GDC MARCH 17-21, 2025 SAN FRANCISCO, CA

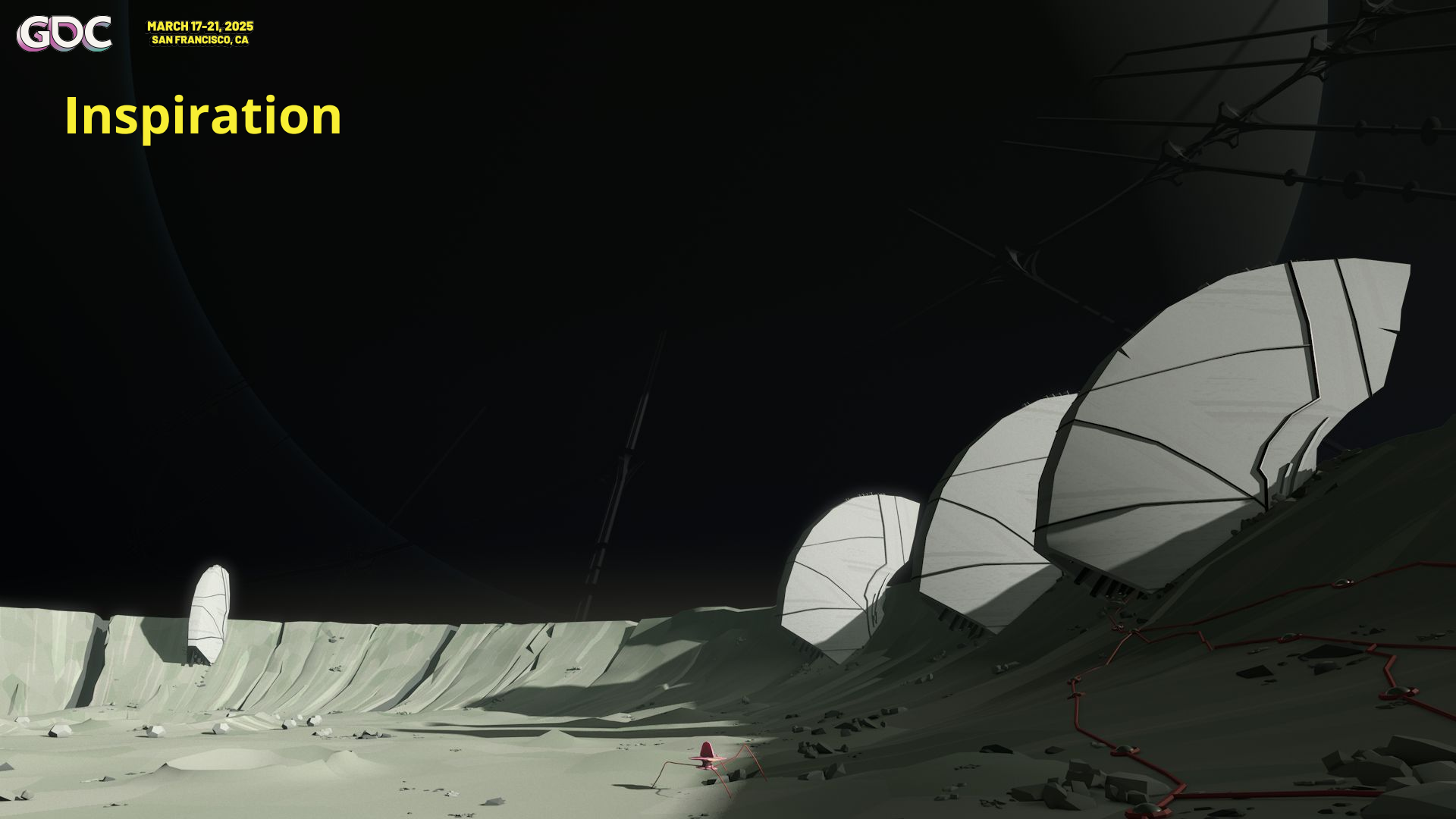
BONUS SLIDES!

The Synthesizer Plugins of COCOON

Jakob Schmid
Geometric Interactive

GEOMETRIC INTERACTIVE

Inspiration

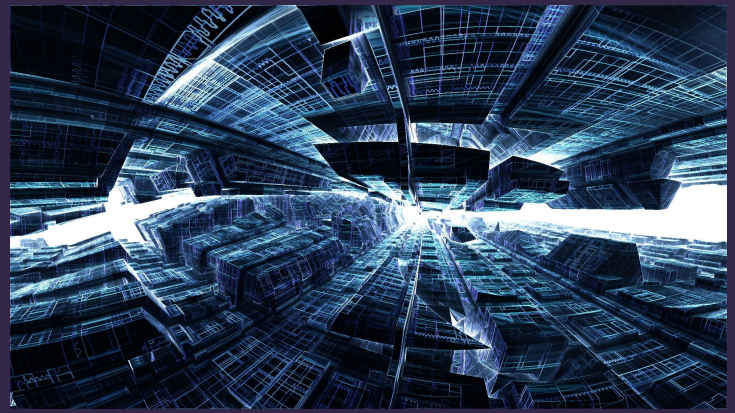


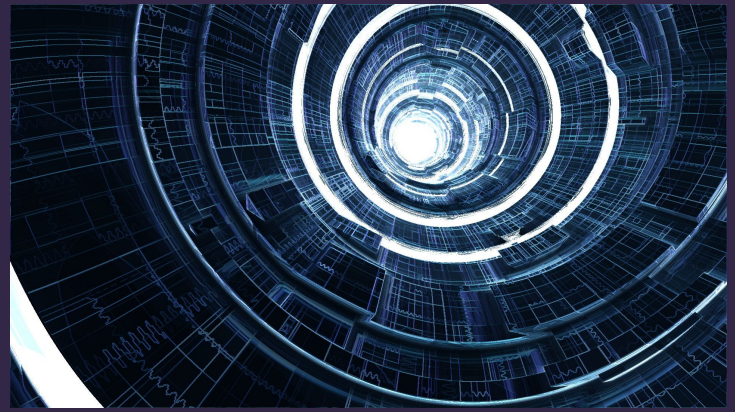
Inspiration

Quite & Orange: CDAK (2010)

Music by Lassi Nikko

4K demo

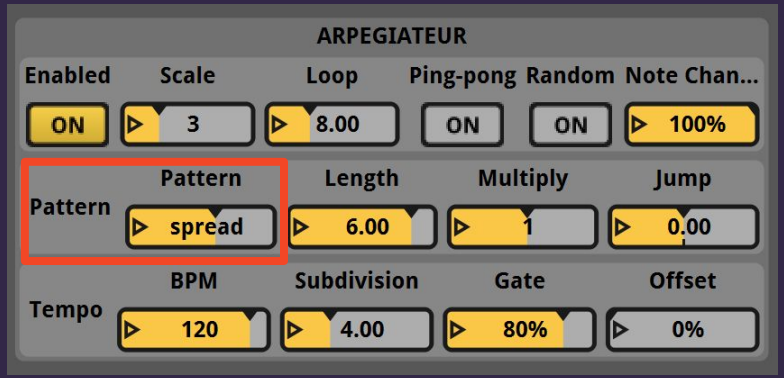




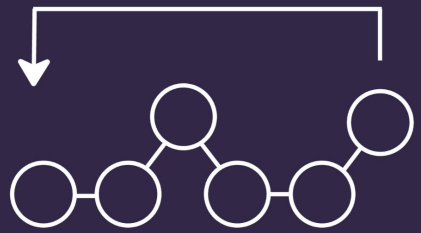
The COCOON Instruments

BOB Arpeggiator

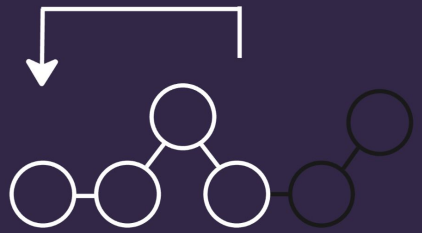
BOB Arpeggiator: Pattern



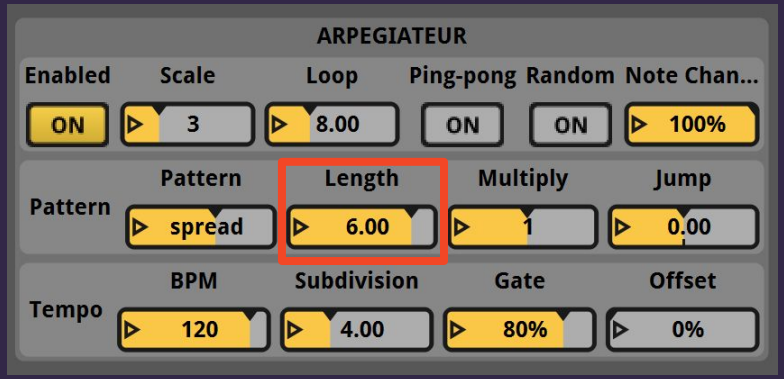
BOB Arpeggiator: Length



Length 6

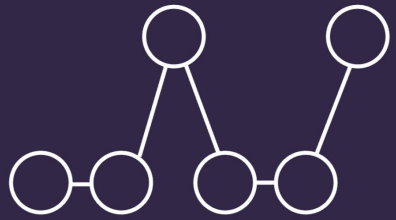


Length 4

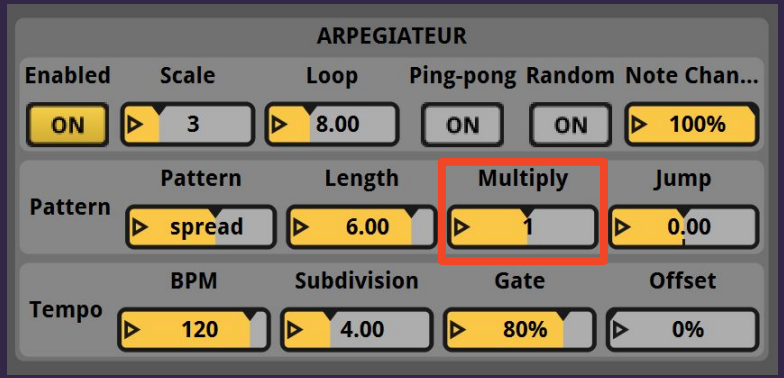


BOB Arpeggiator: Multiply

Multiply 1

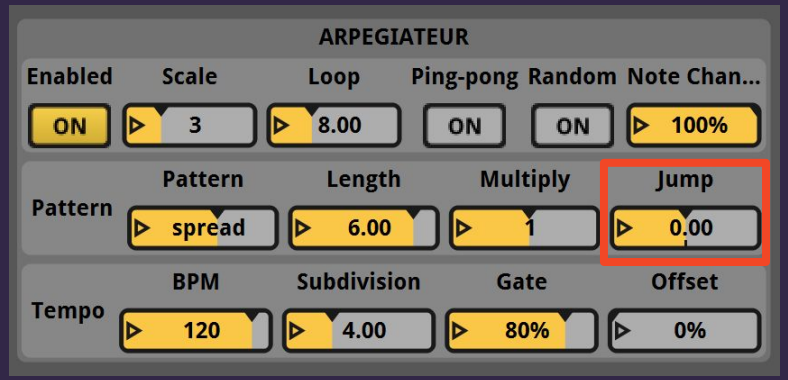


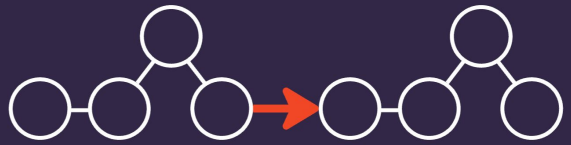
Multiply 2



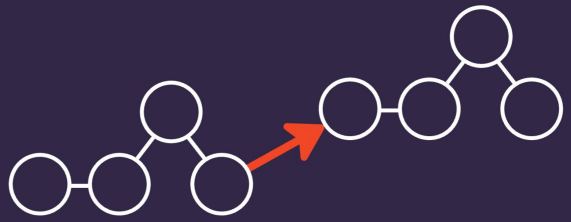
Multiply -1

BOB Arpeggiator: Jump





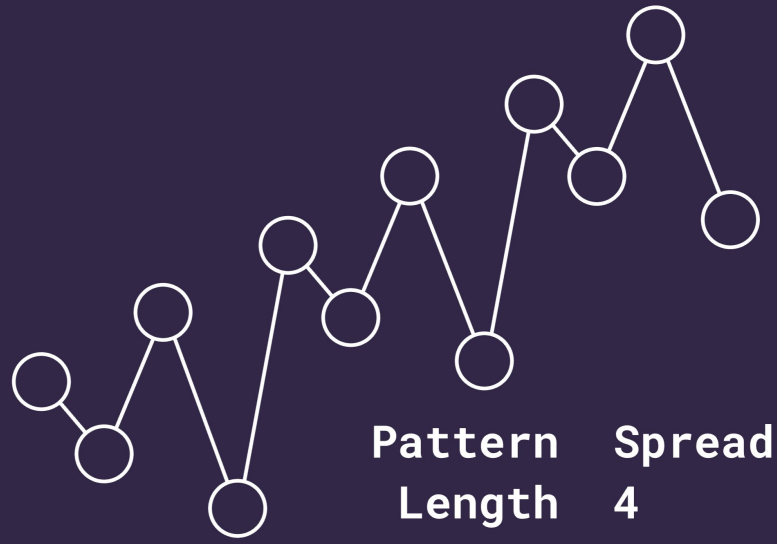
Jump 0

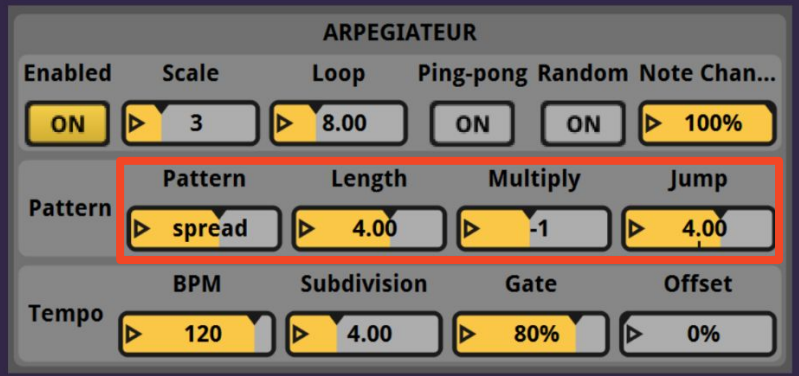


Jump 1

BOB Arpeggiator: Parameters

Example parameters





Composing with Plugins

Parameter-controlled Form

Boss Fights

FMOD Event Structure

Useful event structure for plugin-based music

Null channel :

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

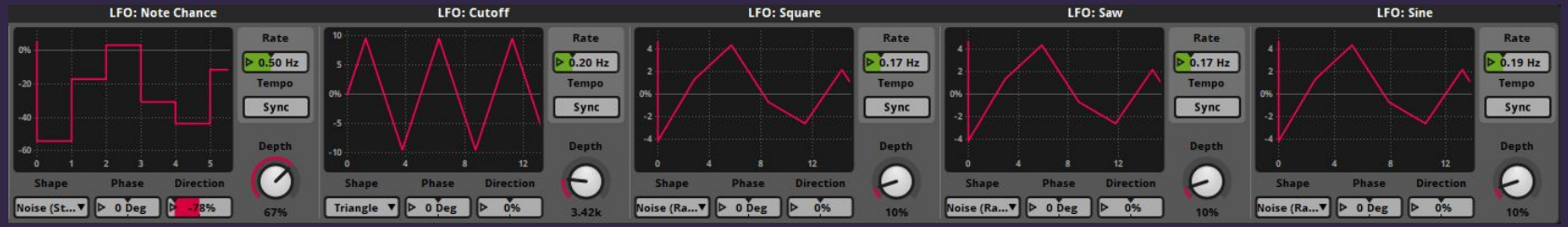
Dry output is a send same as the effects

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

Modulation Problems

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

Feels very dynamic and organic

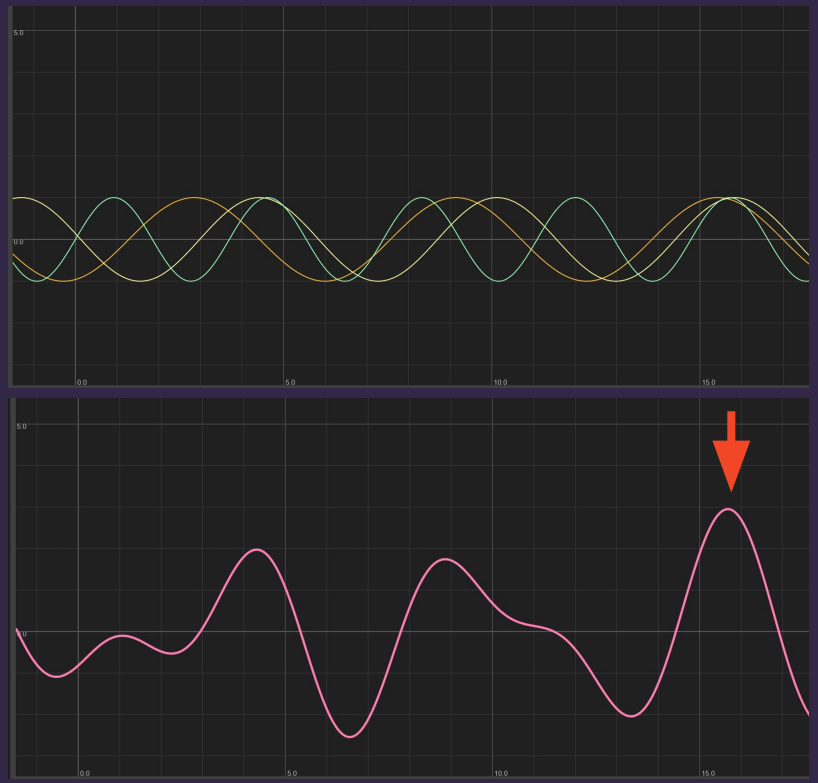


Constructive Interference

Combinations of modulators can produced unexpected results

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

Worst case, could cause clipping



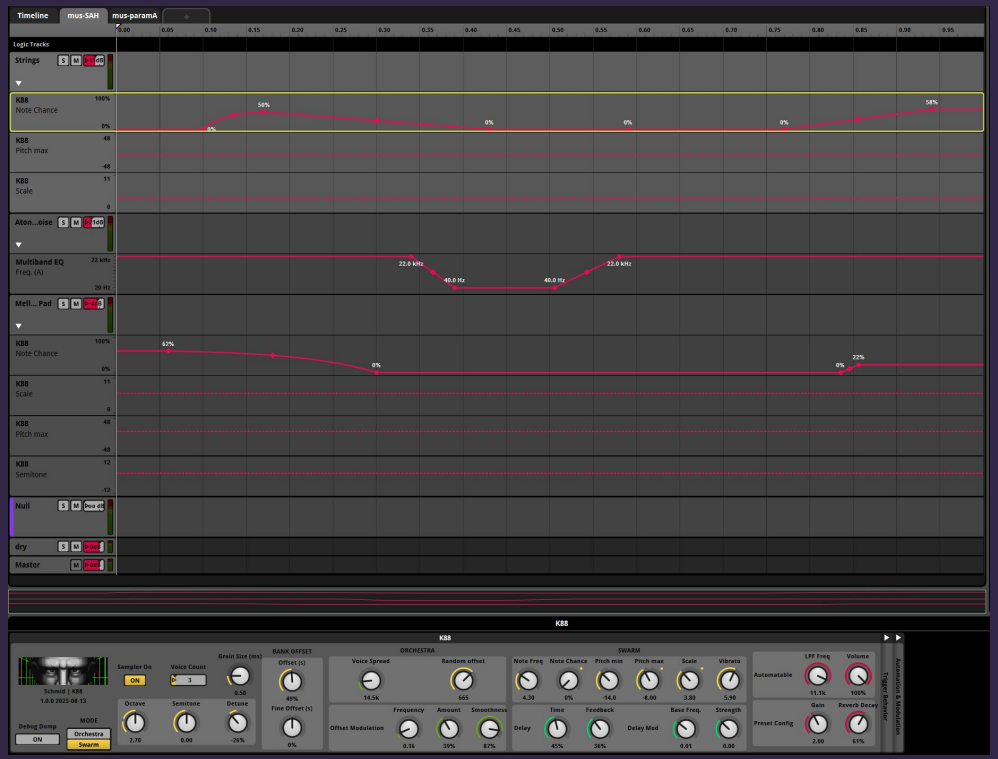
Parameter-controlled Form

Non-linear one dimensional score _

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

Define parameter sheet with all desired instrument configurations

Control using a single parameter

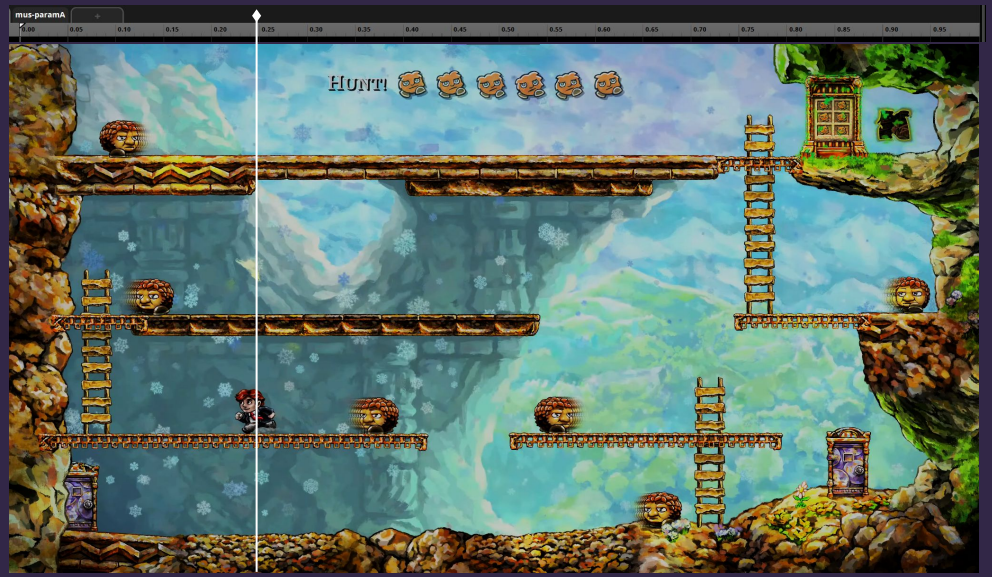


Parameter-controlled Form

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

where you scrub through a short musical piece

But with an FMOD parameter instead of Tim



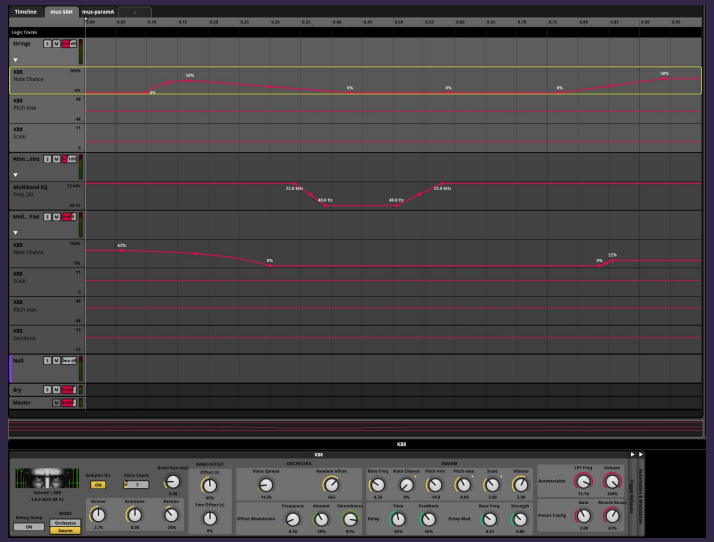
Parameter-controlled Form

Testable by manually scrubbing through the whole range

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

Automate everything including key, scale, timbre, effects

Note chance is useful for transitions



Green in Green: World Position

Key and pitch controlled from world position

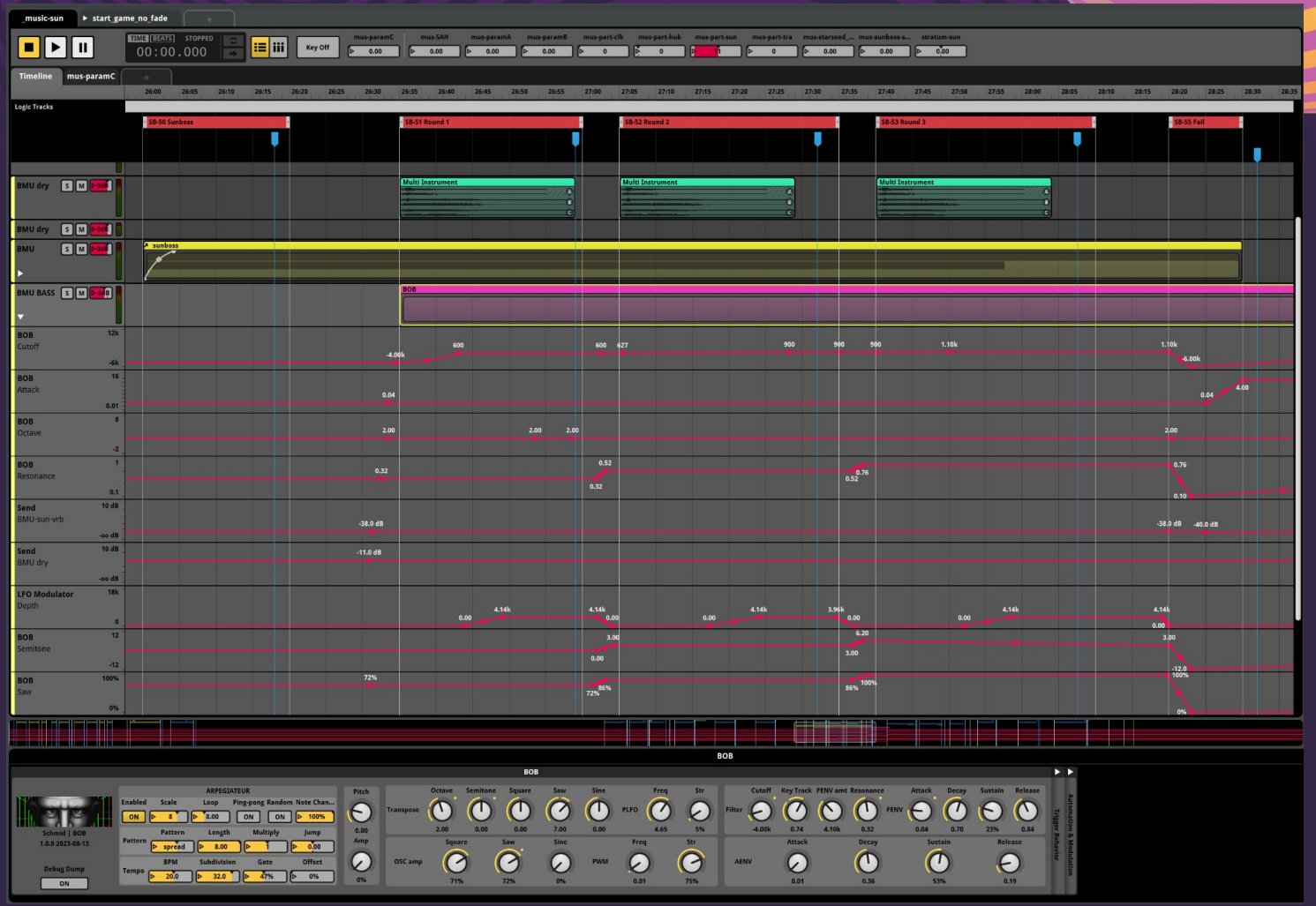
Cloak Boss Automation

intensity	Bass	arpeggio octave and filter frequency
	Insanity	vibrato
cloak	Bass	waveform mix, vibrato, filter frequency and resonance
	Insanity	octave, filter frequency
	Nightmare	octave, grain size, volume
bullet_fired	Bass	affects waveform mix and envelopes
impact	Bass	EQ filter frequency and delay feedback

▸ FIXME: which video?

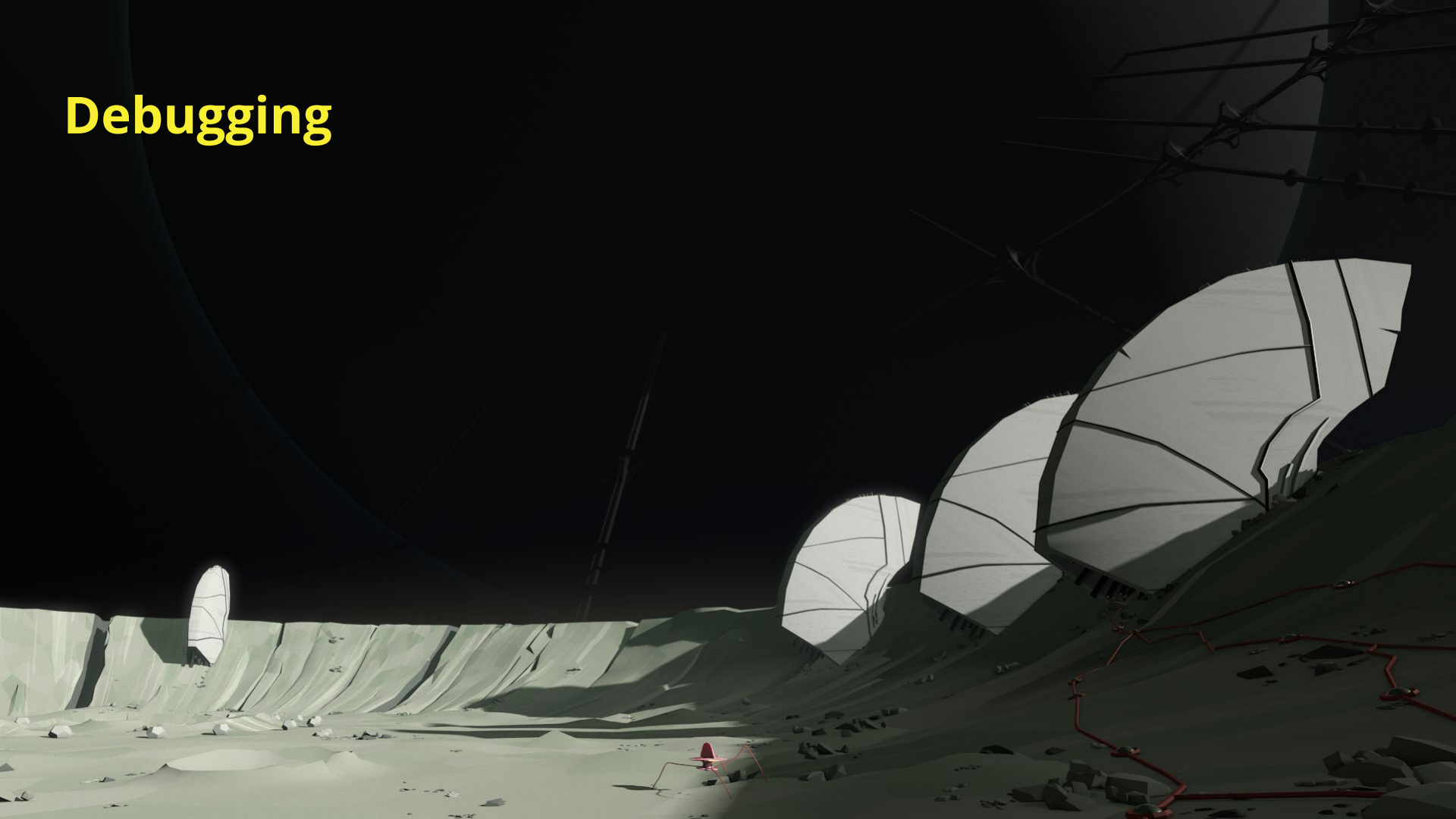
GDC MARCH 17-21, 2025 SAN FRANCISCO, CA

Sun Boss



▸ cocoon-cloak_boss.mkv

Debugging



Debugging in DSPcore.exe

Ideally, we wanted to debug running instances of plugins

both in FMOD Studio and in the running game

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

FileMapping	DSPcore.exe creates a local FileMapping using `CreateFileMapping`
OpenFileMapping	If it exists, plugin instances open it using `OpenFileMapping`
MapViewOfFile	File is mapped to a memory buffer using `MapViewOfFile`
Read/write	Now that the memory is shared, plugins can write, and DSPcore.exe can read

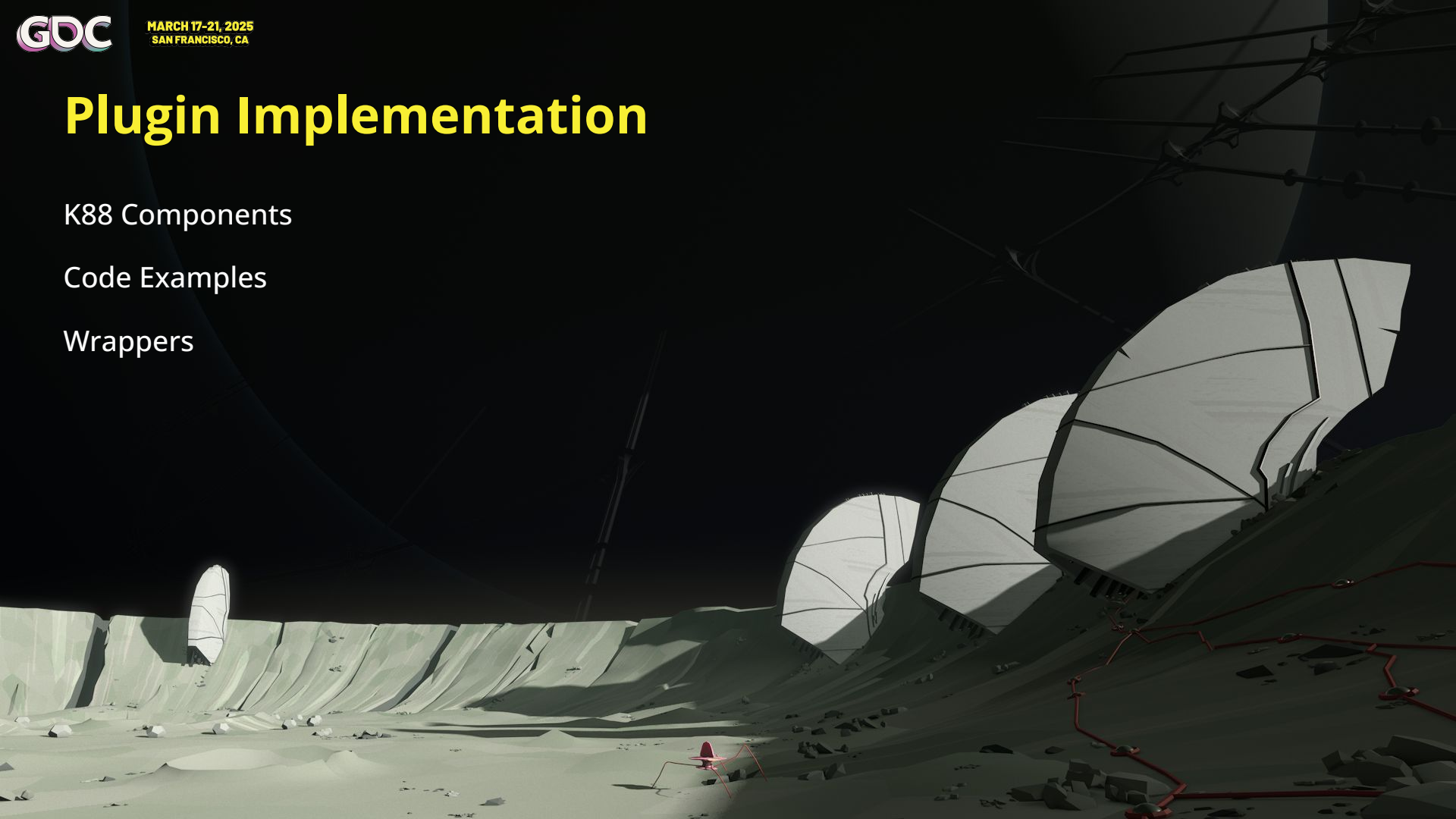
```
const int buf_size = sizeof(Buffer);
Buffer* buf;
HANDLE map_file;
const char* map_name = "Local\\MyApp";

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);
}
```

Plugin Implementation

K88 Components

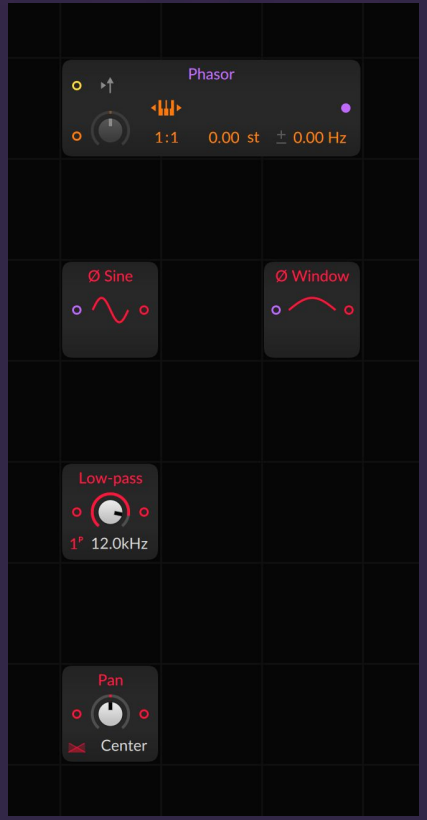
Code Examples

Wrappers



K88 Components

Phasor	Phase generator component, generates a control signal 0..1
LUT	Lookup table combines with phase generator to make oscillators or grain windows
Low-pass filter	Remove unwanted high frequency content with simple 1-pole LPF
DC filter	to avoid build-up of DC offset
Panner	Constant power panner for spreading voices in the stereo field



K88 Components

Sampler	Sampler with linear interpolation
Sample and hold	Sample/hold LFO
Modulation delay	Modulation delay based on circular buffer with linear interpolation
All-pass filter	All-pass filter based on circular buffer
Pitch quantizer	Quantizes arbitrary frequencies to selected scale

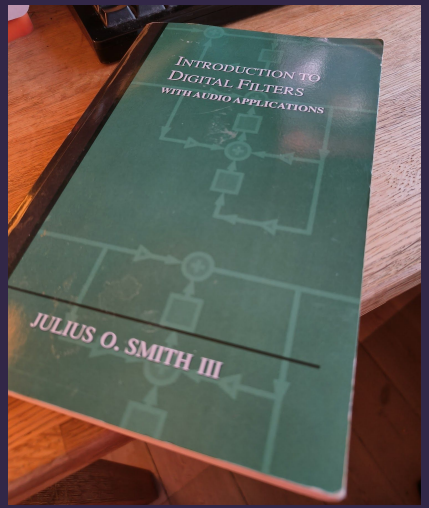
Core Component: DC Filter

DCF

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

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

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



Phase Generator Implementation

Effective implementation using 32-bit unsigned integer as clock counter

Modular arithmetic using the 'wrapping' type uint32_t

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

Update method is a single line:

```
phase += freq;
```

Frequency is represented as phase increment per update

Phase Generator Implementation

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

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

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

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

```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];
}
```

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;
}
```

Modulation Delay


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

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

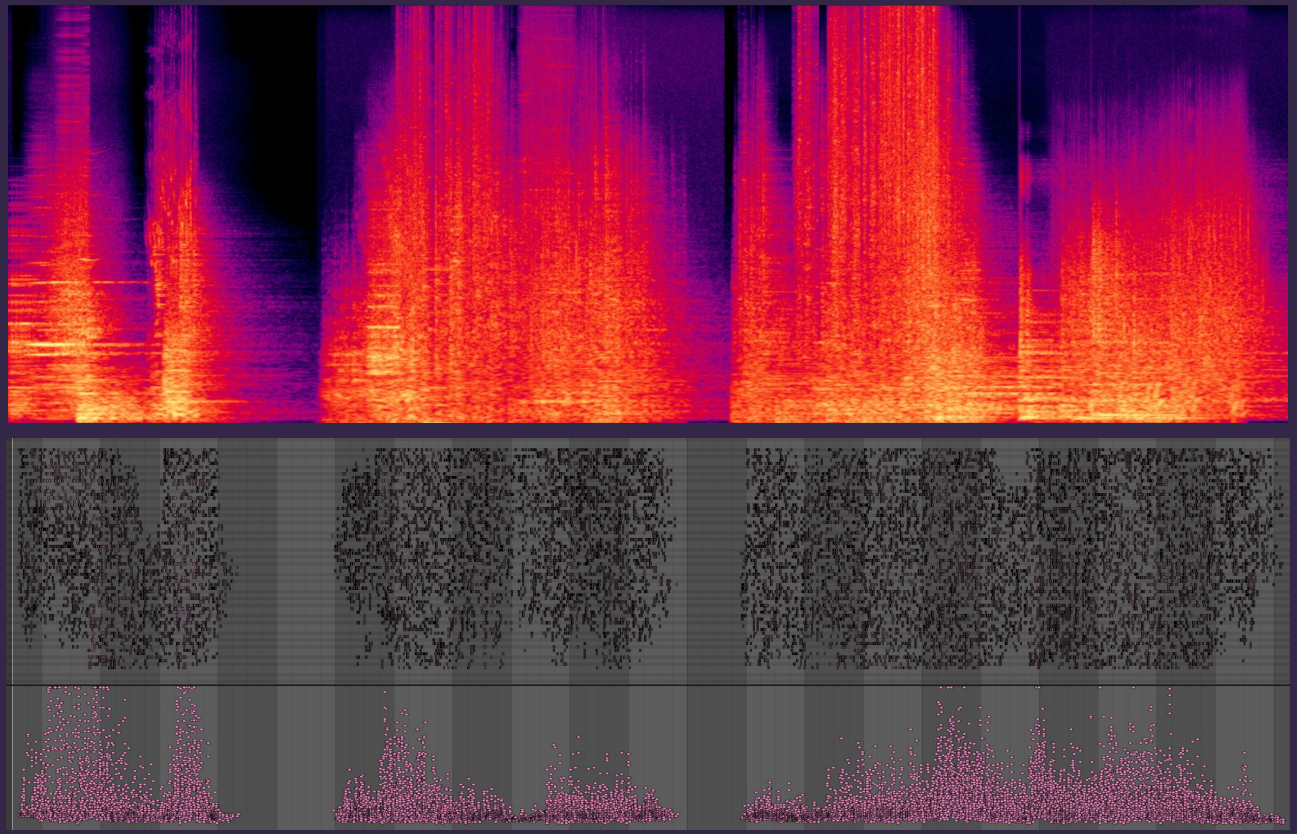
MIDI Vocoder

MIDI Vocoder

Home-made vocoder

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

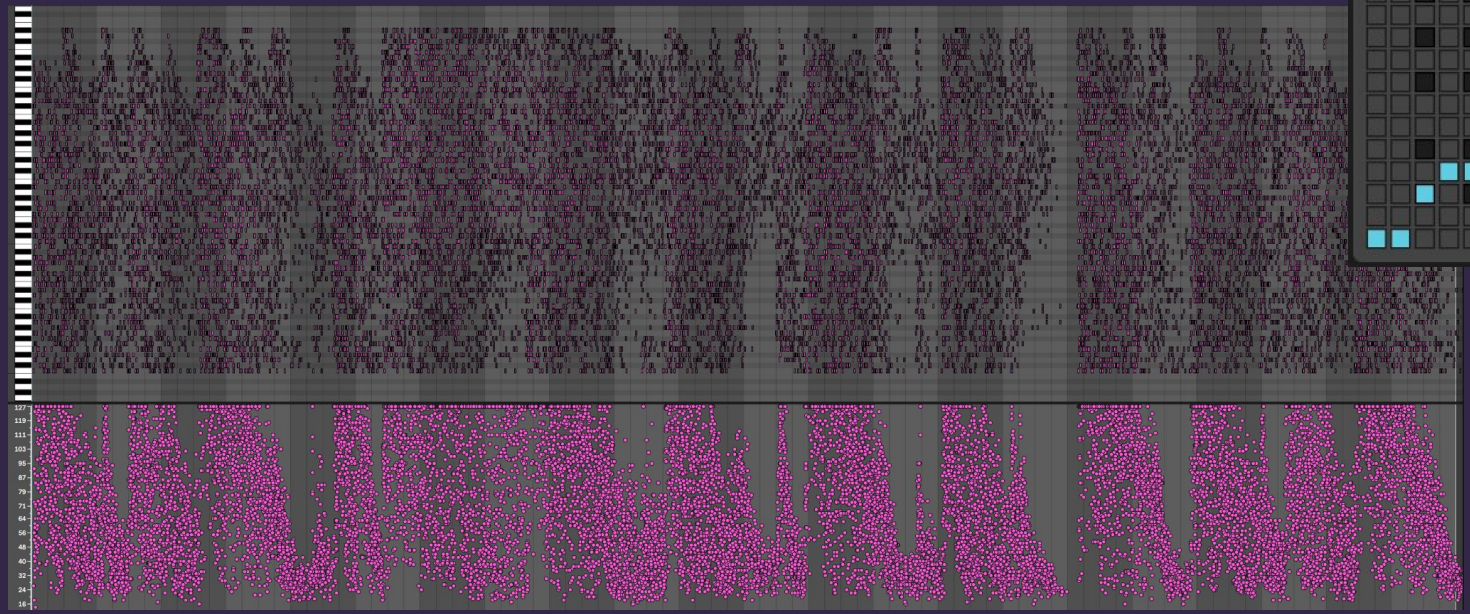
MIDI Vocoder: Dyson Gate

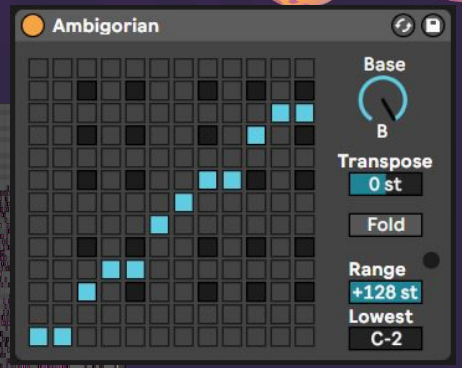


▸ midi_vocoder-bitwig, midi_vocoder-ableton, cocoon-gate

Puzzle Feedback Music

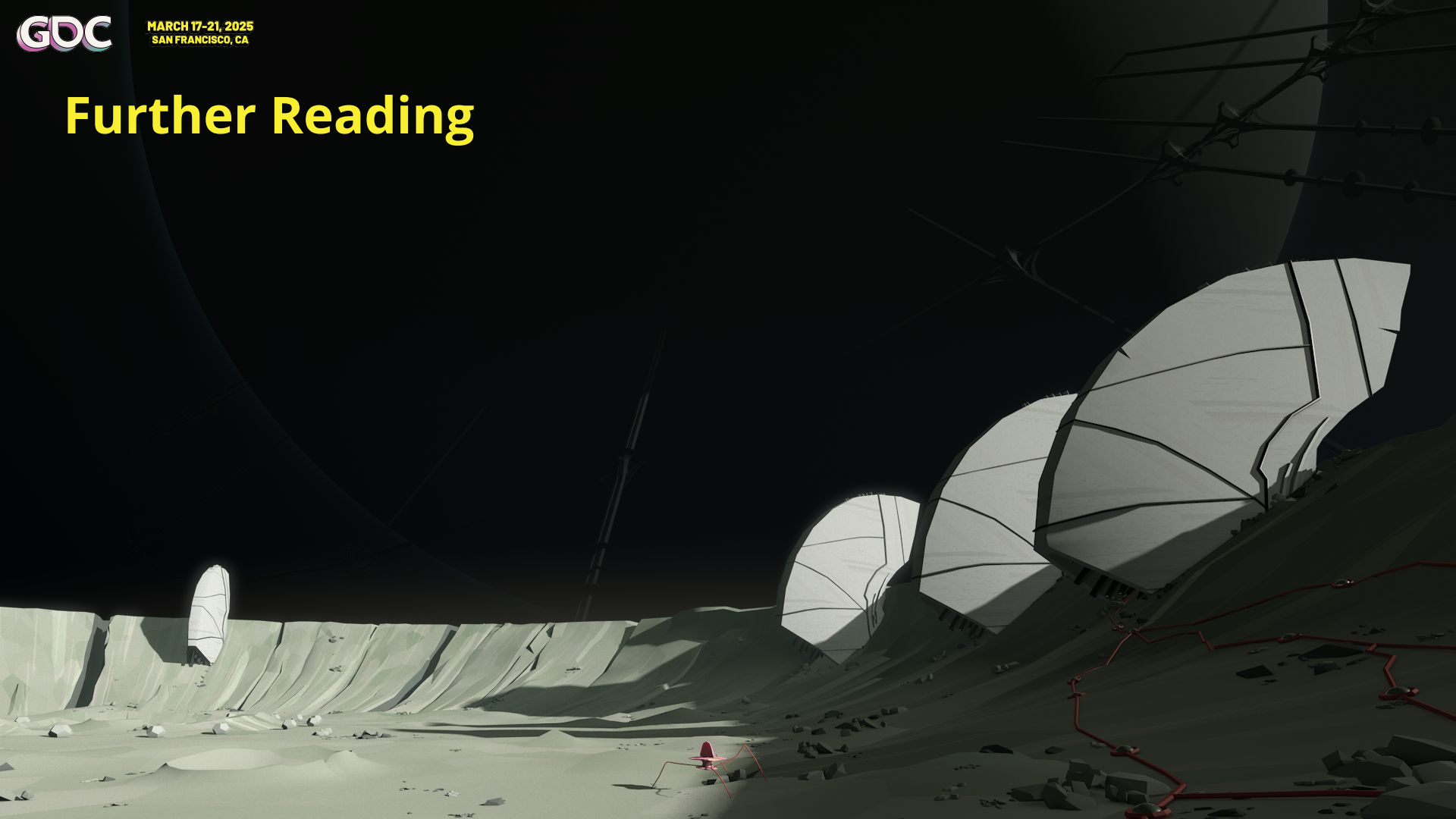
MIDI Vocoder: Puzzle Feedback





▸ midi_vocoder-puzzfeed

Further Reading



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)

schmid-cocoon-gdc-bonus-2025-03-25-1546.pdf