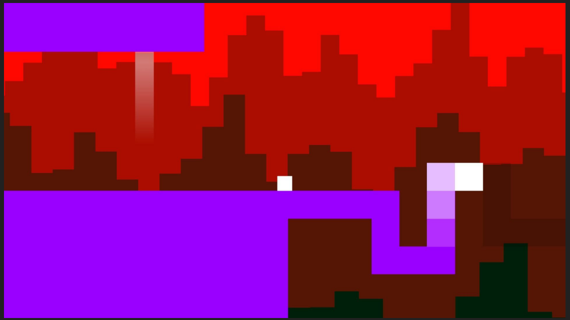
- Loop free during ‘thinking breaks’
- Unique soundtrack for each player

Sound Design Concept

Synthetic sound design - no recorded sound!

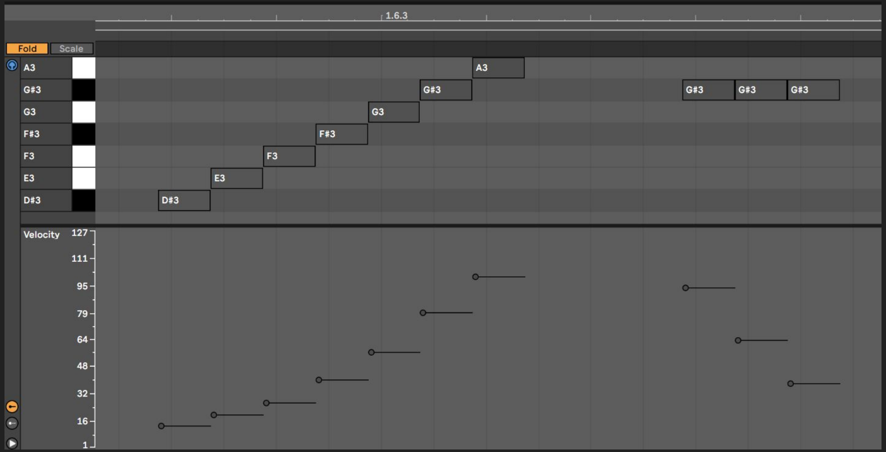
- Fits aesthetics of generative music
- Fits Erwin’s art style: artificial but alive
- Familiar process from ‘140’

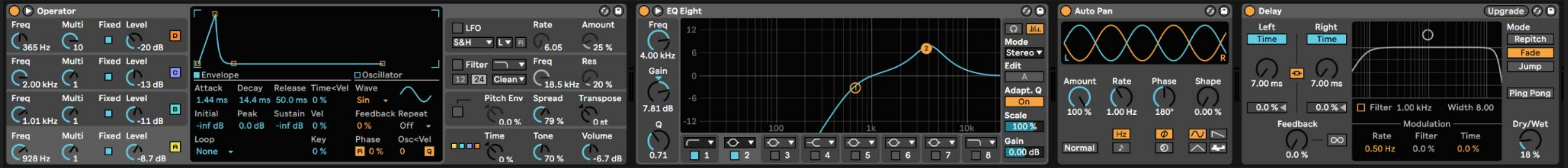




Synthetic Sound Design Experiments

Frogs, footsteps, portals





▸ frog

Sound Designers Needed!

Synthetic sound design

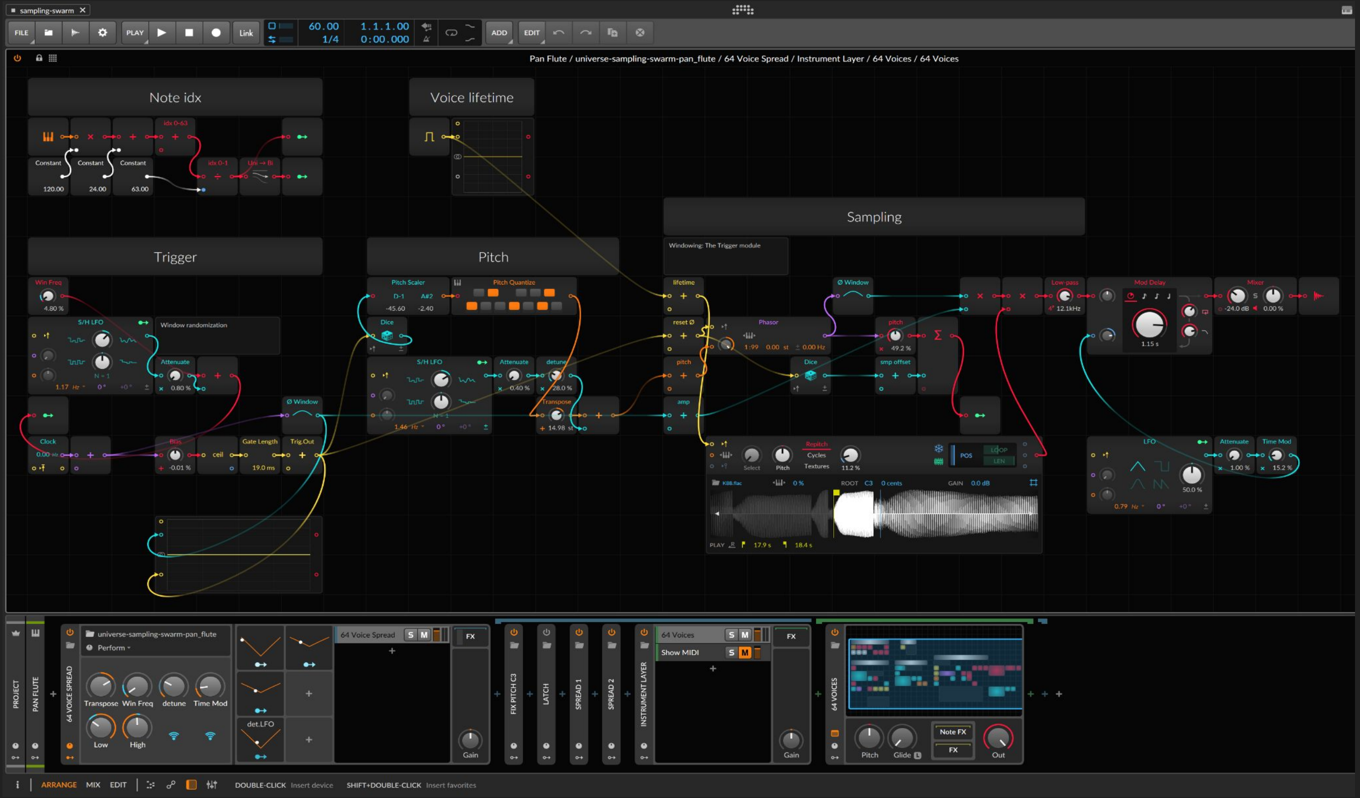
- Challenging and fun
- Slow process
- I needed help!

And now … Julian

Real-time Synthesized Music

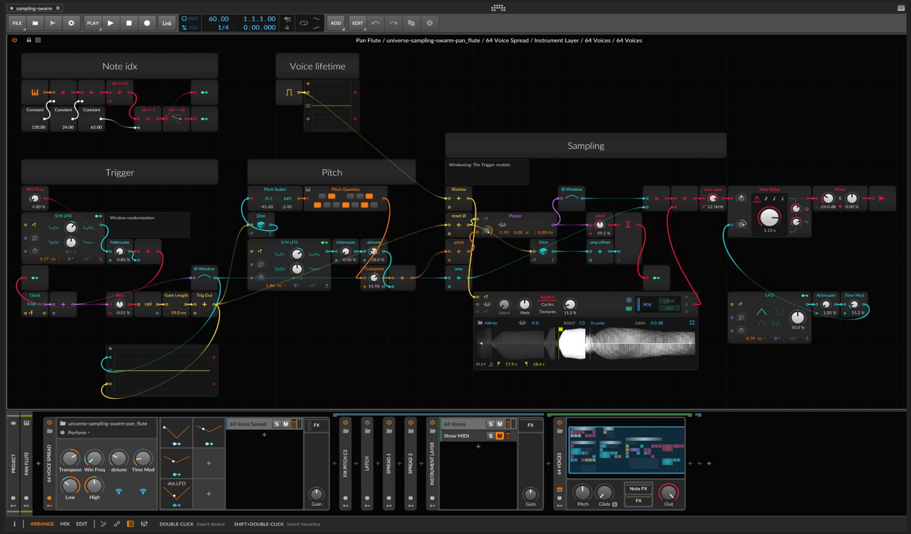
▸ cocoon-gig

Bitwig Granular Swarm Experiment

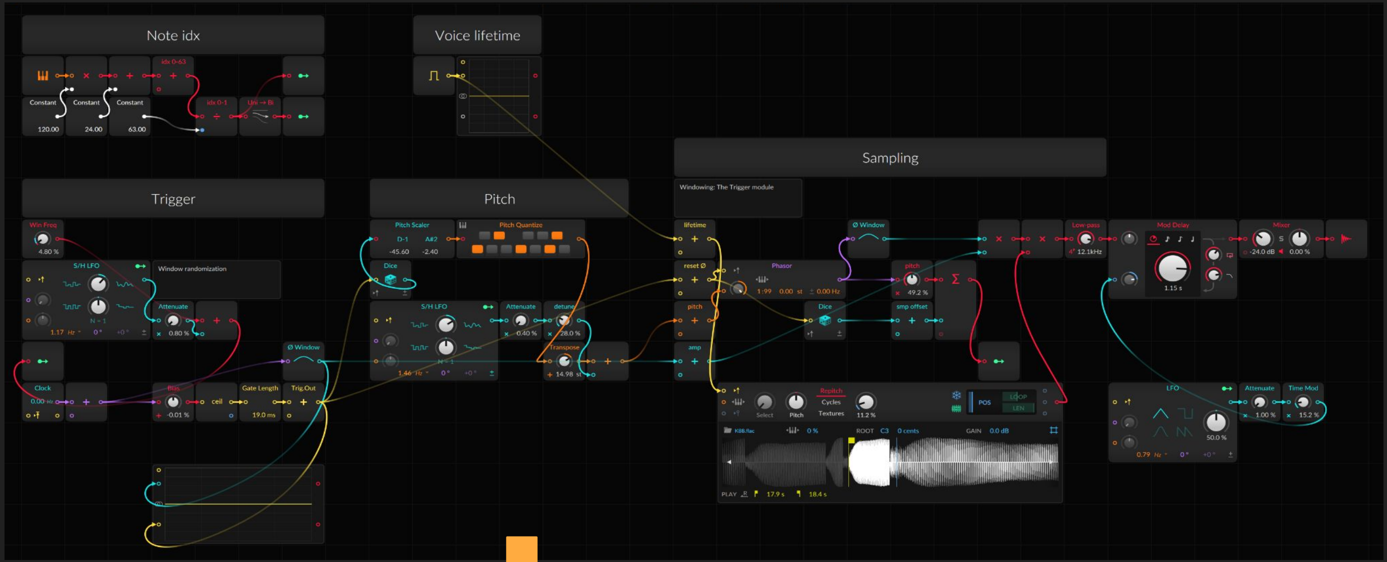


▸ bitwig-swarm

What if this was in the Game?



From Bitwig Prototype to FMOD Plugin



Translate Components to C++

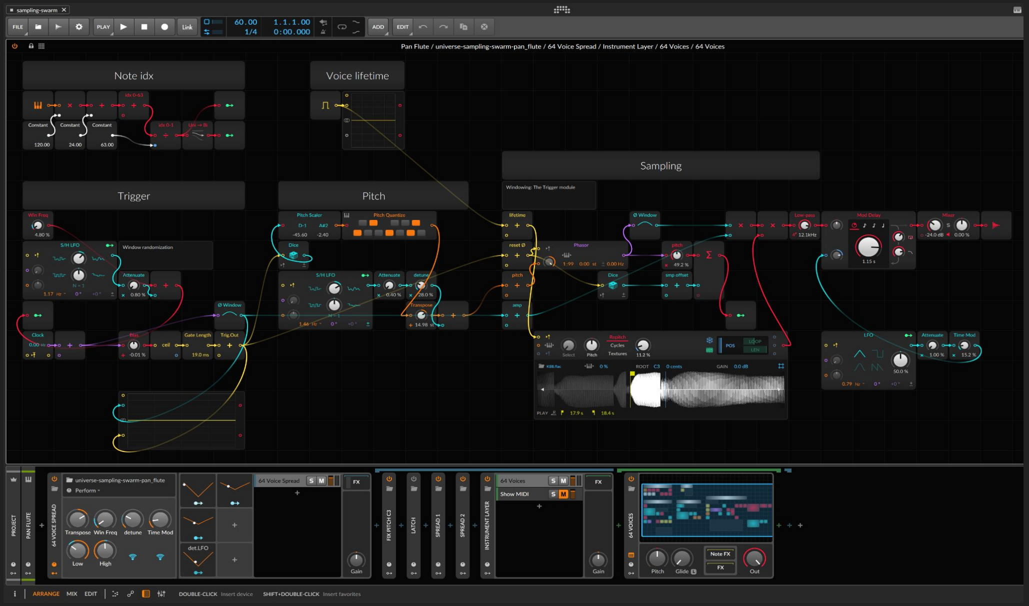

```
// Uniformly quantized pitch
// int note              = (pitch % 12)
// octave01              = note / 12.0
// pitch_quantized_idx = octave01 * selected_pitches
// pitch_quantized     = selected_pitches[ pitch_quantized_idx ]
inline int quantize_pitch_uniformly(int pitch, int *selected_pitches, int selected_pitches_count)
{
    int octave = pitch / 12;
    int note = mod_wrap_i(pitch, 0, 12);
    float octave01 = note / 12.0;
    int pitch_quantized_idx = octave01 * selected_pitches_count;
    assert(pitch_quantized_idx >= 0 && pitch_quantized_idx < selected_pitches_count);
    int pitch_quantized = selected_pitches[pitch_quantized_idx];
    return pitch_quantized + octave * 12;
}
```


```
class Phaser
{
private:
    const float PHASE_MAX = 4294967296;

public:

    struct State
    {
        uint32_t phase, freq__ph_p_smp;
        bool is_active;
    };

    /// State
    uint32_t phase = 0u; // using an integer type automatically ensures limits
                         // phase is in [0 ; 2^(32-1)]
    //const float PHASE_MAX = (1 << 32); // this yields 0.0f for some reason ??!?
    uint32_t freq__ph_p_smp = 0u;
    bool is_active = true;

    Phaser() {}

    void save_state(State &state)
    {
        state.phase = phase;
        state.freq__ph_p_smp = freq__ph_p_smp;
        state.is_active = is_active;
    }

    void load_state(const State &state)
    {
        phase = state.phase;
        freq__ph_p_smp = state.freq__ph_p_smp;
        is_active = state.is_active;
    }

    inline void restart()
    {
        phase = 0u;
        is_active = true;
    }
    inline void update()
    {
        phase += freq__ph_p_smp;
    }
```


```
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);
};
```

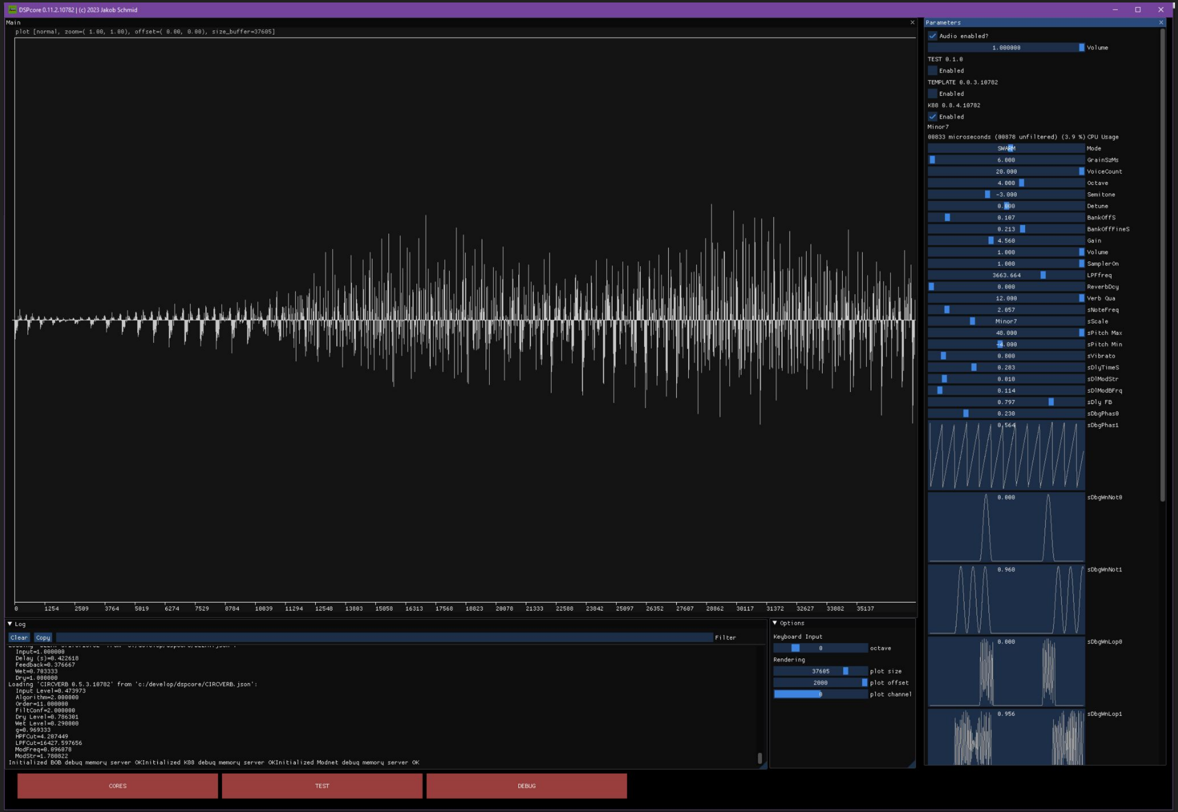

```
class Sample_and_hold
{
    Phaser phaser;

    float target_value;
    float current_value;
    float slew_rate;

    float sample_period;

public:
    Sample_and_hold()
    {
        target_value = random_xor_shift::random_float01();
        current_value = target_value;
        set_smoothness(0);
        sample_period = 1 / 48000.0f;
    }
    void set_freq(float freq, int sample_rate)
    {
        phaser.set_freq(freq, sample_rate);
        sample_period = 1.0f / sample_rate;
    }
    // smoothness01  rate/s
    // 0             100 (change instantly: full change in a 100th of a second)
    // 1             0.1 (full change in 10 seconds)
    void set_smoothness(float smoothness01)
    {
        float smoothness01_exp = ease_out(smoothness01, 4.0f);

        float slew_rate_per_second = lerp_inline(100.0f, 0.1f, smoothness01_exp);
        slew_rate = slew_rate_per_second * sample_period;
    }
    void update()
    {
        if (phaser.is_pulse_now())
        {
            target_value = random_xor_shift::random_float01();
        }
        phaser.update();

        current_value = slew(current_value, target_value, slew_rate);
    }
    float get_value01() // Call update first
    {
        return current_value;
    }
};
```

Translate Patch to C++




```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();
        }
    }
}
```

Test in custom GUI

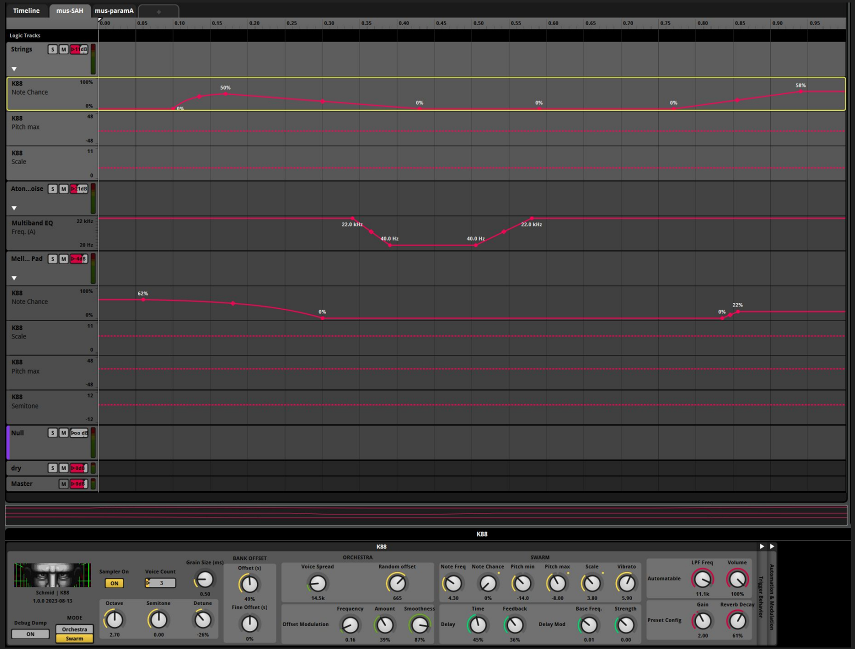


Wrap as FMOD Plug-in Instrument

▶ fmod-k88

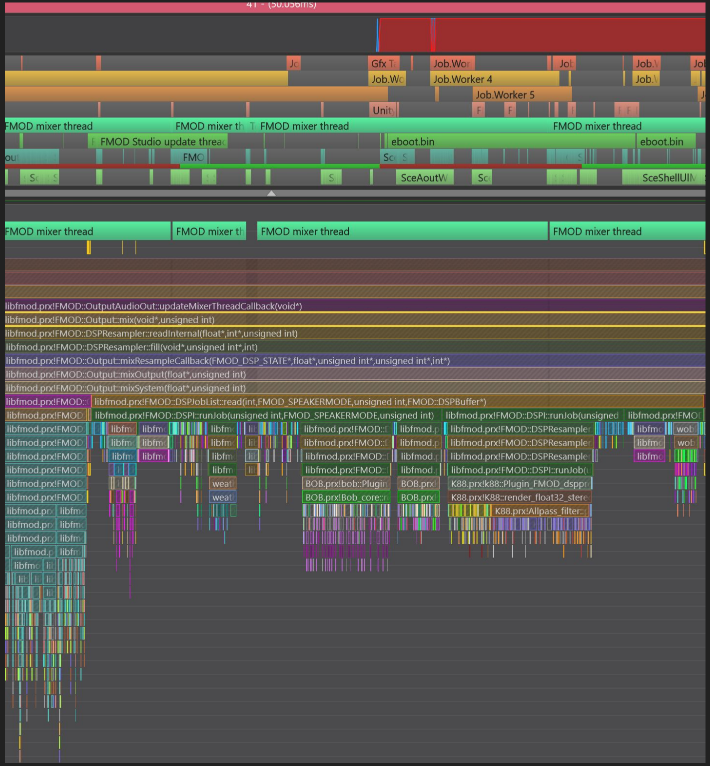
COCOON Plugin Instruments

Real-time Synthesized Music in FMOD

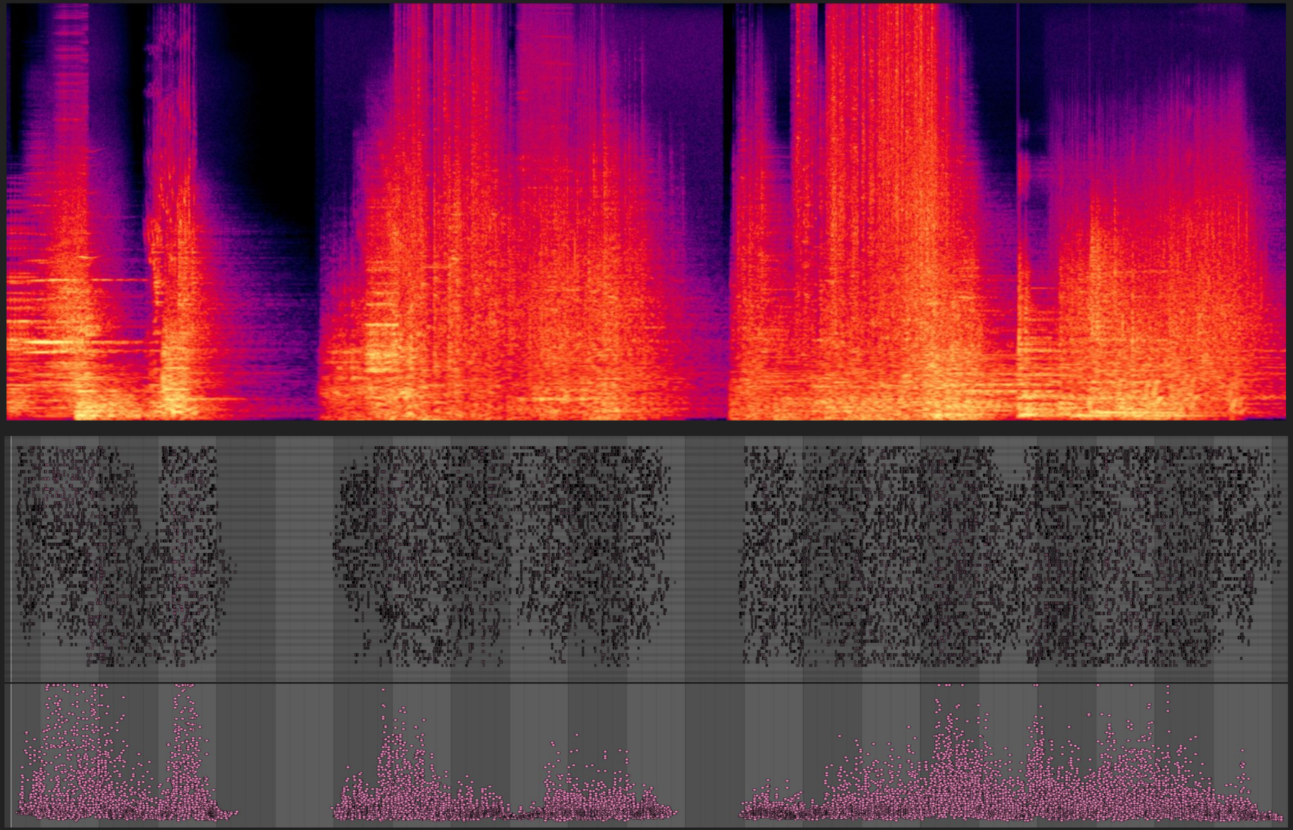


Real-time Synthesized Music on All Platforms

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



MIDI Vocoder: Dyson Gate

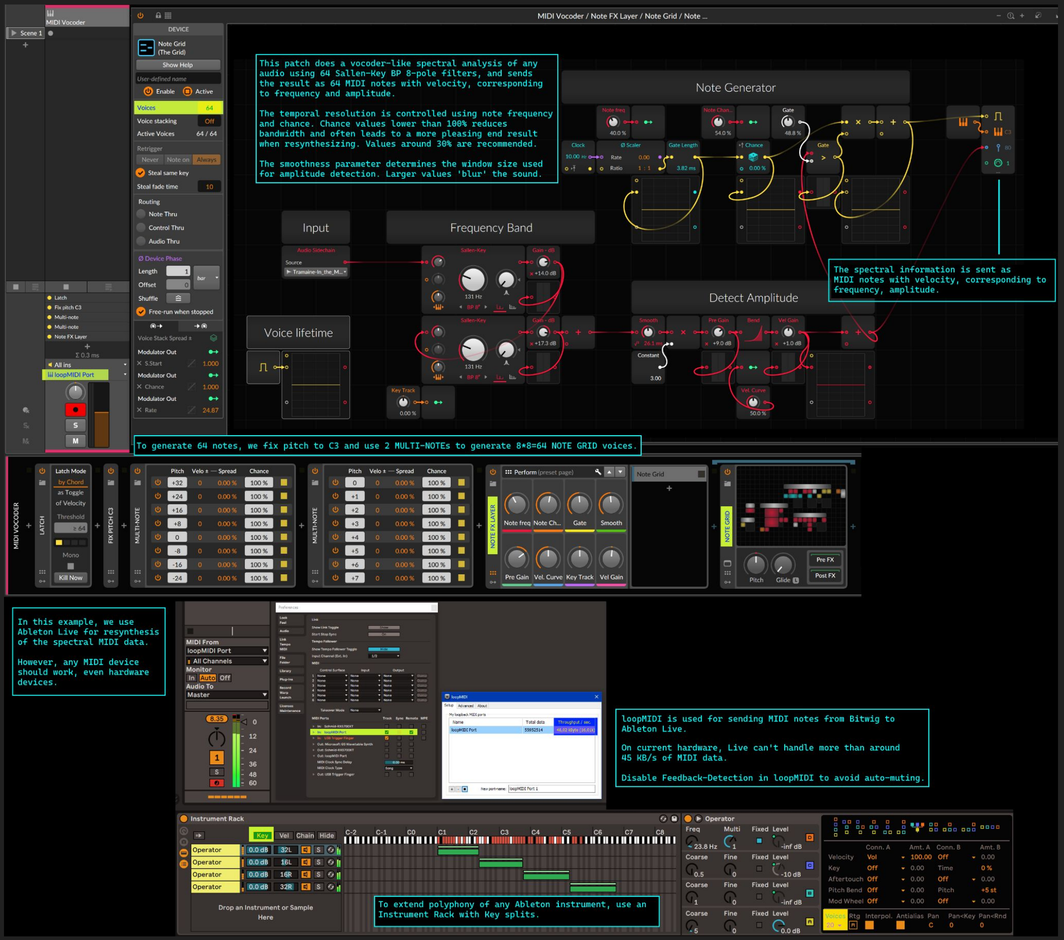


▸ midi_vocoder-bitwig, midi_vocoder-ableton, cocoon-gate

MIDI Vocoder

Home-made vocoder

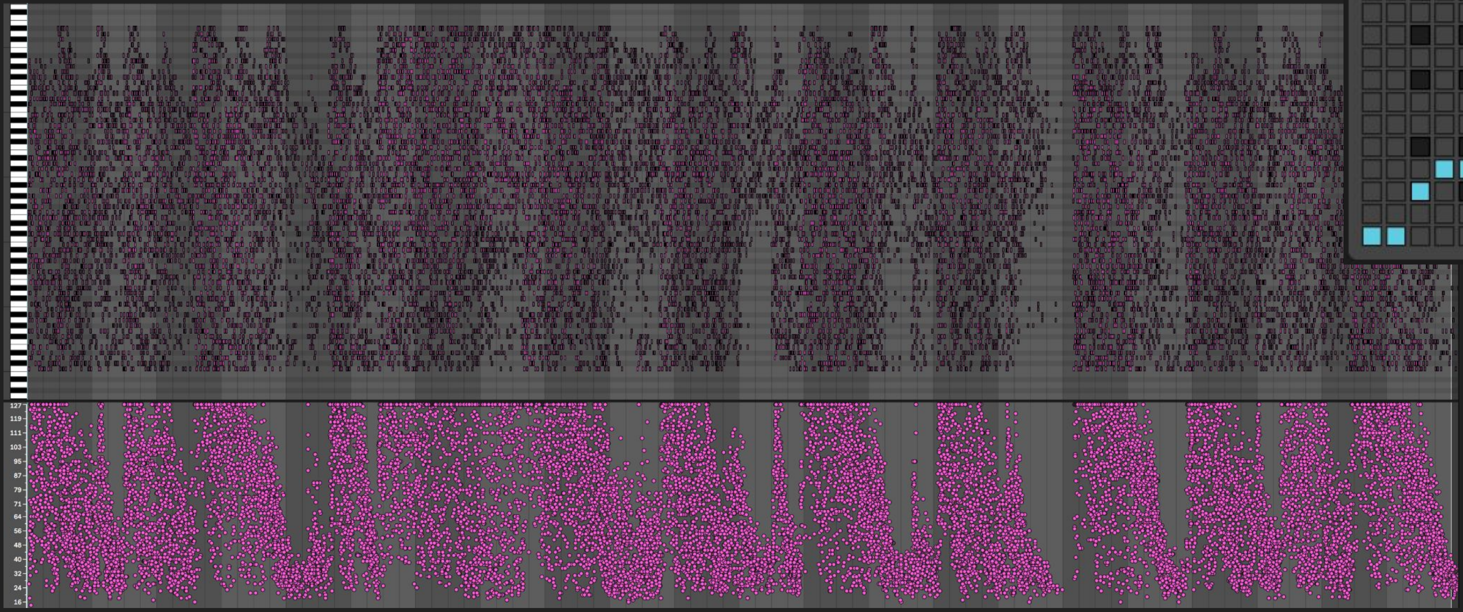
- Bitwig audio analysis
- MIDI sent via loopMIDI
- Record MIDI in Ableton Live

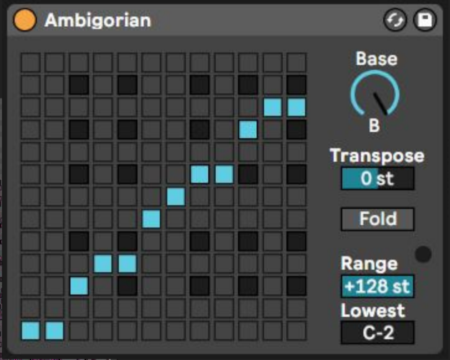


Puzzle Feedback Music

▸ cocoon-puzzfeed

MIDI Vocoder: Puzzle Feedback





▸ midi_vocoder-puzzfeed

Questions?

Contact Jakob on

- twitter.com/jakobschmid
- jakob@schmid.dk

AVAILABLE NOW ON

STEAM XBOX NINTENDO SWITCH PS5 | PS4