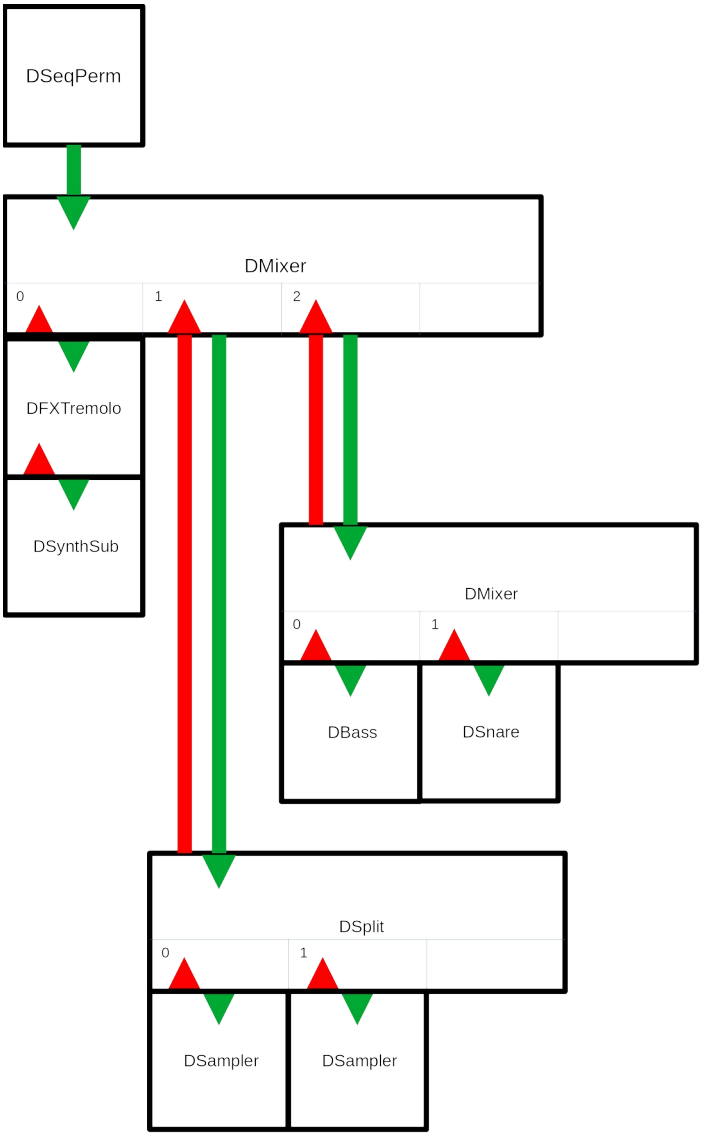
DSTUDIO




```
Project: DStudio – addons and examples to make music with openFrameworks
Author: Staffan Melin, staffan.melin@oscillator.se
License: GNU General Public License v3.0. DaisySP is licensed under the MIT
license.
Version: 1.1.0 (20221020)
Project site: http://oscillator.se/opensource
Source:
https://github.com/StaffanMelin/ofxDStudio
https://github.com/StaffanMelin/ofxDaisySP
```

Introduction

Welcome to the DStudio – addons for creating music and sounds with openFrameworks!

At its core DStudio combines the ofSoundStream class of openFrameworks with the DSP library DaisySP from Electro-smith. Although DaisySP is originally made for a specific hardware it uses the same Processing model that openFrameworks does, so they work very well together. What DStudio does is to provide another abstraction layer where you don't deal with soundstreams and filters but with synthesizers, mixers, plugins and sequencers. Like a studio made with code!

DStudio is:

- **ofxDaisySP** – an addon with DaisySP, a DSP library originally created for the Daisy Seed family of audio oriented hardware. Thank you Electro-smith for making this!
- **ofxDStudio** – an addon which builds upon the DaisySP library, that gives you synthesizers, a sample player, drum sounds, FX plugins, a mixer with FX and sequencers as well as utilities. All can be controlled by MIDI.
- **examples** – examples that shows you how to use DStudio.

```
If you find any bugs or have suggestions for improvements, please contact
me. And if you make some interesting music/noise, I would be very happy to
hear about it!
```

If you think DaisySP sounds good, check out my projects using the Daisy Seed MCU: The OscPocketD family! You can find demos, build instructions and code on my site: https://www.oscillator.se/opensource/#daisy.

New in this version

DMixer

(BREAKING CHANGE) The DMixer wet/dry properties didn't do what they were supposed to. The mixer config struct has three new members:

- float chorus_return - typical value 0.5
- float reverb_return - typical value 0.5
- float mix_dry - typical value 0.5

The following methods are now deprecated:

- SetChorusSend()
- SetReverbSend()

And the following methods are added:

- SetShorusReturn()
- SetReverbReturn()
- SetMixdry()

DSynthSub

New method:

- SetEGLevel() - set level of EG (PITCH or FILTER) for selected target (ie how much the EG should influence the sound)

DSynthVar

New methods:

- SetEGLevel() - set level of EG (PITCH or FILTER) for selected target (ie how much the EG should influence the sound)
- SetMod() - set all mod targets
- SetSM() - set mod parameters
- SetSMSeq() - set sequence for mod. Note that if seq len > 0 there must be at least 1 element in seq_val
- SetSMSeqStep() - set sequence step value

Changes:

- SetLFO() - now has an offset argument

DSampler

New methods:

- GetLength() - return length of sample
- SetEGLevel() - set level of EG (PITCH or FILTER) for selected target (ie how much the EG should influence the sound)

Information about sample moved into config struct:

```
bool loop;

std::string sample_file_name;

uint32_t sample_phase_start;

uint32_t sample_phase_loop_start;

uint32_t sample_phase_loop_end;

uint32_t sample_phase_end;

uint32_t sample_length;

uint8_t sample_channels;
```

The variables sample_phase_start, sample_phase_loop_start, sample_phase_loop_end, sample_phase_end and sample_length are set in DSampler::Load(). They are set to play and loop the whole sample. To avoid this, for example when loading a preset, pass false as the second parameter to Load() -- which is what you normally should do. If you want to base these parameters on the sample itself, pass true.

Presets

You can now save and load presets of all synthesizers and drums using the DSettings static class. They are stored as XML files.

You must use the addon ofxXmlSettings for this to work.

There is no error checking on loading presets.

The example-edit program lets you explore all settings for synthesizers and drums and save and load settings for use in any project.

See example-edit and example-10-generative-space-music.

FX: Panner

Two new effects. They are both used in example-4-sampler.

Panner is an autopanner with 3 modes (called types) and a built in LFO:

- type == DFXPanner::STATIC: offset is pan value (0.0 - 1.0)
- type == DFXPanner::LFO: offset is added to the LFO signal

- type == DFXPanner::RANDOM: A new random value is generated when the LFO has changed more than the offset value. Amplitude is the width of generated values.

Panner is a stereo effect that preferably operates on mono DSounds.

FX: Slicer

Used in example-4-sampler.

The slicer samples a random length of the audio, and repeats it random times, then starts anew.

Example: example-9-visualizer

This is a sketch of how to integrate visuals based on MIDI.

Example: example-10-generative-space-music

This is an example of how to make generative music using a DGen class.

It creates never ending music based on simple MIDI note input and random changes.

Channels can be of different types that affect how notes for that channel is generated:

- DGenDrone::BASS: low droning notes
- DGenDrone::TREBLE: high droning notes
- DGenDrone::PAD: chords
- DGenDrone::MELODY: melody
- DGenDrone::ARPEGGIO: arpeggio
- DGenDrone::EMBELLISH: quick details

The DGenDrone class works on some note input:

- dgen_note_base: a vector with the base MIDI note of each channel
- dgen_note_pad: notes that are used for the channel type PAD. The values are relative to the base note for the channel.
- dgen_note_arp: notes that are used for the channel type ARPEGGIO. The values are relative to the base note for the channel.
- dgen_note_melody: notes that are used for the channel type MELODY. The values are relative to the base note for the channel

The algorithm uses states, called drama; INTRO, VERSE, CHORUS, BREAK, OUTRO. The DGenDrone::NoteCreate() method creates notes for a specific channel type and drama state. Notes are queued and sent when the time is right.

Transitions from one drama state to another is handles by a drama_order_ vector, indexed on drama state with three values that indicates which transitions are possible (with 60%, 30% and 10% probability respectively).

Every channel can have a drama_fade_ value. This indicates how long (in specified fraction of drama length) should be used for a fade in or fade out.

The example also shows how to work with presets.

Example: example-edit

This example creates GUIs for all synthesizers and drums and lets you edit parameters in real time.

You can play the selected sound with the keyboard (imagine a keyboard starting at C on key Z and running to the right one octave). You can also attach a MIDI keyboard so this transforms the example into a live synth with 4 sound engines and drums!

It uses the static class DSettings for saving and loading presets.

When saving and loading drum sounds you have to select which sound to save/load. Type the inital character of the sound, except use Y for cymbal.

Misc

- Drum sounds now respond to MIDI pitch values.
- A lot of bug fixes.

Table of Contents

How to use DStudio

DStudio is a collection of classes.

The majority of them create sounds and are all derived classes of the DSound class:

- classic synthesizers: DSynthSub (a classic subtractive two oscillator synth), DSynthFm (an FM synth), with filter, envelopes for pitch, filter, amplitude, an LFO that can affect pitch, filter, amplitude, delay and overdrive
- experimental synthesizer: DSynthVar (a variable shape synth), with filter, envelopes and unique modulation with different random modes as well as a modulation sequencer, delay and distortion
- sampler: DSampler, with filter, envelopes, LFO, delay and distortion
- drum sounds: DBass, DClap, DCymbal, DDrum, DHihat and DSnare, most with three different sound generating algorithms, all with lots of options for shaping the sound

They can be combined using the mixer class DMixer. The mixer can pan and adjust levels of all channels, as well as providing two send channels for reverb and chorus.

You can use the sequencer, DSeqMidi, to sequence them. Or you can use and control them directly in your code.

You can also use the permutating sequencer DSeqPerm. Feed it with sequences and let it provide you with ever changing results!

The header file dstudio.h is a global settings file that contains some common settings, most notably

```
#define MIXER_CHANNELS_MAX 16
```

which determines the max number of channels in the mixer, DMixer. You can run one sound object on each channel, but all channels can be a mixer so you can chain an infinite number of sounds.

All objects follow the same pattern. You create a configuration struct, set it up and pass it to the Init() method.

The Process() method is called for each object to make do its work (produce sound).

If you are using the mixer it will take care of calling all sound generating objects for you.

All classes derived from DSound can be of different types and have a get and set method for working with this: GetType() and SetType():

- TUNED - used and set as default for all tuned instruments, ie synths
- PERCUSSION - used and set as default for all drum sounds
- FX - used and set as default for all FX plugins
- MIXER - analyses the incoming MIDI and sends it to the channel indicated by (in) the MIDI message

- MIXER_SUB - a submixer is a mixer put on one channel of the main mixer. When a DMixer of this type receives a MIDI message it sends it to all instruments on every channel. Use this to build fat patches.
- MIXER_PERCUSSION - looks at incoming MIDI and sends it on to the channel indicated by the NOTE value - 36 (the 36 comes from the constant MIDI_PERCUSSION_START defined in dstudio.h. This way you can create a drum set using a DMixer of type MIXER_PERCUSSION.

For details on how to use the classes, see the reference section, the corresponding header files and the demo projects.

Installation

Install openFrameworks (https://openframeworks.cc/download/).

Install the ofxDaisySP and ofxDSound directories into the addons directory in openFrameworks.

Install the demo programs into the apps/myApps directory in openFrameworks.

It should look like this:

```
of/addons/ofxDaisySP
of/addons/ofxDStudio
```

and

```
of/apps/myApps/ofDStudioDemo0
of/apps/myApps/ofDStudioDemo1
```

DStudio is developed on GNU/Linux Debian 11 Testing. openFrameworks version 0.11.2, IDE QT Creator 7.

Basic tutorial

Lets create a subtractive synthesizer that you can play on the computer keyboard.

Start QT Creator (or whatever IDE you are using) and create an openFrameworks project named ofDSoundBasic in the myApps folder. Close the project.

Add a file called addons.make:

```
ofxDaisySP
ofxDStudio
```

Edit the ofDSoundBasic.qbs file and add the DSound libraries:

```
        of.addons: [
            'ofxDaisySP',
            'ofxDStudio'
        ]
```

Delete the ofDSoundBasic.qbs.user file and open the project again in the IDE.

Edit the ofApp.h file. Delete the methods we don't need. Add the basic necessary includes, a DSynthSub object, the audioOut() method and a ofSoundStream object.

```
#pragma once

#include "ofMain.h"

#include "dsound.h"
#include "dsynthsub.h"

class ofApp : public ofBaseApp{

private:
    DSynthSub dsynthsub; // a subtractive synthesizer
    uint8_t note;

public:
    void setup();
    void update();
    void draw();

    void keyPressed(int key);
    void keyReleased(int key);

    void exit();

    void audioOut(ofSoundBuffer &outBuffer);
    void keyReleased(int key);

    ofSoundStream soundStream;
};
```

Edit the ofApp.c file. Delete the methods we don't need. Add the basic necessary includes, the soundstream configuration, the DSynthSub object configuration, start the ofSoundStream object. Add content for the audioOut() method and the keyPressed() and keyReleased() methods.

```
#include "ofApp.h"
#include "dstudio.h"
#include "dsynthsub.h"

void ofApp::setup()
{
    // configure soundstream
    ofSoundStreamSettings settings;
    settings.numOutputChannels = 2;
    settings.sampleRate = DSTUDIO_SAMPLE_RATE;
    settings.bufferSize = DSTUDIO_BUFFER_SIZE;
    settings.numBuffers = DSTUDIO_NUM_BUFFERS;

    // configure subtractive synthesizer
    DSynthSub::Config dsynth_config;
    dsynth_config.sample_rate = settings.sampleRate;
    dsynth_config.voices = 1;
    dsynth_config.waveform0 = DSynthSub::WAVE_POLYBLEP_SAW;
    dsynth_config.waveform1 = DSynthSub::WAVE_POLYBLEP_TRI;
    dsynth_config.tune = 0.0f;
    dsynth_config.detune = 0.0f;
    dsynth_config.transpose = 0;
    dsynth_config.osc0_level = 0.3f;
    dsynth_config.osc1_level = 0.3f;
    dsynth_config.noise_level = 0.0f;
    dsynth_config.filter_type = DSynthSub::LOW;
    dsynth_config.filter_cutoff = 800.0f;
    dsynth_config.filter_res = 0.2f;
```

```
    dsynth_config.eg_p_level = 0.0f;
    dsynth_config.eg_p_attack = 0.0f;
    dsynth_config.eg_p_decay = 0.0f;
    dsynth_config.eg_p_sustain = 0.0f;
    dsynth_config.eg_p_release = 0.0f;
    dsynth_config.eg_f_level = 1.0f;
    dsynth_config.eg_f_attack = 0.3f;
    dsynth_config.eg_f_decay = 0.01f;
    dsynth_config.eg_f_sustain = 1.0f;
    dsynth_config.eg_f_release = 0.2f;
    dsynth_config.eg_a_attack = 0.1f;
    dsynth_config.eg_a_decay = 0.01f;
    dsynth_config.eg_a_sustain = 1.0f;
    dsynth_config.eg_a_release = 0.7f;
    dsynth_config.lfo_waveform = DSynthSub::WAVE_TRI;
    dsynth_config.lfo_freq = 0.5f;
    dsynth_config.lfo_amp = 0.7f;
    dsynth_config.lfo_p_level = 0.0f;
    dsynth_config.lfo_f_level = 0.5f;
    dsynth_config.lfo_a_level = 0.0f;
    dsynth_config.portamento = 0.0f;
    dsynth_config.delay_delay = 0.3f;
    dsynth_config.delay_feedback = 0.3f;
    dsynth_config.overdrive_gain = 0.0f;
    dsynth_config.overdrive_drive = 0.0f;
    dsynthsub.Init(dsynth_config);

    // init
    note = 0;

    // start soundstream
    settings.setOutListener(this);
    soundStream.setup(settings);
}

void ofApp::audioOut(ofSoundBuffer &outBuffer)
{
    float sigL, sigR;

    for (size_t i = 0; i < outBuffer.getNumFrames(); i++) {
        dsynthsub.Process(&sigL, &sigR);

        outBuffer.getSample(i, 0) = sigL;
        outBuffer.getSample(i, 1) = sigR;
    }
}

void ofApp::update()
{
}

void ofApp::draw()
{
}

void ofApp::keyPressed(int key)
{
    if ((key > 30) && (key < 130))
    {
        note = key - 30;
        dsynthsub.MidiIn(MIDI_MESSAGE_NOTEON, note, 70);
```

```
    }
}

void ofApp::keyReleased(int key)
{
    dsynthsub.MidiIn(MIDI_MESSAGE_NOTEOFF, note, 0);
}

void ofApp::exit()
{
    ofSoundStreamClose();
}
```

Press the keys on your keyboard to make some sounds!

Reference

General

All DStudio instruments are configured using a struct. They are quite similar and the easiest way to understand them is by looking a the examples and study the header files.

config

init

.h files documented

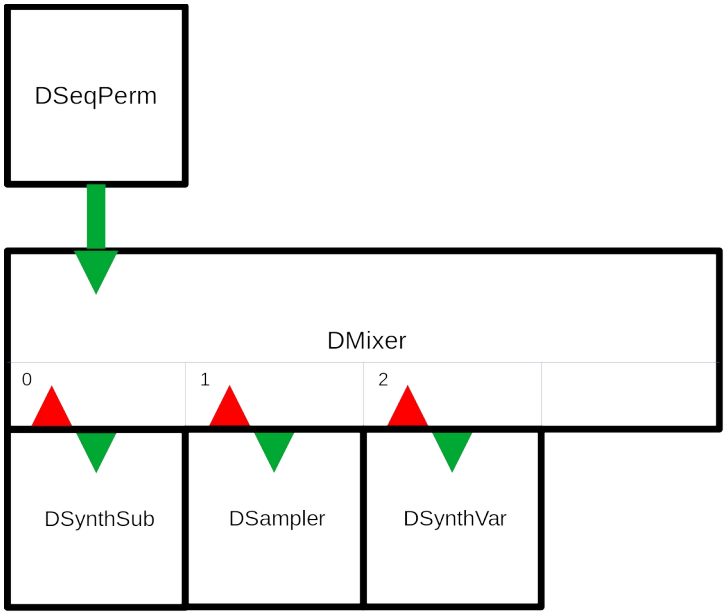
DMixer

This is the mixer. It handles incoming MIDI, distributes it to the right channel, and calls all sound generating objects and all channels, adds FX and produces a stereo signal.

It takes a number of arrays with settings, most notably the sound generators (synthesizers and drum sounds), apply optionally chorus and reverb, then pan and adjust the level for each sound/channel.

DMixer is also a DSound object. This means that you can use it as a channel in the mixer, chaining mixers and sound generators as much as your memory and CPU can handle!

MIDI pan and level messages are handled by DMixer, the rest is sent to the correct channel.



Built in effects: Reverb and chorus

Every channel can have a specific amount sent to a global reverb and/or chorus. The amount for every channel is provided by the reverb_level and chorus_level arrays.

The mixer config struct has three members that control how much FX is sent back to the main mix:

- float chorus_return - typical value 0.5
- float reverb_return - typical value 0.5
- float mix_dry - typical value 0.5

DSeqMidi

A MIDI sequencer that sequences tracks and has a song, ie sequences chaining, capability.

This is a sequencer that you feed with data in vectors.

You can use MIDI values or the shortcuts defined in dseq.h:

- **DTx**. Length of notes or sequences. DT1 = one whole bar, DT2 = a half bar, DT4 = a quarter bar/note etc.
- **DEx**. Sequence event. DEN is Note on.
- **DVx**. Velocity (volume). DV1 to 10 equals MIDI velocity from 10 to 100. DVMAX is max velocity, 127. DVOFF is "turn note off". A velocity of 0 is always Note off.

Song data (dmidisong_t)

A vector describing which sequences each channel should play on a song step. Each song step is a vector, eg

```
        {{0, DT1 * 8}, {5, DT1}, {5, DT1 * 8}, {3, DT1}, {5, DT1}},
```

This line describes one step in a song. Channel 0 should play sequence 0, for 8 bars, channel 1 should play sequence 5 for one bar etc. The duration of a song step is equal to the length of the longest sequence, in this case DT1 * 8 = 8 bars. Shorter sequences are repeated.

Sequence data (dmidiseq_t)

A vector describing the MIDI events in each sequence. Eg

```
        {
            {0, DEN, 31, DV7},
            {0, DEN, 38, DV7},
            {0, DEN, 43, DV7},
            {DT1*4, DEN, 38, DVOFF},
            {DT1*4, DEN, 39, DV7},
            {DT1*4, DEN, 36, DV7},
            ...
        },
```

At tick 0, start playing (DEN) note 31 (MIDI pitch number) with velocity 70 (DV7), note 38 with velocity 70 and note 43.

Four bars later (at position DT1*4) stop playing note 38, and start playing note 39 and 36. Note 31 will still be playing as it hasn't been turned off (DEN event with velocity DVOFF).

You can also send Cutoff and Resonance CCs.

Sequences with no notes should be defined as empty vectors:

```
        { // 2 bass2 verse
            {}
        }
```

Drum data

If you create a drum machine using a DMixer object of type MIXER_PERCUSSION, you can use the pitch value to control which drum sound will be played. Use the constants defined in dstudio.h. The DMixer will subtract MIDI_PERCUSSION_START from all pitch values it receives and then send the MIDI to the channel with the resulting value. Eg a note with pitch value DESNARE will be sent to channel 1 of the DMixer.

Threaded execution

The sequencer runs in its own thread:

```
void ofApp::setup()
{
...
    dseqmidi.Start();
    dseqmidi.startThread();
}
```

DSeqPerm

This is a MIDI sequencer that can permutate the music played, ie manipulate the note data in individual sequences and mute/unmute channels. It creates automatic remixes of your songs!

It also works differently than the DMidiSeq under the hood, creating a MIDI queue of events.

Sequence data (dmidiseqin_t)

Event (note) data doesn't use Note off events. Instead, you specify the length of the note and a Note off event will automatically be generated.

```
        { // 1 bass1 verse
            {DT1*0+DT16*0, DEN, 38, DV10, DT16*1},
            {DT1*0+DT16*3, DEN, 38, DV10, DT16*2},
        ...
        }
```

At tick 0, play note 38 for a duration of 1/16. At tick 3*1/16 play note 38 for a duration of 2*1/16.

For drum data you use the duration constant DTD.

Permutations (changes)

The strength parameter in the config-structure decides how forcefully the sequencer should change music. It is a float that gives the probability of a change occuring. Permutations are performed at each song step.

The permutations (changes) that can happen are:

- PERMUTATE_SWAP - swap pitch values with another note in the sequence
- PERMUTATE_RHYTHM - swap length with another note in the sequence
- PERMUTATE_SIMPLIFY - remove note from the sequence
- PERMUTATE_ADD - add a note by repeating an already existing note
- PERMUTATE_SHIFT - shift sequence in time
- PERMUTATE_TRANSPOSE - transpose sequence to other pitches used
- PERMUTATE_ORIGINAL - revert back to the original sequence

How large these changes will be and the number of notes affacted depends on the strength parameter.

In addition, the sequencer can also mute/unmute channels depending on the strength parameter.

DGen

This is an example of how to make generative music. The implemented class is called DGenDrone.

It creates never ending music based on simple MIDI note input and random changes.

Channels can be of different types that affect how notes for that channel is generated:

- DGenDrone::BASS: low droning notes
- DGenDrone::TREBLE: high droning notes
- DGenDrone::PAD: chords
- DGenDrone::MELODY: melody
- DGenDrone::ARPEGGIO: arpeggio
- DGenDrone::EMBELLISH: quick details

The DGenDrone class works on some note input:

- dgen_note_base: a vector with the base MIDI note of each channel
- dgen_note_pad: notes that are used for the channel type PAD. The values are relative to the base note for the channel.
- dgen_note_arp: notes that are used for the channel type ARPEGGIO. The values are relative to the base note for the channel.
- dgen_note_melody: notes that are used for the channel type MELODY. The values are relative to the base note for the channel

The algorithm uses states, called drama; INTRO, VERSE, CHORUS, BREAK, OUTRO. The DGenDrone::NoteCreate() method creates notes for a specific channel type and drama state. Notes are queued and sent when the time is right.

Transitions from one drama state to another is handles by a drama_order_ vector, indexed on drama state with three values that indicates which transitions are possible (with 60%, 30% and 10% probability respectively).

Every channel can have a drama_fade_ value. This indicates how long (in specified fraction of drama length) should be used for a fade in or fade out.

See example-10-generative-space-music. The example also shows how to work with presets.

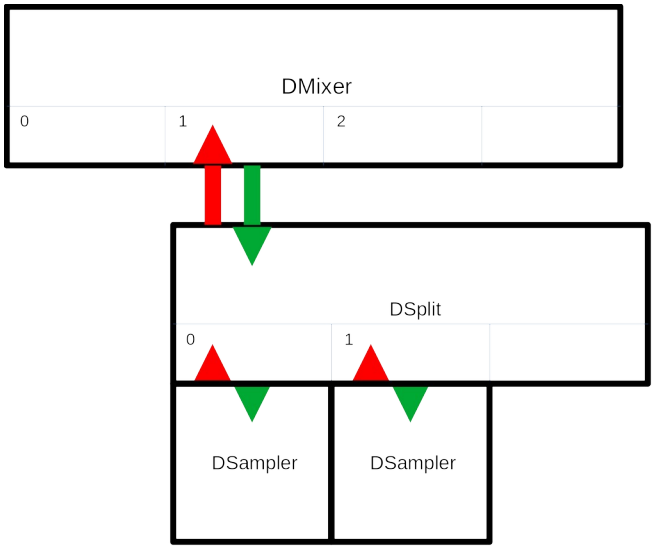
DSplit

A DSplit object is a kind of MIDI splitter. It works a bit like a mixer in that you attach several sound generating objects to different channels. You specify which MIDI pitch intervals should be sent to each channel.

In this example we have 4 DSampler objects. The DSplitInfo array tells the DSplit object to send MIDI notes up to 36 to channel 0 and add 36 to the MIDI pitch, send MIDI notes up to 48 to channel and add 24 etc. You can also use negative offsets.

```
    DSound *dsplit_synth[MIXER_CHANNELS_MAX];
    dsplit_synth[0] = &dsampler1;
    dsplit_synth[1] = &dsampler2;
    dsplit_synth[2] = &dsampler3;
    dsplit_synth[3] = &dsampler4;
    DSplitInfo dsplit_split[MIXER_CHANNELS_MAX];
    dsplit_split[0] = {36, 36, 0}; // end, offset, channel
    dsplit_split[1] = {48, 24, 1}; // end, offset, channel
    dsplit_split[2] = {60, 12, 2}; // end, offset, channel
    dsplit_split[3] = {72, 0, 3}; // end, offset, channel
    DSplit::Config dsplit_config;
    dsplit_config.sample_rate = settings.sampleRate;
    dsplit_config.channels = 4;
    dsplit_config.synth = dsplit_synth;
    dsplit_config.split = dsplit_split;
    dsplit.Init(dsplit_config);
```

Note that the DMixer input for DSplit must be stereo.



Midi input

MIDI input from a keyboard or DAW is possible with the ofxMidi addon that you can download from: https://github.com/danomatika/ofxMidi.

The ofApp must inherit (also) from ofxMidiListener, so in the beginning of ofApp.h you must have something like this:

```
#include "ofxMidi.h"
class ofApp : public ofBaseApp, public ofxMidiListener {
```

Add

```
    void newMidiMessage(ofxMidiMessage& eventArgs);
    ofxMidiIn midiIn;
```

to your ofApp class in ofApp.h.

Add the following calls to your ofApp::setup() method:

```
    // print input ports to console
    midiIn.listInPorts();
    // open port by number (you may need to change this)
    midiIn.openPort(1);
    // don't ignore sysex, timing, & active sense messages,
    // these are ignored by default
    midiIn.ignoreTypes(false, false, false);
    // add ofApp as a listener
    midiIn.addListener(this);
    // print received messages to the console
    midiIn.setVerbose(true);
```

And something like this for the newMidiMessage() method:

```
void ofApp::newMidiMessage(ofxMidiMessage& msg) {
    uint8_t midi_message = msg.status & MIDI_MESSAGE_MASK;
    uint8_t midi_channel = msg.channel;
    // ofxMidi: channel 1-16
    midi_channel--;
    if (midi_channel < dmixer.GetChannels())
    {
        switch (midi_message)
        {
        case MIDI_MESSAGE_NOTEON:
        case MIDI_MESSAGE_NOTEOFF:
            // mixer will send note on/off to synth on channel 0
            dmixer.MidiIn(midi_message + midi_channel, msg.pitch,
msg.velocity);
            break;
        case MIDI_MESSAGE_CC:
            // mixer willl handle pan and level
            // and send cutoff and res to synth on channel 0
            dmixer.MidiIn(midi_message + midi_channel, msg.control,
msg.value);
            break;
        default:
            break;
        }
    }
```

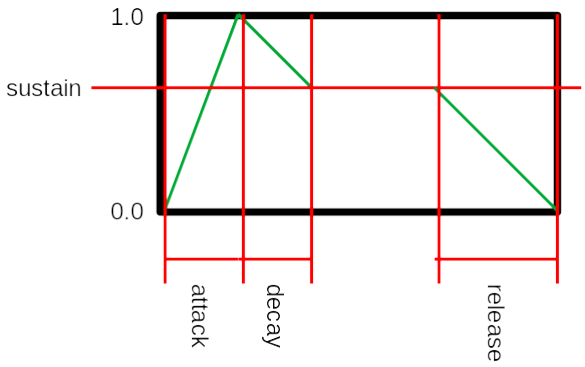
```
}
```

Now you can connect a MIDI keyboard or use the connection software of your OS to let a DAW control DStudio. Demo 5 is a simple example where a keyboard can control DStudio, while demo 6 is an example where a DAW (the GNU/Linux DAW MusE in my case) controls several sounds.

Instruments

The instruments/synthesizers have their own sound generation methods, but here are the features that are common to them all (with some exceptions to the DSynthVar which has more flexible types of modulation). They are all initialized using a config struct with the following common members:

- float sample_rate: set this to the global settings DSTUDIO_SAMPLE_RATE
- uint8_t voices: maximum number of polyphony
- float tune: detuning in hertz
- uint8_t transpose: MIDI transpose value
- float osc_level: level of oscillator (or oscillators)
- float noise_level: level of white noise (0.0 - 1.0)
- FilterType filter_type: the filter can be BAND, HIGH, LOW, NOTCH, PEAK or PASSTHROUGH (no filter)
- float filter_res: filter resonance (0.0 - 1.0)
- float filter_cutoff: filter cutoff frequency (should go no higher than half the sample frequency, ie DSTUDIO_FILTER_BASE (defined in dstudio.h)



- float eg_p_level: how much should the pitch envelope affect the pitch (0.0 - 1.0)
- float eg_p_attack: time (in seconds) to go from 0 to 1.0
- float eg_p_decay: time to go from 1.0 to sustain
- float eg_p_sustain: level of sustain ("holding down a key")
- float eg_p_release: time to go from sustain to 0
- float eg_f_level: how much should the filter envelope affect the cutoff frequency (0.0 - 1.0)
- float eg_f_attack: time (in seconds) to go from 0 to 1.0
- float eg_f_decay: time to go from 1.0 to sustain

- float eg_f_sustain: level of sustain ("holding down a key"); if this is 0.0 and the filter is LOW no sound will possibly be heard
- float eg_f_release: time to go from sustain to 0
- float eg_a_attack: time (in seconds) to go from silent to 1.0
- float eg_a_decay: time to go from 1.0 to sustain
- float eg_a_sustain: level of sustain ("holding down a key")
- float eg_a_release: time to go from sustain to 0
- Waveform lfo_waveform: waveform of the LFO: WAVE_SIN, WAVE_TRI, WAVE_SAW, WAVE_RAMP, WAVE_SQUARE, WAVE_POLYBLEP_TRI, WAVE_POLYBLEP_SAW, WAVE_POLYBLEP_SQUARE
- float lfo_freq: frequency of LFO in hertz
- float lfo_amp: amplitude of LFO (0.0 - 1.0+)
- float lfo_p_level: how much should the LFO affect the pitch (0.0 - 1.0+)
- float lfo_f_level: how much should the LFO affect the filter (0.0 - 1.0+)
- float lfo_a_level: how much should the LFO affect the amplitude (0.0 - 1.0+)
- float portamento: portamento time (in seconds to reach half the distance), works best when voices_ is 1
- float delay_delay: delay time (seconds)
- float delay_feedback: feedback level (0.0 - 1.0)
- float overdrive_gain: overdrive gain (0.0 - 1.0), it is usually good to reduce the gain when applying overdrive drive.
- float overdrive_drive: overdrive drive (0.0 - 1.0)

Presets

You can now save and load presets of all synthesizers and drums using the DSettings static class. They are stored as XML files.

You must use the addon ofxXmlSettings for this to work.

There is no error checking on loading presets.

The example-edit program lets you explore all settings for synthesizers and drums and save and load settings for use in any project.

See example-edit and example-10-generative-space-music.

DSynthSub

A classic virtual analog subtractive synth with two oscillators, noise, a selectable filter, an EG, a LFO that can control amplitude, filter or pitch, portamento, delay and overdrive FX.

Additional properties:

- Waveform waveform0; waveform of oscillator 0: WAVE_SIN, WAVE_TRI, WAVE_SAW, WAVE_RAMP, WAVE_SQUARE, WAVE_POLYBLEP_TRI, WAVE_POLYBLEP_SAW, WAVE_POLYBLEP_SQUARE
- Waveform waveform1; waveform of oscillator 1: WAVE_SIN, WAVE_TRI, WAVE_SAW, WAVE_RAMP, WAVE_SQUARE, WAVE_POLYBLEP_TRI, WAVE_POLYBLEP_SAW, WAVE_POLYBLEP_SQUARE
- float detune: a detune parameter which controls the interval between the two oscillators. Set it to 12.0 for a whole octave
- float osc0_level: level of oscillator 0 (0.0 - 1.0)
- float osc1_level: level of oscillator 1 (0.0 - 1.0)

DSynthFm

Like the DSynthSub but with FM at its heart.

Additional properties:

- float ratio: ratio between modulator and carrier signal
- float index: FM depth

DSynthVar

Variable wave shape synthesizer.

Additional properties:

- float waveshape: 0 is saw/ramp/tri, 1 is square
- float pulsewidth: pulsewidth when shape is square. Saw, ramp, tri otherwise
- bool sync_enable: whether or not to sync the oscillators
- float sync_freq: sync oscillator freq in Hz

Because this is mostly cool when you can modulate this, the DSynthVar has a more flexible modulation system.

Available modulators:

- 3 EGs (ADSR)
- 3 LFOs
- 3 SM (special modulators)

Additional properties that tell DSynthVar which modulators affect the different parameters. They can all be set to one of the constants defined in dsynthvar.h:

```
#define DSYNTHVAR_MOD_NONE 0
#define DSYNTHVAR_MOD_EG0 1
#define DSYNTHVAR_MOD_EG1 2
#define DSYNTHVAR_MOD_EG2 3
#define DSYNTHVAR_MOD_LFO0 4
#define DSYNTHVAR_MOD_LFO1 5
#define DSYNTHVAR_MOD_LFO2 6
#define DSYNTHVAR_MOD_SM0 7
#define DSYNTHVAR_MOD_SM1 8
#define DSYNTHVAR_MOD_SM2 9
```

Properties:

- uint8_t mod_eg_p: pitch envelope modulator
- uint8_t mod_eg_f: filter envelope modulator
- uint8_t mod_eg_a: amplitude envelope modulator
- uint8_t mod_filter_cutoff: filter cutoff frequency modulator
- uint8_t mod_waveshape: waveshape modulator
- uint8_t mod_pulsewidth: pulsewidth modulator
- uint8_t mod_sync_freq: sync oscillator frequency modulator. Works best when set to same modulator as pitch envelope.
- uint8_t mod_delay: delay modulator (this can produce some spacey effects)

The DSynthVar can use 3 Special Modulators (see dsm.h and dsm.cpp). How they use the proprties depend on the type.

SM: Noise (DSTUDIO_SM_TYPE_NOISE)

White noise signal (0.0 - 1.0).

Properties:

- float sm_0_freq: frequency of signal change.
- float sm_0_amp: amplitude of signal (0.0 - 1.0)
- float sm_0_offset: offset added to signal (0.0 - 1.0)
- uint8_t sm_0_seq_len: not used
- std::vector<float> sm_0_seq_val: not used

SM: Crawl (DSTUDIO_SM_TYPE_CRAWL)

Signal moves ("crawls") from a value to the next (0.0 - 1.0).

Properties:

- float sm_0_freq: frequency of signal change.
- float sm_0_amp: how much to change each step (0.0 - 1.0)
- float sm_0_offset: probability the signal will not change (0.0 - 1.0)
- uint8_t sm_0_seq_len: not used
- std::vector<float> sm_0_seq_val: not used

SM: Interval (DSTUDIO_SM_TYPE_INTERVAL)

Outputs signal or zero at random intervals (0.0 - 1.0).

Properties:

- float sm_0_freq: frequency of signal change.
- float sm_0_amp: amplitude of signal to output (0.0 - 1.0)
- float sm_0_offset: probability the signal will not change (0.0 - 1.0)
- uint8_t sm_0_seq_len: not used
- std::vector<float> sm_0_seq_val: not used

SM: Chaos (DSTUDIO_SM_TYPE_CHAOS)

Outputs random signal at random intervals (0.0 - 1.0).

Properties:

- float sm_0_freq: frequency of signal change.

- float sm_0_amp: max amplitude of signal to output (0.0 - 1.0)
- float sm_0_offset: probability the signal will not change (0.0 - 1.0)
- uint8_t sm_0_seq_len: not used
- std::vector<float> sm_0_seq_val: not used

SM: Sequencer (DSTUDIO_SM_TYPE_SEQ)

Outputs signal defined in sequence (0.0 - 1.0).

Properties:

- float sm_0_freq: frequency of signal change.
- float sm_0_amp: not used
- float sm_0_offset: not used
- uint8_t sm_0_seq_len: length of sequence
- std::vector<float> sm_0_seq_val: vector of values to output (0.0 - 1.0)

DC offset

This synthesizer can introduce a DC offset into the signal. It helps to insert a high pass filter before the DSynthVar. (See demo 3, 7 and 8.)

DSampler

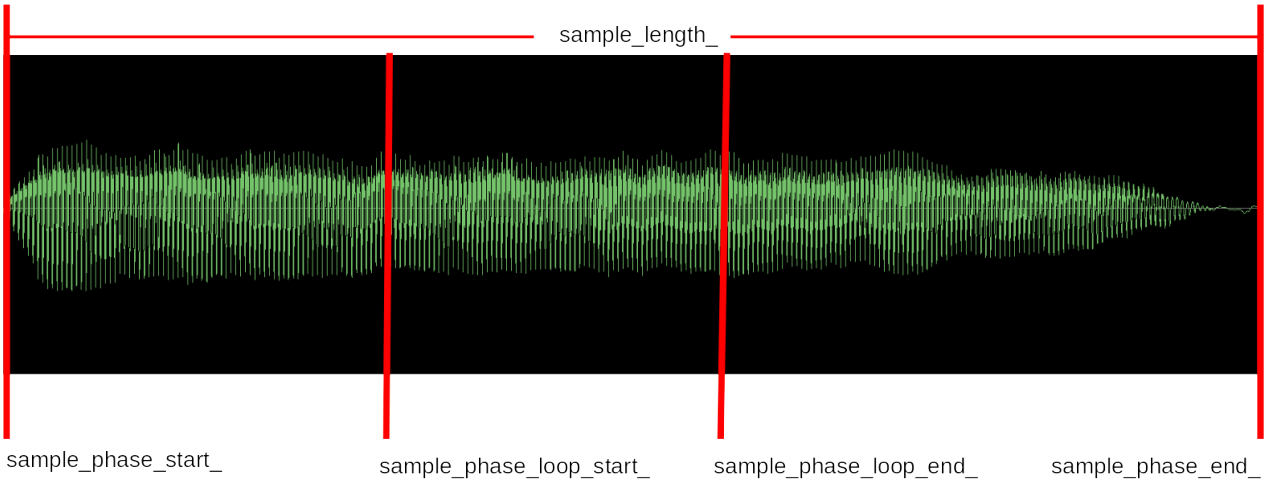
A sample player that uses linear interpolation for pitch changes. Tuned to A 440Hz but this can be set with the constant DSAMPLER_BASE_FREQ (dsampler.h).

The DSampler ned libsndfile (http://www.mega-nerd.com/libsndfile/api.html) and uses dynamic memory to load the sample.

It can handle both stereo and mono samples at 44100 hz.

Additional properties:

- bool loop: to loop (true) or not loop (false) the sample
- std::string sample_file_name;
- uint32_t sample_length_: length of sample
- uint32_t sample_phase_start_: start of sample (0)
- uint32_t sample_phase_loop_start_: start of loop (0)
- uint32_t sample_phase_loop_end_: end of loop (0)
- uint32_t sample_phase_end_: end of sample (length of sample)
- uint8_t sample_channels_: mono (1) or stereo (2)



The variables sample_phase_start, sample_phase_loop_start, sample_phase_loop_end, sample_phase_end and sample_length are set in DSampler::Load(). They are set to play and loop the whole sample. To avoid this, for example when loading a preset, pass false as the second parameter to Load() -- which is what you normally should do. If you want to base these parameters on the sample itself, pass true.

You load a sample using the Load() method:

```
dsampler.Load("data/test.wav");
```

You can use an absolute path or a relative path (for the oF examples relative to the bin directory).

Sample size is limited by the constant SAMPLE_BUFFER_MAX defined in dsampler.h

Drum instruments

All drum sounds have a lot of sound shaping properties. See the header files and demo examples.

Drum sounds respond to MIDI pitch values.

All drum instruments can also load presets (see Instruments > Presets).

DBass

Bass drum, with three different sound engines (type): analog (808, DTYPE_ANALOG), synthetic (909, DTYPE_SYNTHETIC) and opd (my home cooked engine, DTYPE_OPD).

DSnare

Snare. Like the bass drum it has three types.

DClap

Home cooked clap sound.

DCymbal

Use it for ride and crash sounds. See demo 1.

DDrum

A kind of tuned drum that can be used for things like toms. See demo 1.

DHihat

Open and closed hihat with three different sound engines like the bass and snare.

DFX

The DFX "plugins" can be used to enhace any DSound sound source. You can chain several plugins.

A DFX plugin passes MIDI straight through, process the child sound, adds fx and returns the modified signal. The child sound source is always called as a stereo source.

How to use a DFX plugin

Here is an example on how to add a DFXFlanger to a DSynthSub synthesizer (from demo 3).

Create a sound source (a DSynthSub):

```
dsynth_config.sample_rate = settings.sampleRate;
dsynth_config.voices = 6;
dsynth_config.waveform0 = DSynthSub::WAVE_SAW;
...
dsynthbass.Init(dsynth_config);
```

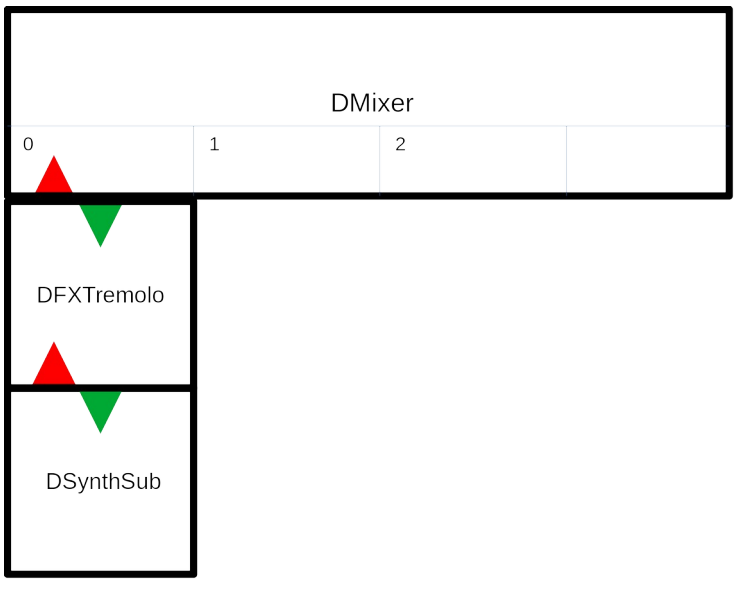
Create the DFXFlanger plugin and add the DSynthSub as a child:

```
// flanger on bass pad
DFXFlanger::Config dfxflanger_config;
dfxflanger_config.sample_rate = settings.sampleRate;
dfxflanger_config.level = 0.8f;
dfxflanger_config.feedback = 0.7f;
dfxflanger_config.lfo_depth = 0.8f;
dfxflanger_config.lfo_freq = 0.3f;
dfxflanger_config.delay = 0.8f;
dfxflanger_config.child = &dsynthbass;
dfxflanger.Init(dfxflanger_config);
```

Add it to the DMixer:

```
dmix_synth[0] = &dfxflanger;
```

Now when the DMixer's Process() method is called, it will call the DFXFlanger plugin which in turn will call the child, the DSynthSub plugin.



All DFX plugins can be found in the dfx.cpp and dfx.h files.

Common properties in config struct:

- float sample_rate: set this to the global settings DSTUDIO_SAMPLE_RATE
- float level: output level of plugin (0.0 -)
- DSound *child: connected child DSound sound source

Decimator

A decimator with bitcrush.

Additional properties in config struct:

- float downsample_factor: amount of downsample (0.0 - 1.0)
- float bitcrush_factor: amount of bitcrushing (0.0 - 1.0)
- uint8_t bits_to_crush: the number of bits to crush (0 - 16)

Delay

Stereo delay, can also be used for "ping pong" effects. Useful for drums that unlike the DSynth* don't have it built in.

Additional properties in config struct:

- float delay_delay_l; delay time left channel (seconds)
- float delay_feedback_l; feedback level left channel (0.0 - 1.0)
- float delay_delay_r; delay time right channel (seconds)
- float delay_feedback_r; feedback level right channel (0.0 - 1.0)

Filter

A filter with different modes of the same type as the filter in DSynth*.

- FilterType filter_type_: the filter can be BAND, HIGH, LOW, NOTCH, PEAK or PASSTHROUGH (no filter)
- float filter_res_: filter resonance (0.0 - 1.0)
- float filter_cutoff_: filter cutoff frequency (should go no higher than half the sample frequency, ie DSTUDIO_FILTER_BASE (defined in dstudio.h)

Flanger

A flanger where a delayed signal is modulated using an LFO. This signal is added back (feedback) to the dry signal which results in a kind of changing filter effect.

Additional properties in config struct:

- float feedback: flanger feedback level (0.0 - 1.0)

- float lfo_depth: LFO depth (0.0 - 1.0)
- float lfo_freq: LFO frequency (0.0 - 1.0)
- float delay: delay time (0.0 - 1.0, maps to .1 to 7 ms)

Overdrive

Useful for drums that unlike the DSynth* don't have it built in.

Additional properties in config struct:

- float gain: overdrive gain (0.0 - 1.0), it is usually good to reduce the gain when applying overdrive drive
- float drive: overdrive drive (0.0 - 1.0)

Panner

Panner is an autopanner with 3 modes (called types) and a built in LFO:

- type == DFXPanner::STATIC: offset is pan value (0.0 - 1.0)
- type == DFXPanner::LFO: offset is added to the LFO signal
- type == DFXPanner::RANDOM: A new random value is generated when the LFO has changed more than the offset value. Amplitude is the width of generated values.

Panner is a stereo effect that preferably operates on mono DSounds.

Used in example-4-sampler.

Slicer

The slicer samples a random length of the audio, and repeats it random times, then starts anew.

Used in example-4-sampler.

Tremolo

A tremolo that rhythmically changes the amplitude of the signal.

Additional properties in config struct:

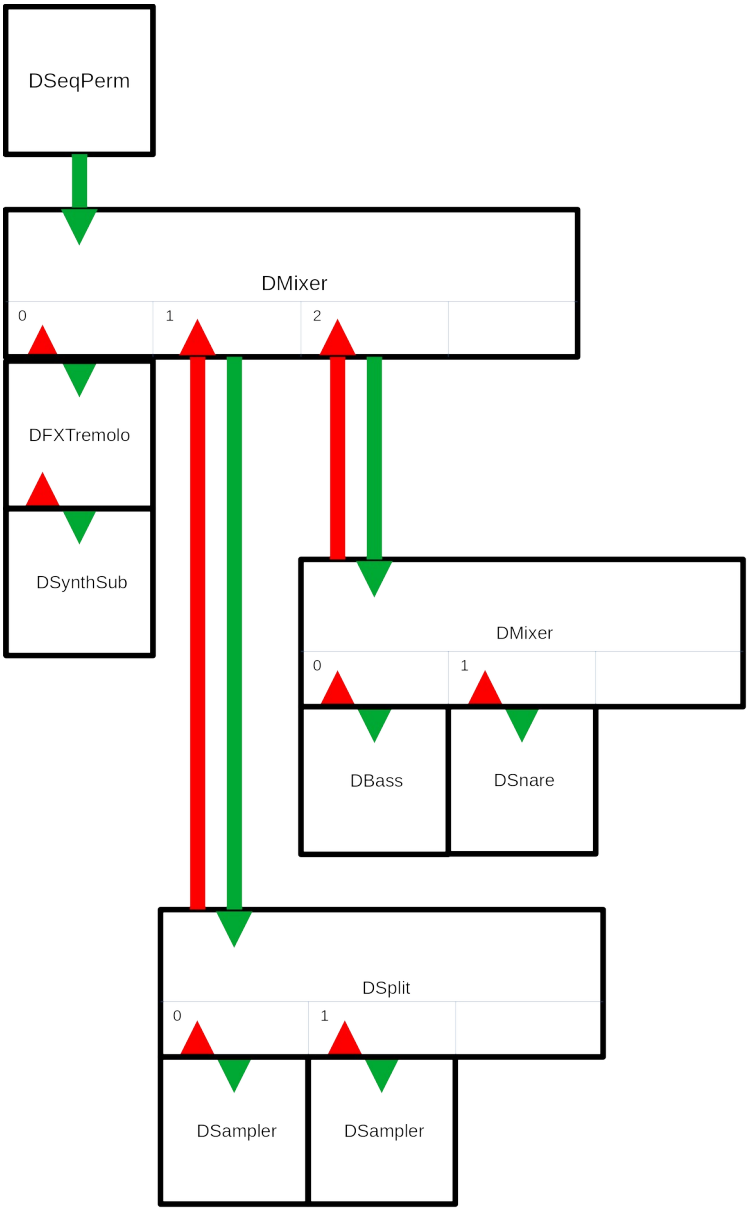
- float freq: frequency in hertz
- uint8_t waveform: waveform of amplitude change, can be WAVE_SIN, WAVE_TRI, WAVE_SAW, WAVE_RAMP, WAVE_SQUARE, WAVE_POLYBLEP_TRI, WAVE_POLYBLEP_SAW, WAVE_POLYBLEP_SQUARE,
- float depth: depth of tremolo (0.0 - 1.0)

Examples

A good way to learn to use DStudio is by running and studying the demo examples.

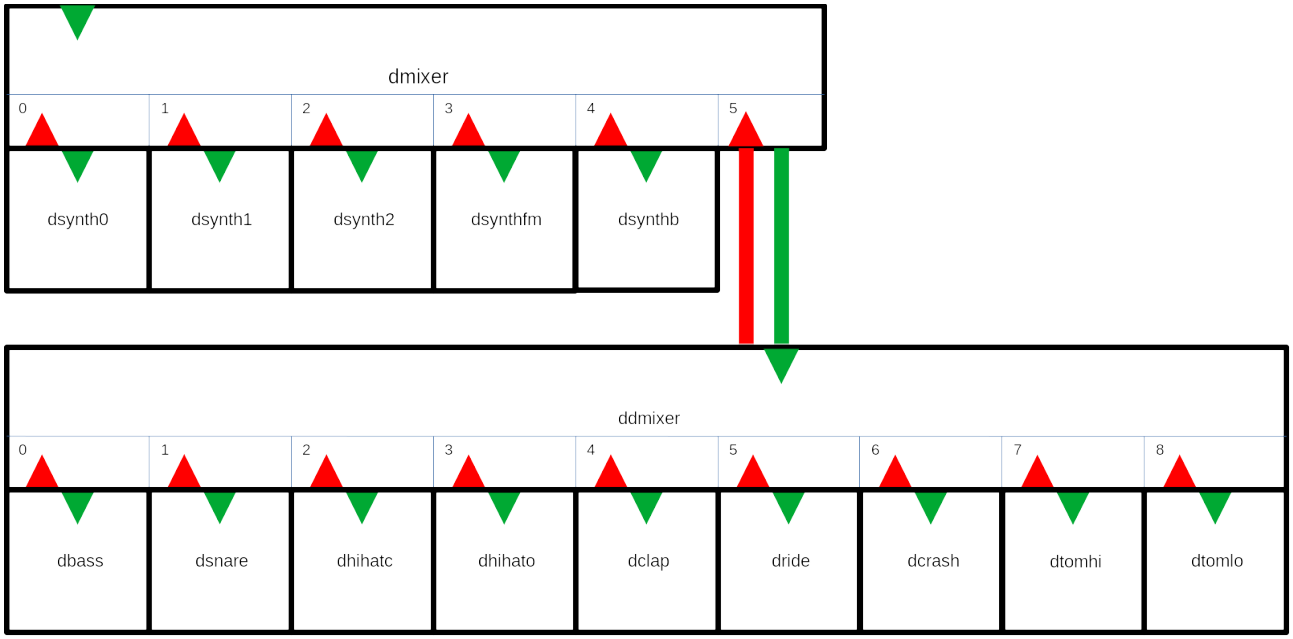
Components (example-0-components)

This is the end result of the showcase from the video.



Algorithmic (example-1-algo)

Demonstrates how to setup and use a bunch of different synthesizers and sounds, mix them together, and trigger them using the Metro timing class, feeding them various randomized data. A kind of algorithmic composition program.

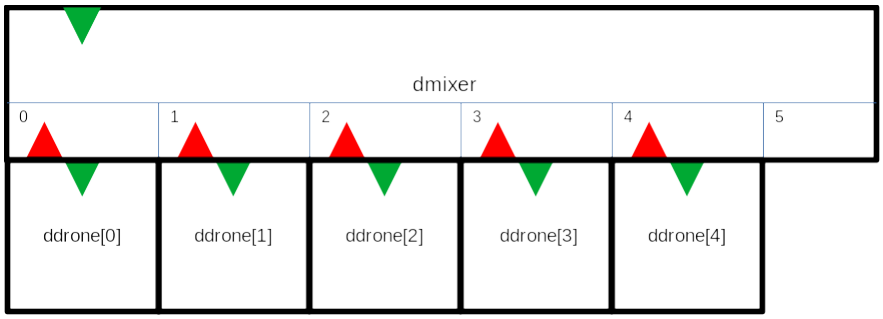


Drone (example-2-drone)

A drone experiment. In the ofApp.h you can configure the number of drones

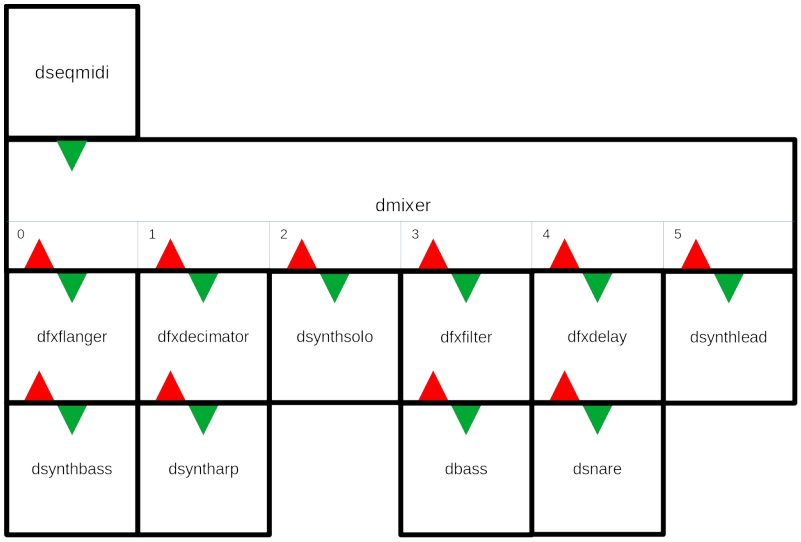
```
#define DRONES 5
```

Remember that the mixer handles 16 channels by default. If you create more drones you also have to increase the number of mixer channels in dstudio.h. I have successfully created 100 drones running at the same time.



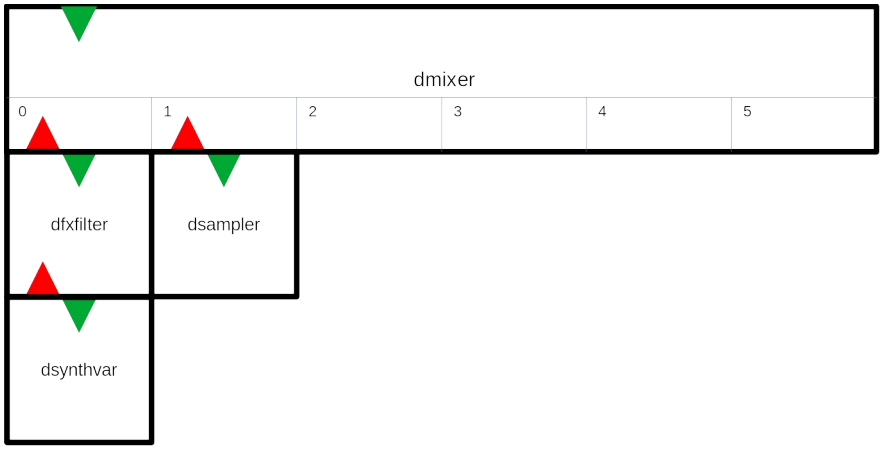
Sequencer (example-3-sequencer)

A sequenced song that shows how to create sequences and chain them into a song.



Sampler with FX (example-4-sampler)

An example of using the DSampler class and the DSynthVar class.



MIDI input (example-5-midi-input)

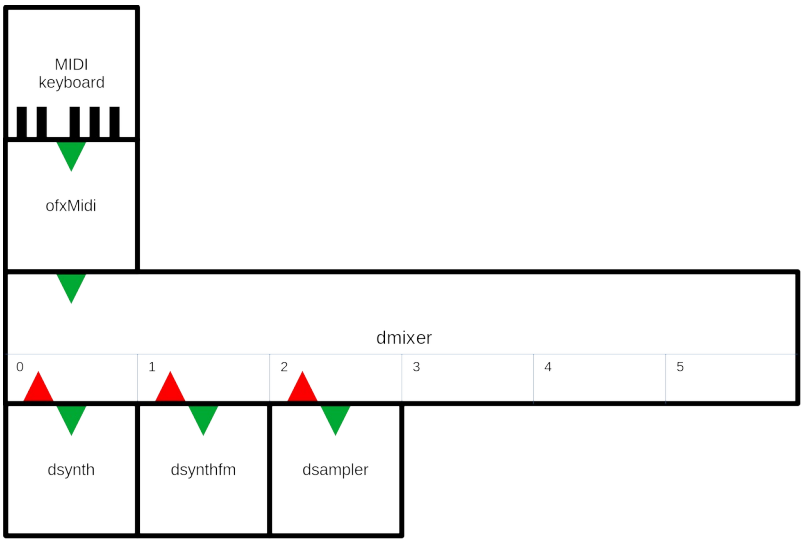
You have to download and install the MIDI addon: https://github.com/danomatika/ofxMidi.

This example is made to handle a connected keyboard. It handles both MIDI note on/off messages as well as pan and level (volume) messages.

You can change which of the three sound sources receives the MIDI by changing which sound source has index 0 in the DMixer config struct (ofApp.cpp):

```
dmix_synth[2] = &dsynth;
dmix_synth[1] = &dsynthfm;
dmix_synth[0] = &dsampler;
```

In this case the dsampler will play the incoming MIDI.

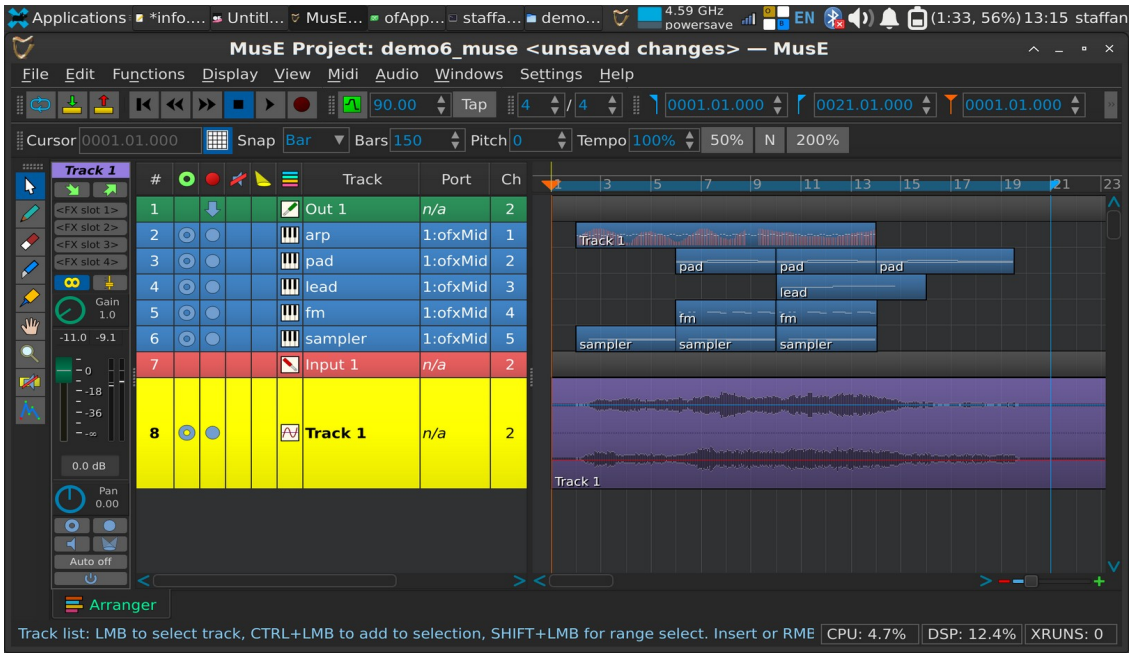


Midi input from DAW (example-6-midi-daw)

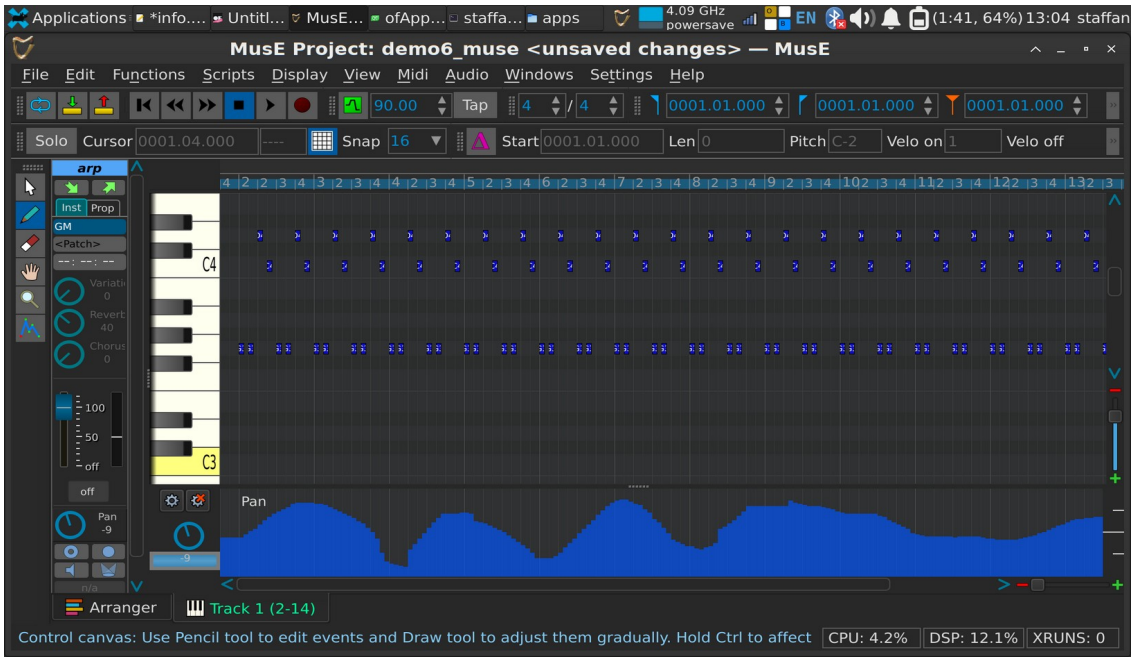
You have to download and install the MIDI addon: https://github.com/danomatika/ofxMidi.

An example on how to connect a DAW to DStudio. In my case I used the LIBRE DAW MusE (https://muse-sequencer.github.io/).

This is how the composition looks like in MusE (the audio track at the bottom is not part of the composition - I used MusE to record the output from DSound):

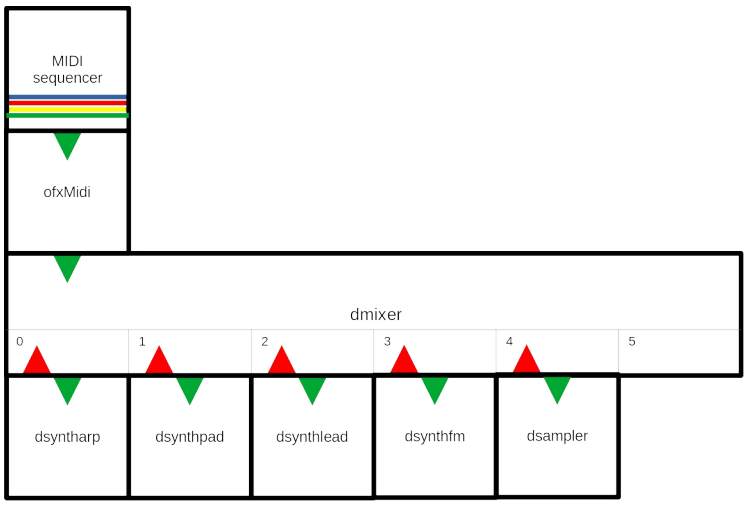


The arp track has an additional controller track that affects the pan of the channel:



I started MusE with only Alsa:

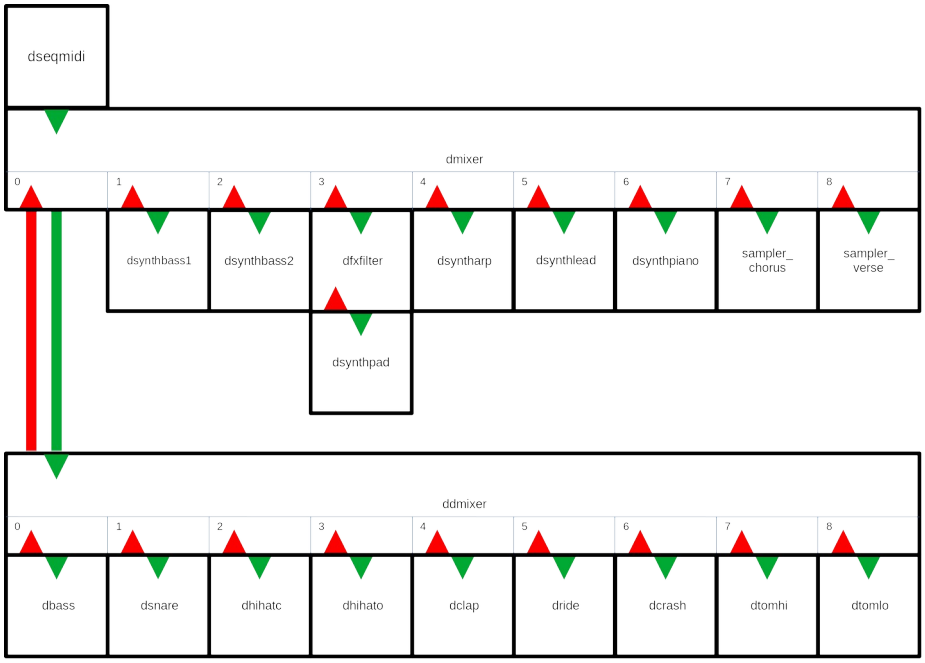
```
muse4 -A
```


MIDI
sequencer
ofxMidi
dmixer
0
1
2
3
4
5
dsynthарp
dsynthpad
dsynthlead
dsynthfm
dsampler

Synthpop (example-7-synthpop)

A sequenced synthpop song using DSeqMidi with DSampler providing vocals.

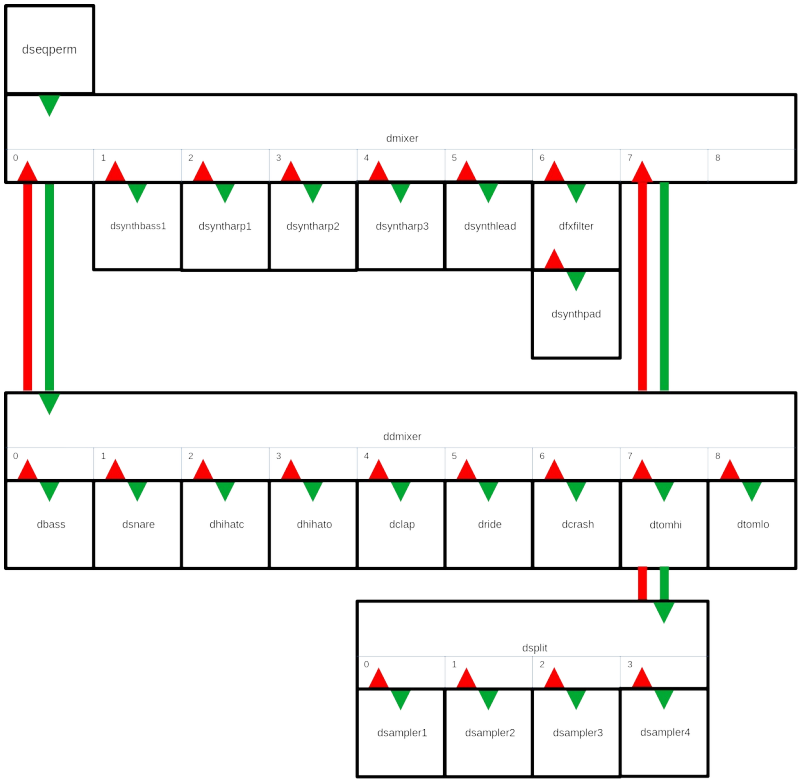
Demonstrates how to use the sequencer in a separate thread.



Permutating electropop (example-8-permutating-electropop)

An evolving electropop song using DSeqPerm and DSampler providing some electropop sounds.

Demonstrates how to use the sequencer in a separate thread.



MIDI visualizer (example-9-visualizer)

This is a sketch of how to integrate visuals based on MIDI.

Generative space music (example-10-generative-space-music)

This is an example of how to make generative music using a DGen class.

It creates never ending music based on simple MIDI note input and random changes.

Channels can be of different types that affect how notes for that channel is generated:

- DGenDrone::BASS: low droning notes
- DGenDrone::TREBLE: high droning notes
- DGenDrone::PAD: chords
- DGenDrone::MELODY: melody
- DGenDrone::ARPEGGIO: arpeggio
- DGenDrone::EMBELLISH: quick details

The DGenDrone class works on some note input:

- dgen_note_base: a vector with the base MIDI note of each channel
- dgen_note_pad: notes that are used for the channel type PAD. The values are relative to the base note for the channel.
- dgen_note_arp: notes that are used for the channel type ARPEGGIO. The values are relative to the base note for the channel.
- dgen_note_melody: notes that are used for the channel type MELODY. The values are relative to the base note for the channel

The algorithm uses states, called drama; INTRO, VERSE, CHORUS, BREAK, OUTRO. The DGenDrone::NoteCreate() method creates notes for a specific channel type and drama state. Notes are queued and sent when the time is right.

Transitions from one drama state to another is handles by a drama_order_ vector, indexed on drama state with three values that indicates which transitions are possible (with 60%, 30% and 10% probability respectively).

Every channel can have a drama_fade_ value. This indicates how long (in specified fraction of drama length) should be used for a fade in or fade out.

The example also shows how to work with presets.

Editor (example-edit)

This example creates GUIs for all synthesizers and drums and lets you edit parameters in real time.

You can play the selected sound with the keyboard (imagine a keyboard starting at C on key Z and running to the right one octave). You can also attach a MIDI keyboard so this transforms the example into a live synth with 4 sound engines and drums!

It uses the static class DSettings for saving and loading presets.

When saving and loading drum sounds you have to select which sound to save/load. Type the inital character of the sound, except use Y for cymbal.

DStudio with only rtAudio

It can be useful to be able to run DStudio with only rtAudio (used by openFrameworks), for example on a headless computer.

Everything works in the same way, except for the main audiocallback loop. Study the examples.

Install

You need (at least, only tested on GNU/Linux Debian 11)

- build-essential
- make
- libsndfile1

Install all files in a directory.

In a terminal, enter a demo directory and type make.

Run the example: ./rtDStudioExample

Reference: How I created the port to rtAudio only

Download all DStudio code.

Create a directory named dstudio_rtaudio

Copy ofxDaisysP to dstudio_rtaudio/rtDaisySP.

Copy ofDStudio to dstudio_rtaudio/rtDStudio.

Download rtAudio and move to dstudio_rtaudio/rtaudio.

Linking

Link some directories as some parts of DaisySP does some includes:

- cd rtDaisySP/src
- ln -s -r Synthesis Drums/Synthesis
- ln -s -r Utility Synthesis/Utility
- ln -s -r Filters Drums/Filters
- ln -s -r Control Effects/Control
- ln -s -r PhysicalModeling PhysicalModeling/PhysicalModeling
- ln -s -r Noise PhysicalModeling/Noise
- ln -s -r Dynamics PhysicalModeling/Dynamics

This is for GNU/Linux. Do something similar if running another OS.

Edit dstudio.h

Add:

```
#include <chrono>
uint64_t ofGetElapsedTimeMicros();
float ofRandom(float max);
```

Create new file dstudio.cpp

Add:

```
#include "dstudio.h"
#include <stdlib.h>     /* srand, rand */

#include <chrono>
uint64_t ofGetElapsedTimeMicros(){
    return
std::chrono::duration_cast<std::chrono::microseconds>(std::chrono::steady_cl
ock::now().time_since_epoch()).count();
}

float ofRandom(float max) {
    return (max * rand() / float(RAND_MAX)) * (1.0f -
std::numeric_limits<float>::epsilon());
}
```

Edit dseqmidi.h

Remove:

- #include "ofThread.h"
- void threadedFunction();

Change:

- class DSeqMidi : public ofThread -> class DSeqMidi

Edit dseqmidi.cpp

Remove:

- #include "ofUtils.h"
- void DSeqMidi::threadedFunction()

Edit dseqperm.h

Remove:

- #include "ofThread.h"
- void threadedFunction();

Change:

- class DSeqPerm : public ofThread -> class DSeqPerm

Edit dseqperm.cpp

Remove

- #include "ofUtils.h"
- #include "ofMath.h"
- void DSeqPerm::threadedFunction()

Add:

- #include <algorithm>

Edit dgen.h

Remove:

- #include "ofThread.h"
- void threadedFunction();

Change:

- class DSeqPerm : public ofThread -> class DSeqPerm

Edit dgen.cpp

remove

#include "ofUtils.h"

#include "ofMath.h"

void DGenDrone::threadedFunction()

add

#include <algorithm>

Remove dgfx.*

No use for visual fx without oF.

Prepeare examples

Move all examples to same level as rtDaisySP and rtDStudio.

Remove visual projects that use oF:

- example-0-components
- example-9-visualizer
- example-edit

Rewrite each example

Copy rtAudio/DStudio Makefile.

Remove:

- addons.make
- config.make
- *.qbs

Move bin/data/ to example folder.

Move src/* to example folder.

Rename ofApp.* to rtApp.*.

Remove main.cpp.

Copy rtAudio/DStudio main.*.

Edit rtApp.h:

- remove #include "ofMain.h"
- change DStudio includes to #include "../rtDStudio/src/*.h"
- replace DaisySP includes with #include "../rtDaisySP/src/daisysp.h"
- change class ofApp : public ofBaseApp{ to class rtApp {
- remove all methods except Setup() (capital S)
- add void Process(float *, float *);
- remove ofSoundStream soundStream;

Edit rtApp.cpp:

- Change includes:
  - #include "rtApp.h"
  - #include "../rtDStudio/src/*"

Remove soundstream settings and soundstream start

Replace all settings.sampleRate with DSTUDIO_SAMPLE_RATE

Change ofApp::Setup() to class rtApp {

Change void ofApp::audioOut(ofSoundBuffer &outBuffer) to void rtApp::Process(float *sigL, float *sigR)

In Process() process one sample and remove outBuffer so we only have dmixer.Process(sigL, sigR);

Remove other oF methods.

If using dgen/dseq* remove dgen.startThread();

In Process() add dgen.Process();

Replace use of ofXmlSettings

DXMLSettings in dsettings

In rtapp.cpp add #include "dsettings.h"

RtAudio reference

https://www.music.mcgill.ca/~gary/rtaudio/RtAudio_8h.html#a112c7b7e25a974977f6fc094cef1a31f

https://github.com/thestk/rtaudio

http://web.mit.edu/carrien/Public/speechlab/marc_code/ADAPT_VC/rtaudio/doc/html/index.html#playbackc

Compiler flags: http://www.music.mcgill.ca/~gary/rtaudio/compiling.html