Audio Engineering Society

AES Game Audio Webinar 2024

Jakob Schmid
Geometric Interactive

What is COCOON?

A puzzle adventure game by

Geometric Interactive

Director:

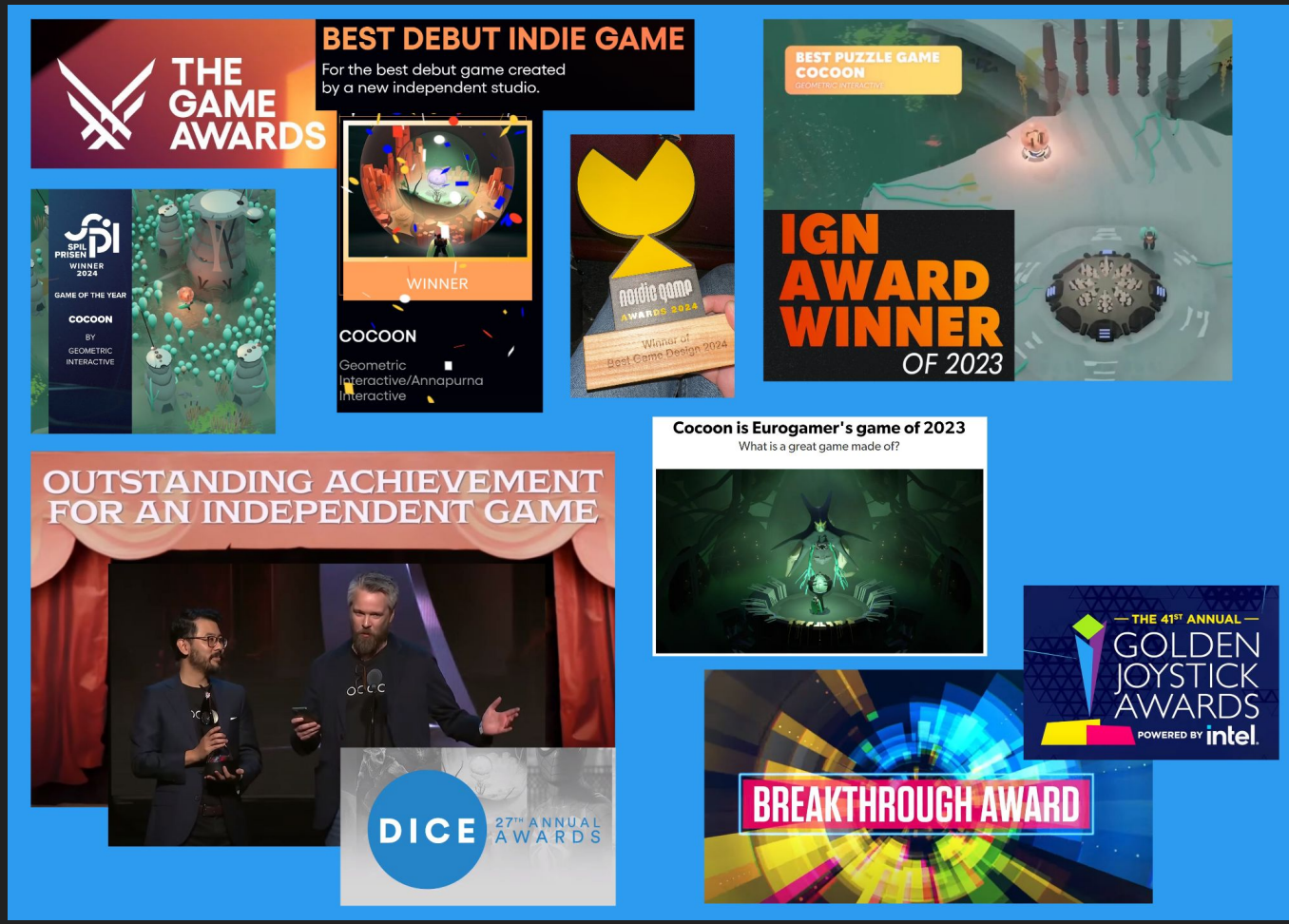
Jeppe Carlsen

Art director:

Erwin Kho

Production time: 6.5 years

Accolades



▸ cocoon-intro

Topics

COCOON audio concepts and production

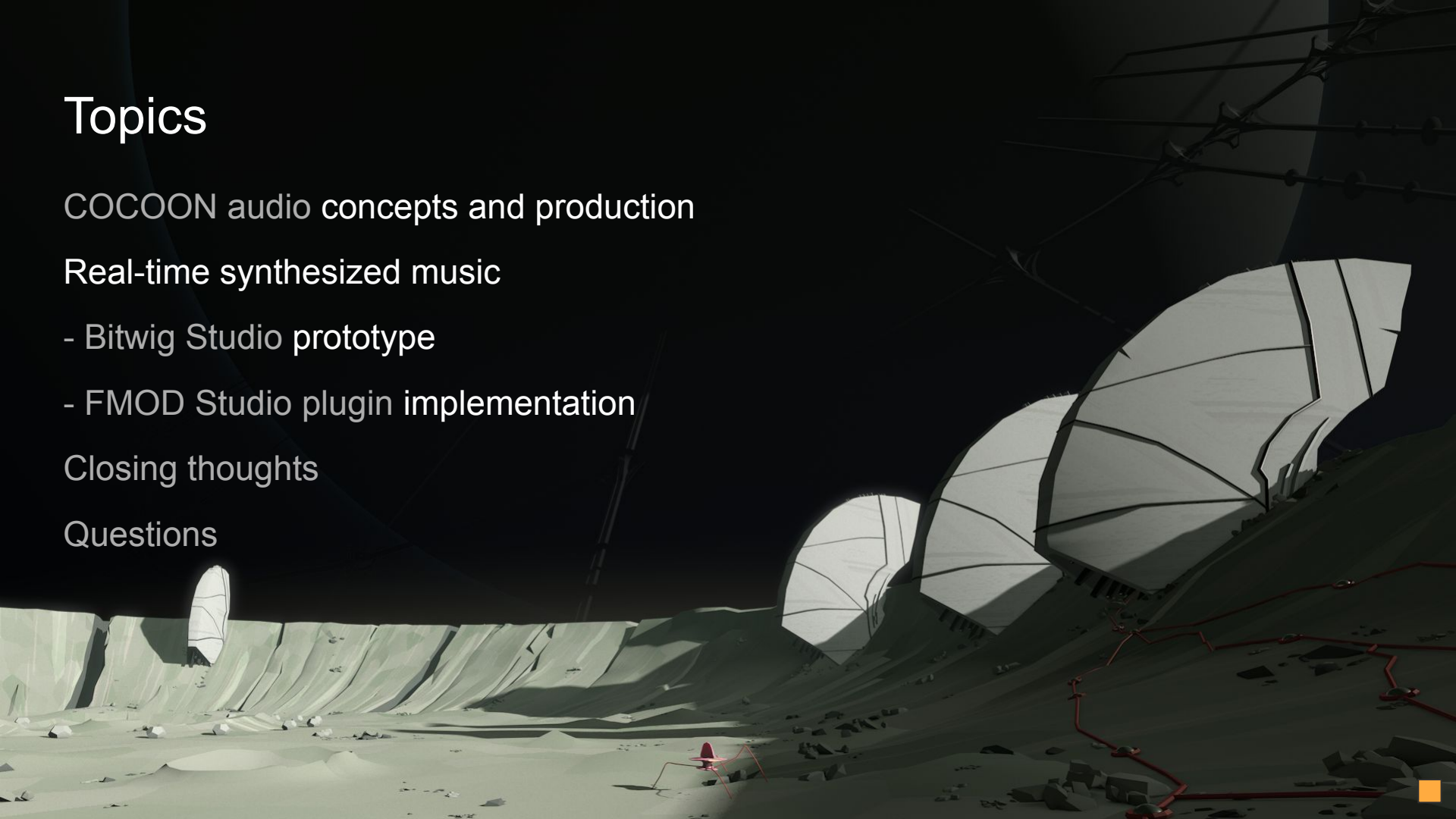
Real-time synthesized music

- Bitwig Studio prototype

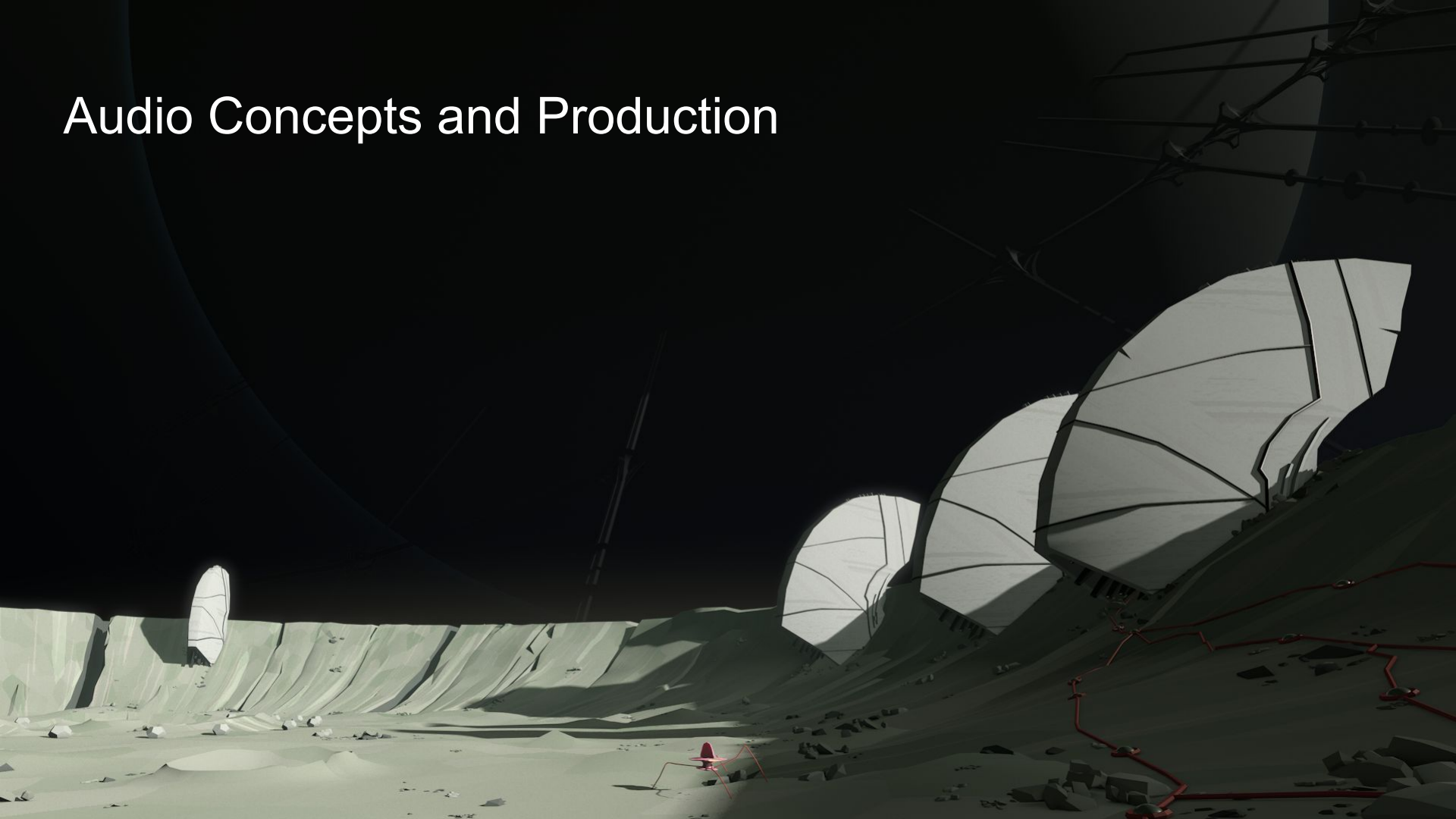
- FMOD Studio plugin implementation

Closing thoughts

Questions



Audio Concepts and Production



COCOON Audio Team

Audio direction / music:

Jakob Schmid

Sound design:

Julian Lentz
Mikkel Anttila

Music Concept

Pre-composed vignettes for the big moments

Synthesized ambient music for puzzle gameplay

Big moment: Vignette

Puzzle gameplay: synthesized ambient music

Synthesized Ambient Music

COCOON's ambient music is done with real-time synthesis

Benefits:

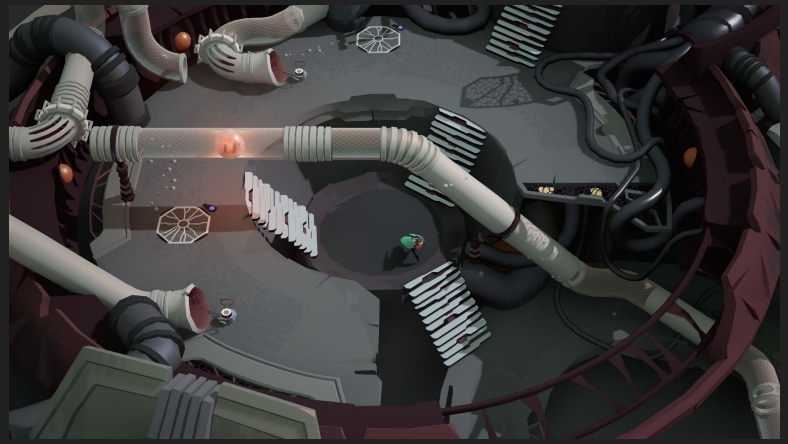
- Loop free during ‘thinking breaks’
- Unique soundtrack for each player
- Music can react to player position and game events
- Ambient music take up 5 MB on disk in total - (the game is around 5 hours long)

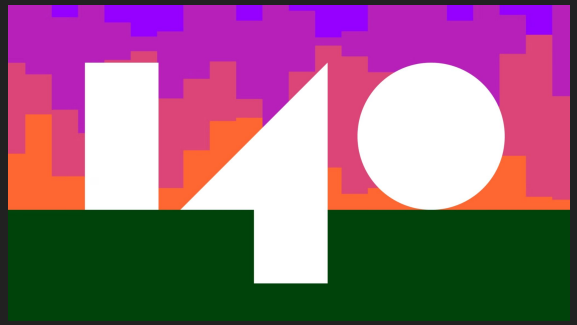
COCOON synthesizer plugin in FMOD Studio

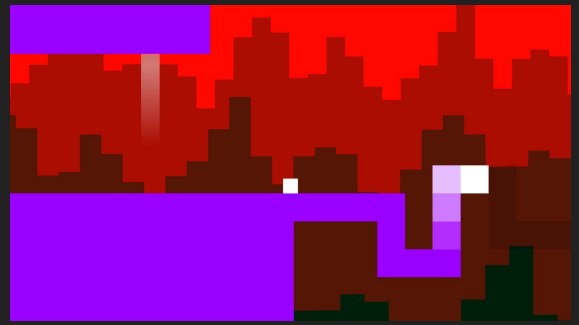
Sound Design Concept

Synthetic sound design - no recorded sound!

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

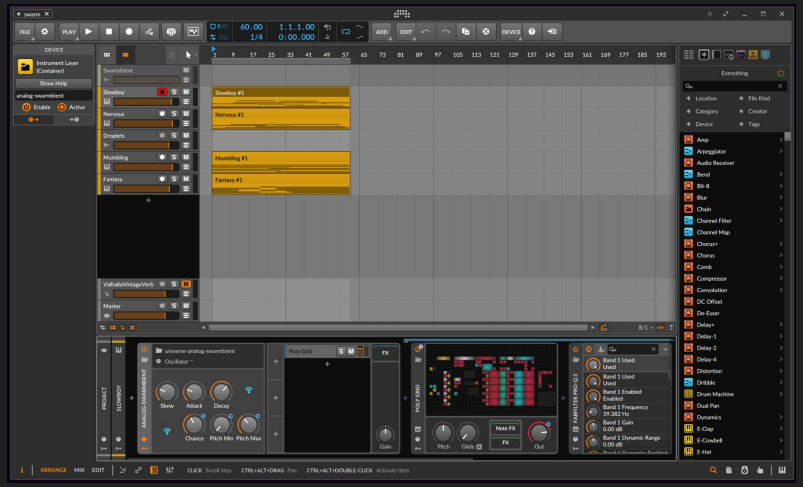




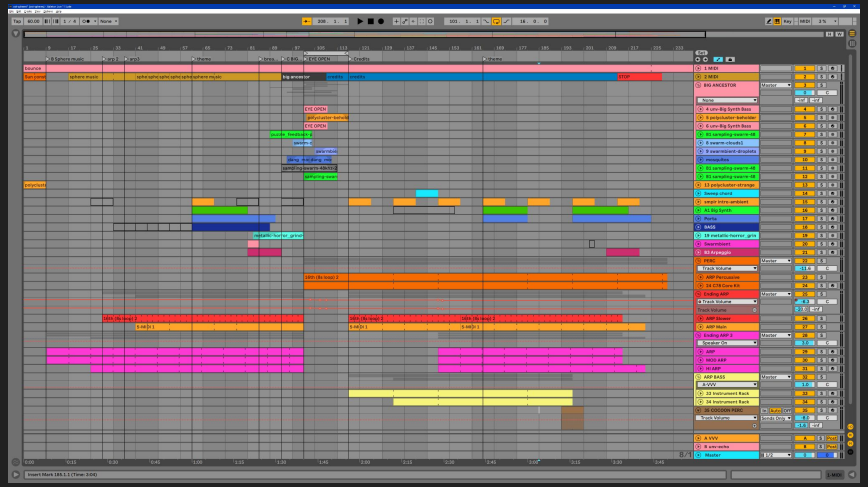


Music software

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



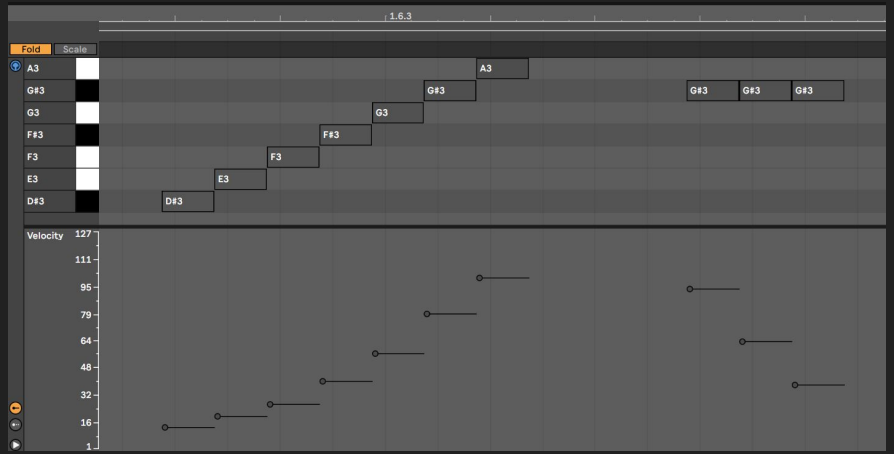
Bitwig Studio 5

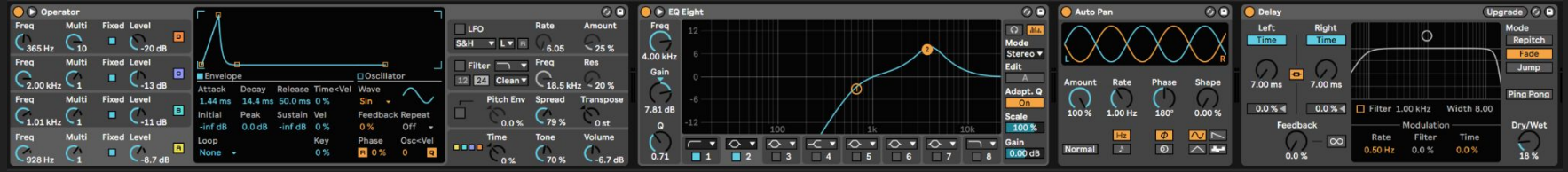


Ableton Live 11

Synthetic Sound Design Experiments

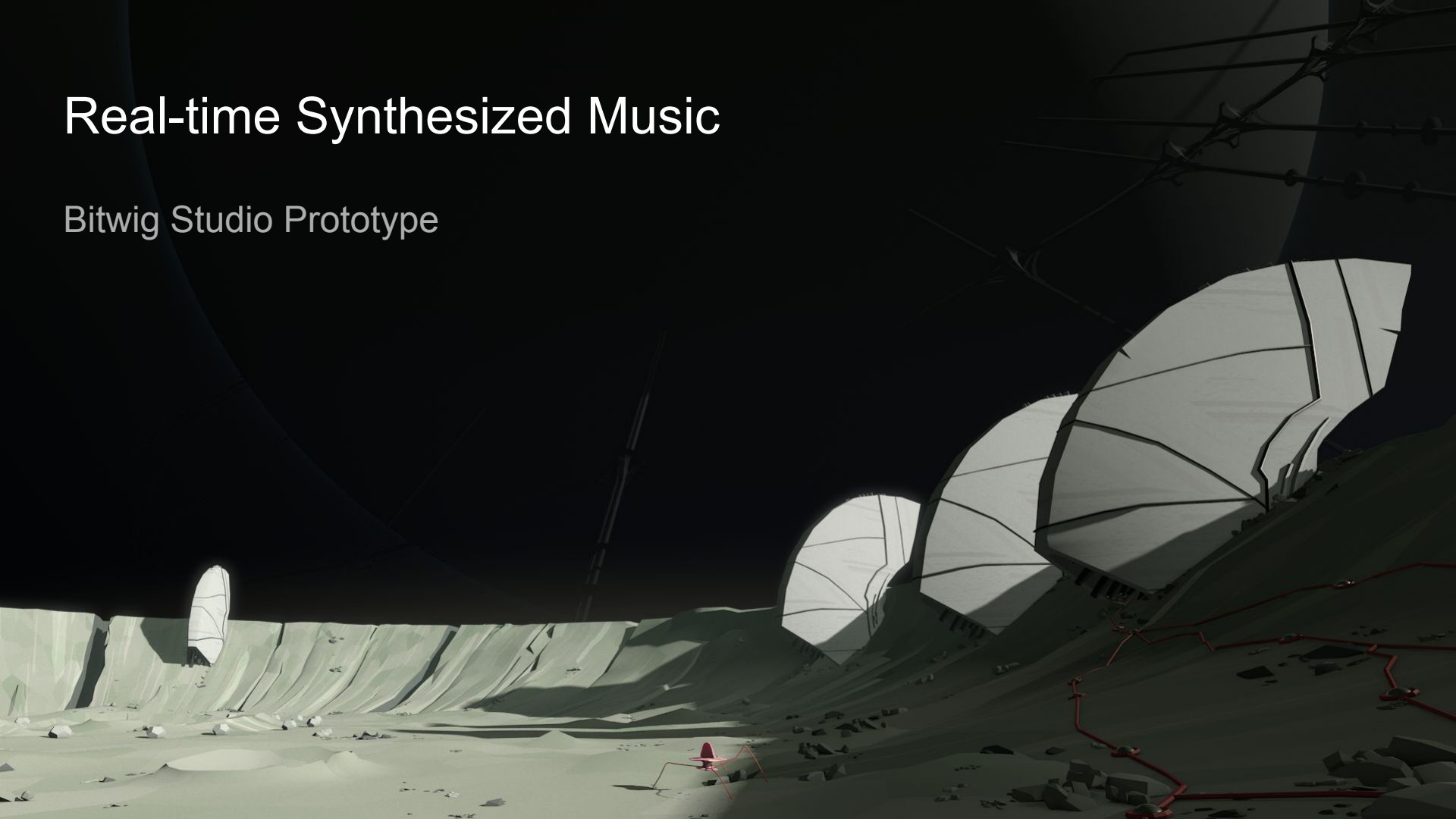
Frogs, footsteps, portals



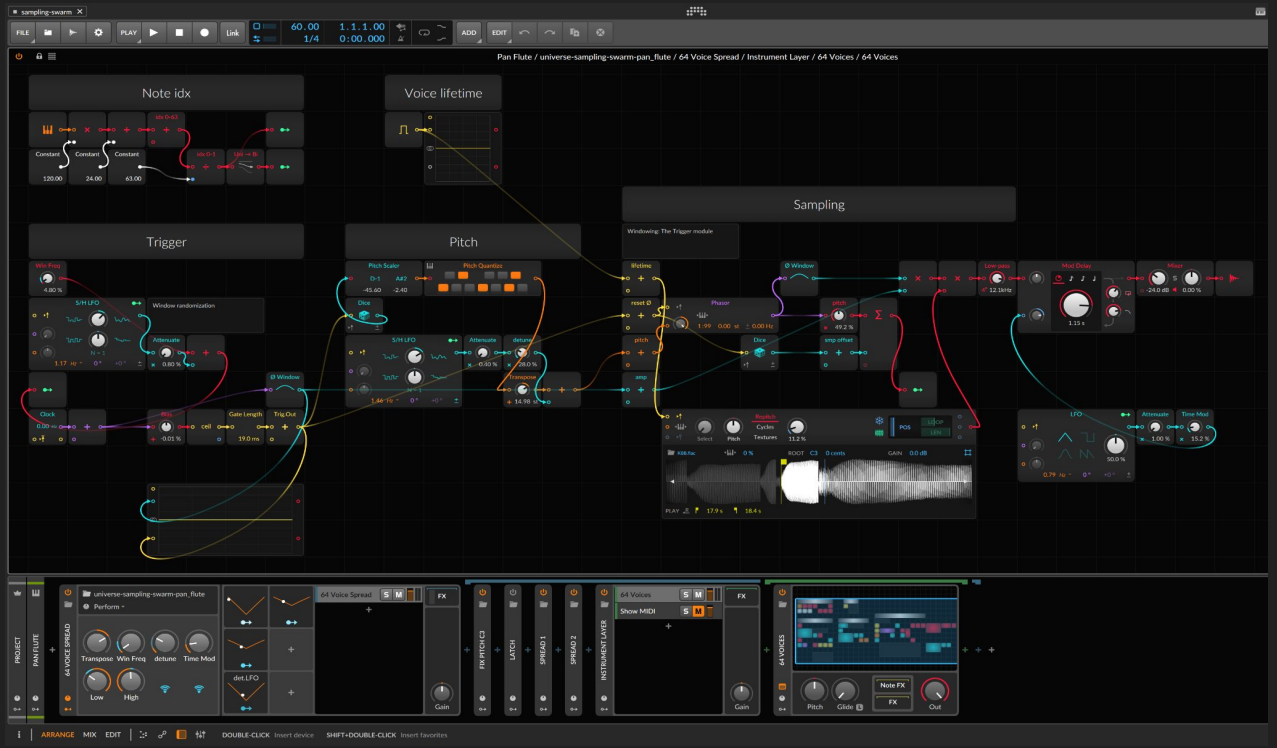


Real-time Synthesized Music

Bitwig Studio Prototype



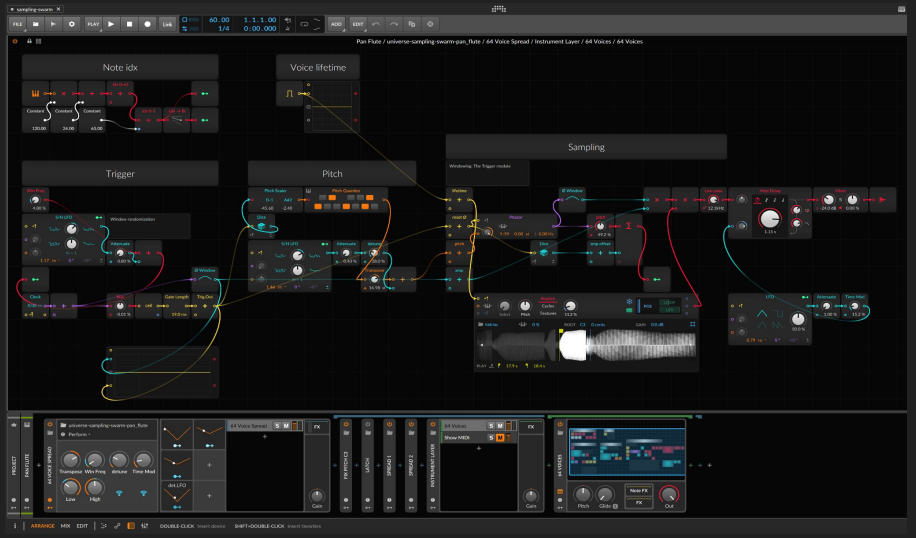
Bitwig Granular Swarm Experiment



Bitwig Grid patch

▸ bitwig-swarm

What if this was in the Game?



Real-time Synthesized Music

FMOD Studio plugin implementation

How to write an FMOD Studio Plugin

FMOD Studio plugin API is open

Plugins are normally written in C++

FMOD Studio plugin K88 used in COCOON

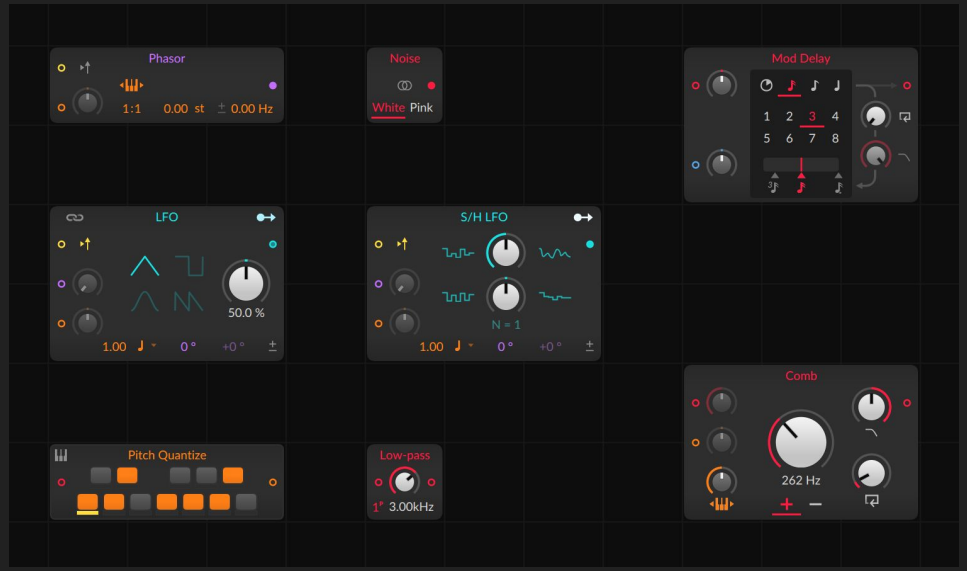
DSP Components

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 DSP component, such as:

- phase generator
- oscillator
- filter
- delay



A few useful Bitwig Grid nodes

Component: Phase Generator

Clock component that generates a control signal 0..1

Can be used as an input for a function or table to generate any periodic signal

Example function:

```
f(x) = sin(x*2pi)
```

Bitwig Grid phase generator with oscilloscope

Generating sine wave using phase generator

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

```
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

Component: Fast_table

Table with very 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

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

```cpp
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];
}
```

Translate More Components to C++


```cpp
// 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;
}
```


```cpp
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 More Components to C++


```
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);
};
```

DSP Components and Signal Graph

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

For example,

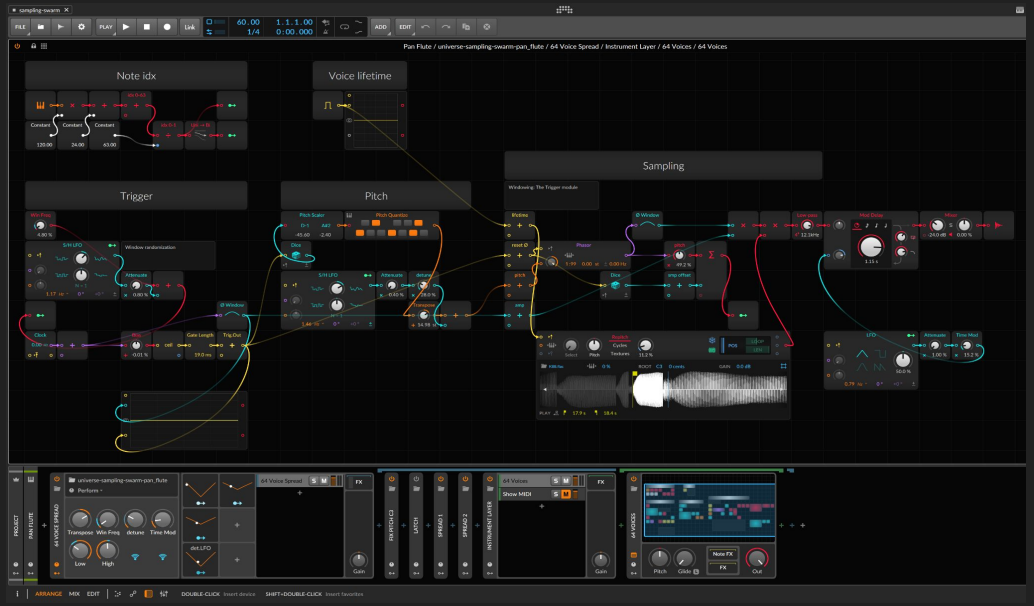
this graph can be rendered like this:

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

```
float osc = get_oscillator_output();
float osc_filtered = filter.process(osc);
float out = delay.render(osc);
```

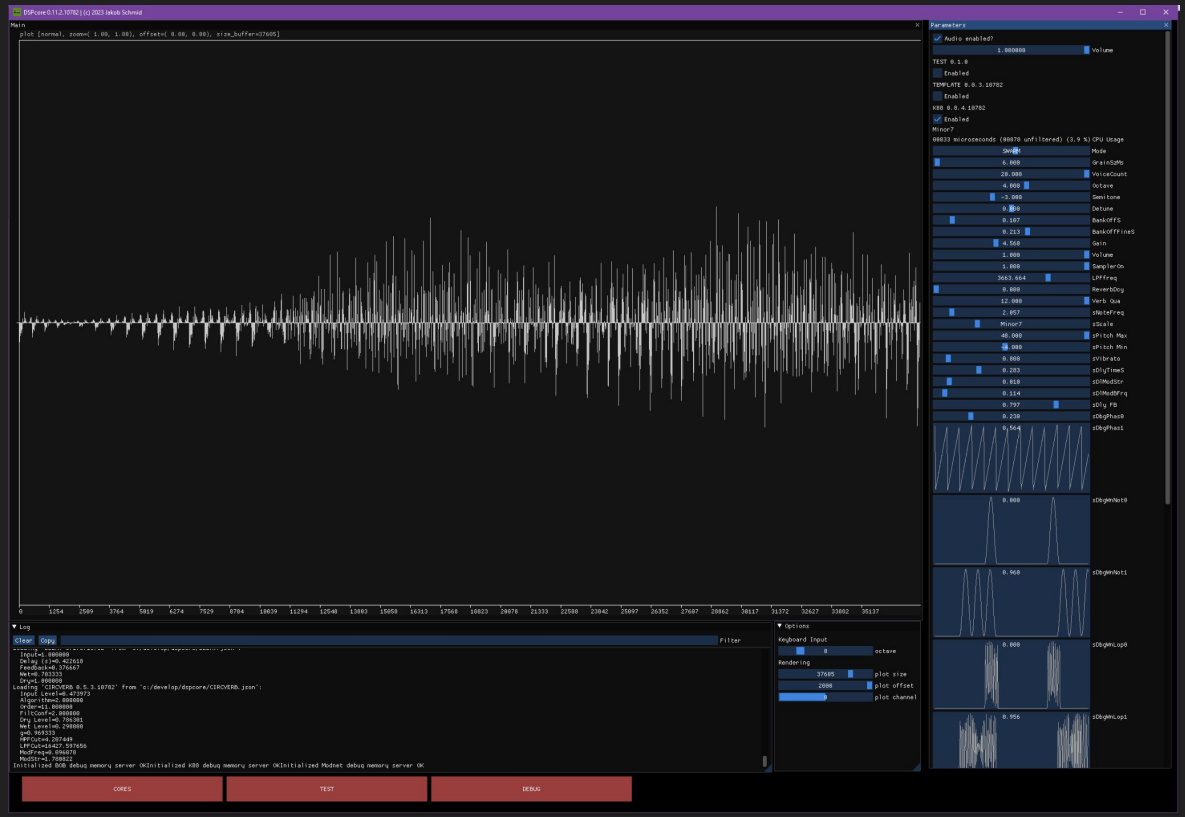
Example C++ signal graph implementation

Translate Signal Graph 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



dspcore

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_dsppparam_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
};
```

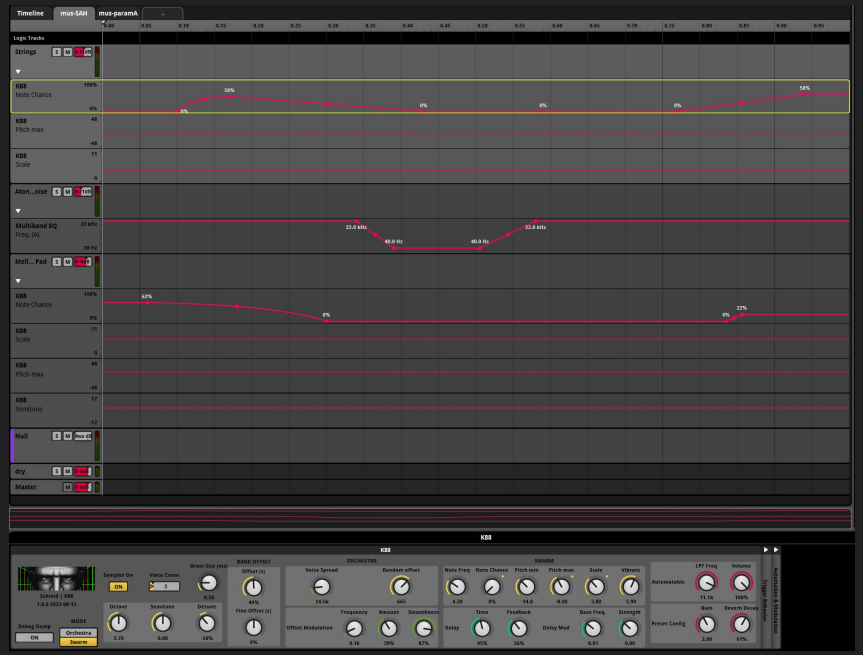
```
FMOD_RESULT F_CALLBACK Plugin_FMOD_dspprocess(
    FMOD_DSP_STATE *dsp, unsigned int length,
    const FMOD_DSP_BUFFER_ARRAY * inbufferarray, FMOD_DSP_BUFFER_ARRAY *outbufferarray,
    FMOD_BOOL inputsidle, FMOD_DSP_PROCESS_OPERATION op)
{
    PluginFMODState *state = (PluginFMODState *)dsp->plugindata;

    // ...

    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;
}
```

Wrap as FMOD Plug-in Instrument

- fmod-k88

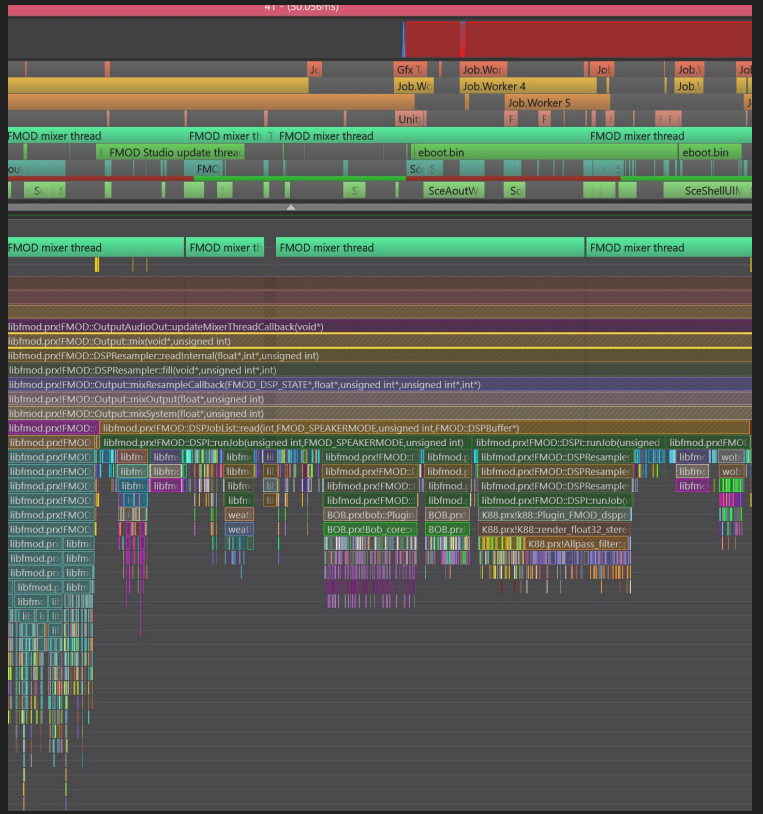
COCOON Plugin Instruments

Real-time Synthesized Music in COCOON



Real-time Synthesized Music on All Platforms

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



Closing Thoughts

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

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;
}
```

Some of the Topics that Required Research

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

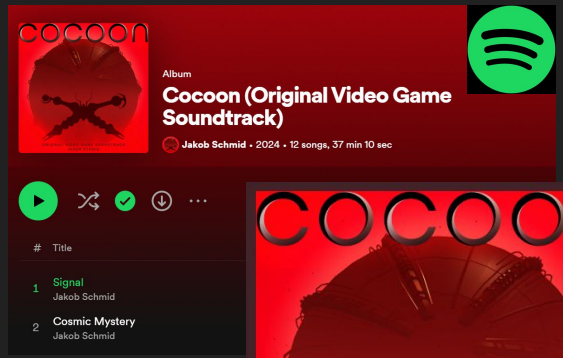
Ladder filter for resonant filtering

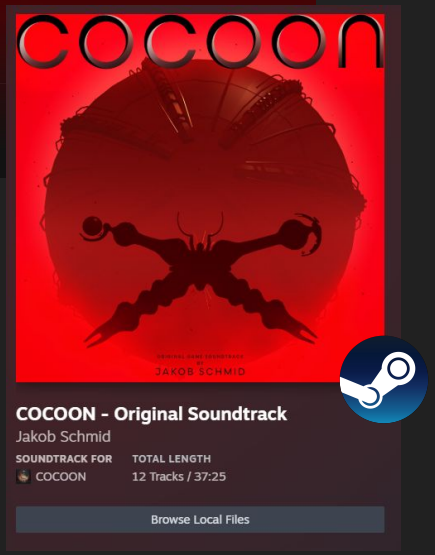
Simple reverb algorithm - a sequence of allpass filters can 'smear' a sound nicely

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

- *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)*

Questions?

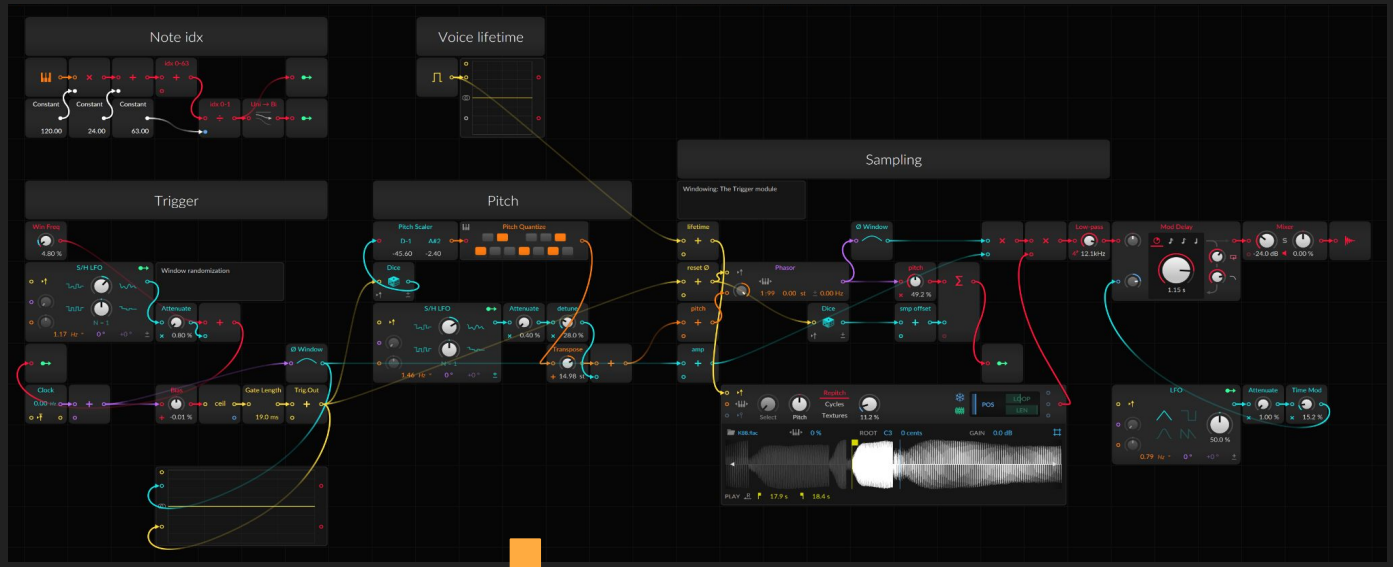




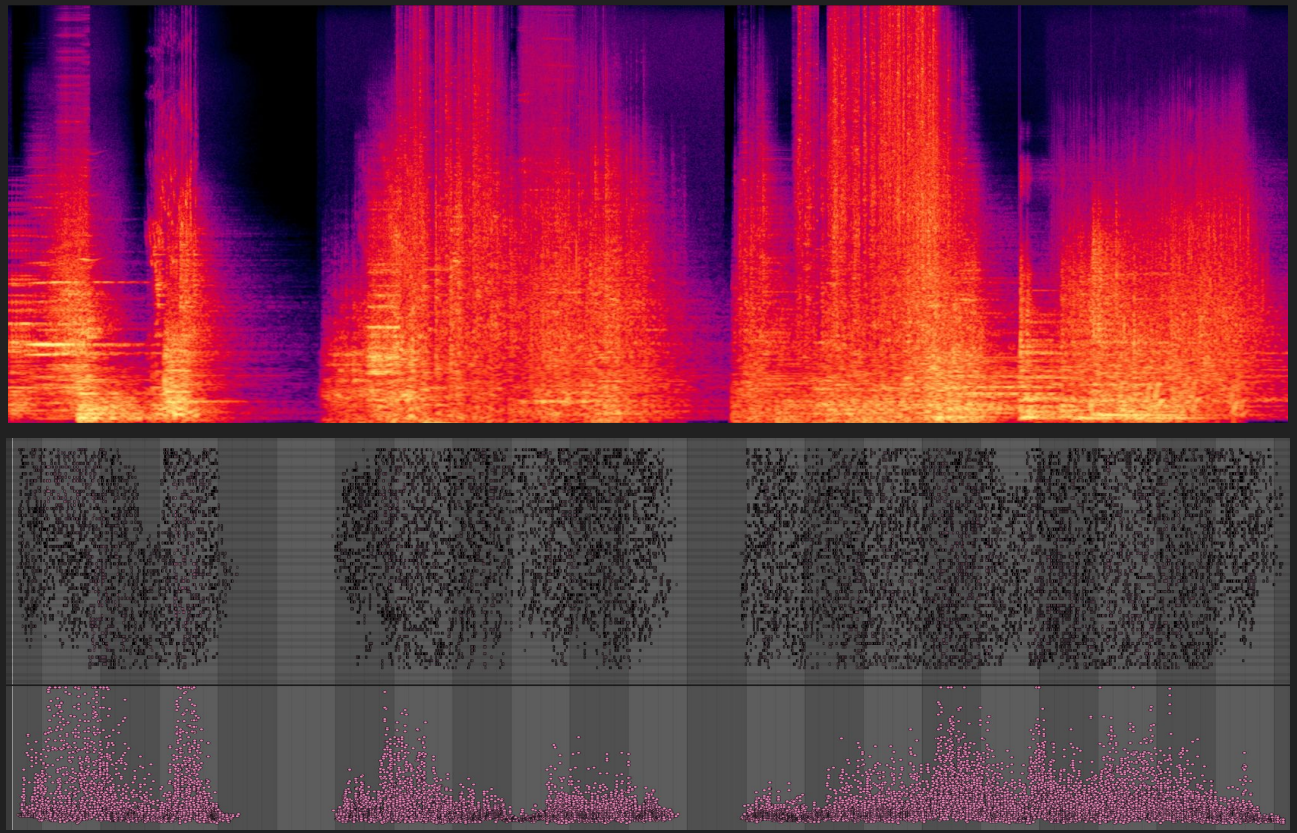
AVAILABLE NOW ON

STEAM XBOX NINTENDO SWITCH PS5 | PS4

Porting Bitwig Prototype to FMOD Studio Plugin



MIDI Vocoder: Dyson Gate

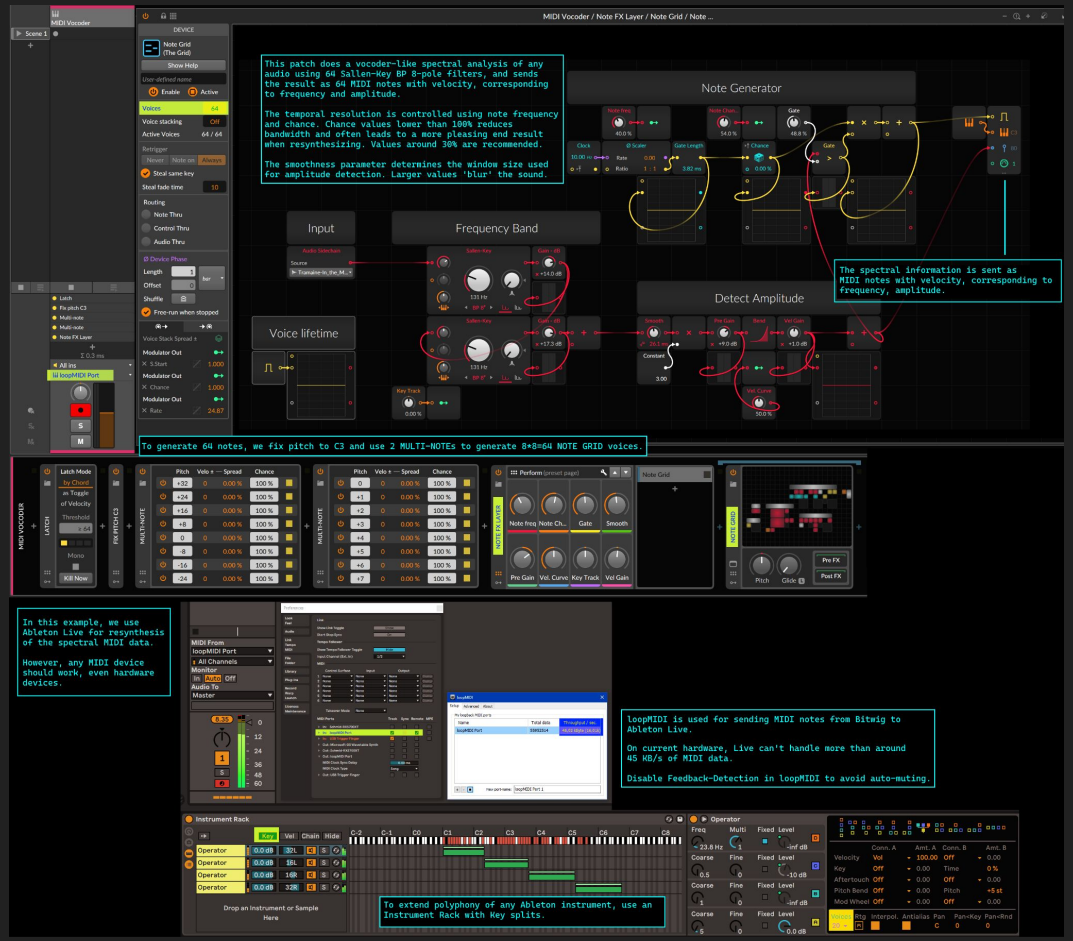


- midi_vocoder-bitwig, midi_vocoder-ableton, cocoon-gate

MIDI Vocoder

Home-made vocoder

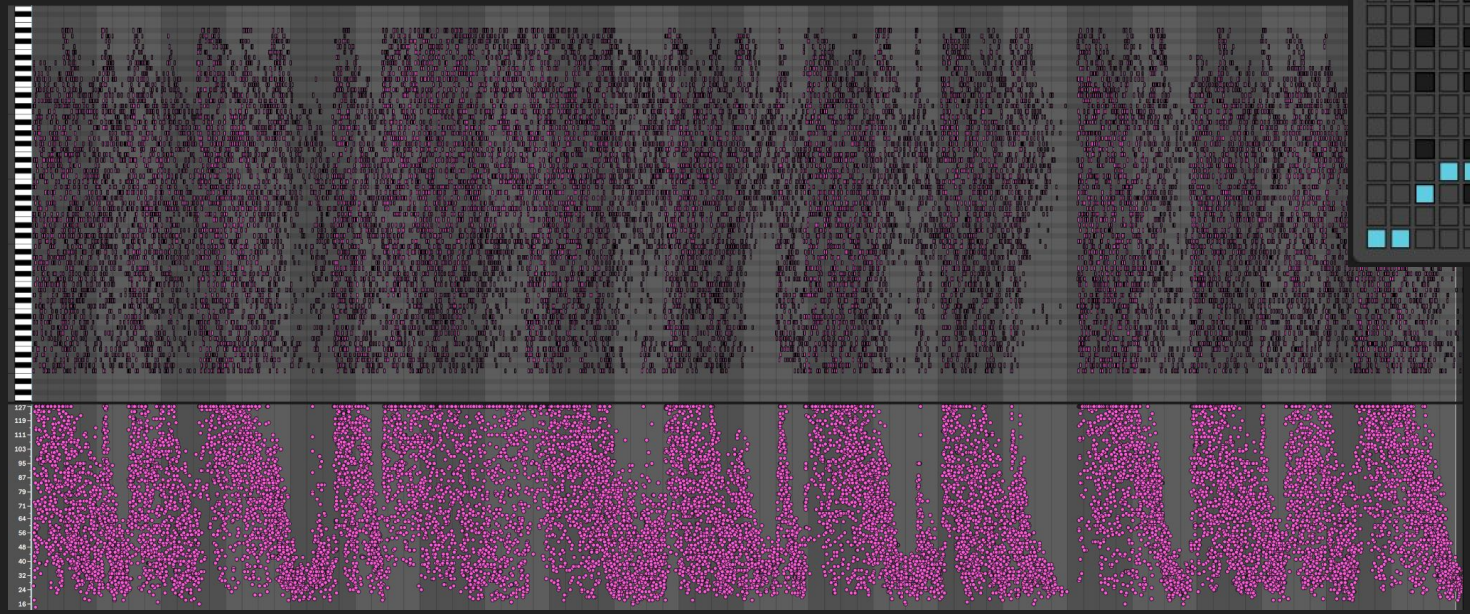
- Bitwig audio analysis
- MIDI sent via loopMIDI
- Record MIDI in Ableton Live

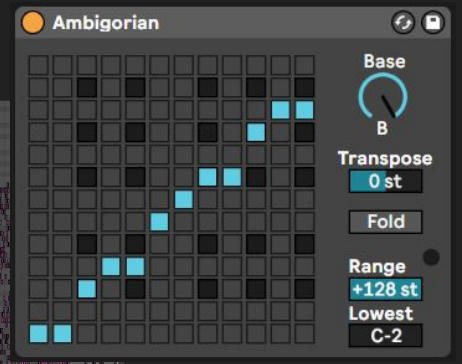


Puzzle Feedback Music

▸ cocoon-puzzfeed

MIDI Vocoder: Puzzle Feedback





▸ midi_vocoder-puzzfeed