vModSynth manual

Rafał Cieślak

January 27, 2013

Contents

Part I

Introduction

Many musicians who work using electronic synthesizers agree that modular synthesizers (often called *modulars*) are the superior instrument in their studio. By allowing it's user to design their own synthesis techniques, modular synthesizers are considered to be limitless, and therefore are popular up to this day, even though majority of synthesizers sold nowadays are not customizable at all, and instead provide the player with either a set of preprogrammed patches and timbres, or a fixed combination of synthesizing units, which forces the user to use only a very limited rage of timbres. A modular synthesizer is a collection of basic units, like an oscillator or a filter, which the user has to connect himself. By carefully designing connection schemes, tuning all module's settings, one is able to create any synthesizing path they want. Some say a modular synthesizer is a *synthesizer designing kit*, which clearly explains their advantage over all-in-one synthesizers.

vModSynth is a piece of software that simulates the behavior of a real, analog modular synthesizer. It's meant to provide it's user with the experience and fun of playing a real machine, and the look and feel of the synthesizers produced by synthesizers.com, one of world's leading modular synthesizers manufacturer. Therefore all modules currently available in vModSynth are very similar to the ones produced by synthesizers.com - not only from visual side, modules are designed to resemble the behavior of their .com equivalents.

vModSynth is by no means meant to be a powerful and feature-packed software synthesizer. It is also not meant to be easy to use by a newcomer. Using an analog modular synthesizer requires some knowledge in sound synthesis, and therefore such tools are not recommended for beginners. Also, by keeping as close to original as possible, vModSynth has many limitations - including monophony - that are not present in most other software synthesizers.

User that plays with vModSynth should feel the possibilities of a modular synthesizer. They should be able to realize any synthesizer design they may come up with, select modules they require, and freely connect them, routing audio and control signals as they wish. All modules equipped with knobs can have their settings tuned to one's wish, to produce desired effects, modifying the timbre in real time.

vModSynth uses MIDI for its signal input, and therefore is compatible with all MIDI controllers. Having connected a controller to vModSynth, you can play it and the synthesizer will react immediately. Sliders and knobs of a real physical device can be bound to knobs of modules within the virtual modular synthesizer, which means the awesome experience of touching a module's parameter is easily achievable. vModSynth can be also driven by a third-party software sequencer that outputs MIDI signal, like *Rosegarden*, *seq24* or *harmonySEQ*.

Part II
User Manual

1 Compiling and running

1.1 Supported platforms

vModSynth works on Linux exclusively. This is because it uses ALSA to communicate directly with the soundcard, it also makes a heavy use of MIDI routing features that are not present on other platforms.

vModSynth was extensively tested on Ubuntu (versions 12.04, 12.10 and 13.04) and UbuntuStudio 12.10, yet it should run on any Linux distribution, as long as it's dependencies are satisfied.

Because vModSynth produces the sound live, minimal latency is desired, and therefore best effects are achieved when run on a *realtime* or *lowlatency kernel*. It's not a requirement, though.

1.2 Dependencies

To run vModSynth, following libraries have to be installed within the host sytem:

gtkmm (version >= 3.5) vModSynth uses GTK to display and manage its graphical user interface, C++ bindings to GTK that are provided by *gtkmm* are preffered as vModSynth's code uses C++. Although it is not guaranteed the synthesizer will run on any older version of *gtkmm*, it may be possible with litte additional work to compile it with *gtkmm* 2.4, because no features new to 3.x are used.

libcairomm (version >= 1.10) To draw the graphical representation of synthesizer modules, vModSynth uses the C++ bindings to *Cairo Graphics* library, because its compatible with and recommended by GTK.

libasound2 (version >= 1.0.23) vModSynth uses *Advanced Linux Sound Architecture* to output audio, and react on sound driver's hardware interrupts, as well as to capture MIDI input, both note and control events.

To compile vModSynth, you will also need the development headers for these libraries.

Because vModsynth makes heavy use of C++11 features, a compatible compiler is required. GCC 4.7 is recommended.

For compiling, you may also want to use *pkg-config*, it is required by default build scripts to fetch data about other libraries (including paths to development headers). It is optional, though, and only required to run preprepared build scripts.

1.3 Compiling

vModSynth uses AutoTools to automate compilation. Therefore, the installation procedure is as simple as running:

```
./configure
```

and then:

```
make
```

You can also optionally install vModSynth, by running (as root):

```
make install
```

1.4 Running

At this stage vModSynth should be ready to run. Enter `./src/vmodsynth` (or `vmodsynth` if you have installed it) to start the application.

The executable accepts one optional command-line parameter, that can be used to explicitly specify the ALSA device vModSynth should use for PCM output. By default, that's *plughw:0,0*, but it may not work on some setups. Example usage: `./vmodsynth hw:0,0`

When starting vModSynth, it may emit a message: `Failed to open PCM device 'device'.` This means some other application is using this PCM device at the moment, and you need to stop it before vModSynth can be granted access to it.

2 Using vModSynth

Typical workflow with vModSynth is likely to look as follows:

1. (optional) Connecting an external MIDI device to vModSynth *(This is outside the scope of this manual).*
2. Adding an initial set of modules.
3. Creating connections between them, which defines how audio signal is routed.
4. Setting knobs and switches on modules on desired position, listening to the outcome.
5. Rearranging modules, removing, and adding new ones.
6. Setting knobs again, possibly binding them to physical sliders on an external device.

Last two steps are usually repeated until the player is happy with the resulting sound.

2.1 Main interface

The main vModSynth's window consists of a toolbar and a large area, where modules are displayed. As you start vModSynth, it should be empty until you add new modules to your setup.

2.1.1 Adding and removing modules

Press the plus (+) icon on the toolbar. You will be presented with a list of available modules, grouped into few categories. Select one of these modules, and double-click it. You will see it appear in the window's main area. Add one more module, it will be aligned to the previous one, just as if you mounted them in a classical wooden cabinet, on a single rack.

By clicking on a module's panel (the gray area that is the main body of a module, or simply any place on a module that is not an inlet/outlet nor a knob) you can select the module. It will have it's borders highlighted in blue, to mark which one is currently selected.

You can remove a selected module by pressing the minus (-) button on the toolbar. You can also rearrange modules order, by moving the selected module one position to left or right, using (<) and (>) buttons respectively.

2.1.2 Managing wires

All modules have some inlets and/or some outlets. Inlets is a socket that excepts an input signal, while outlet is a module's audio output socket. Unlike on a real modular synthesizer, vModSynth helps you distinguish between inlets and outlets - all inlets have a green tint, while outlets have a bit of red color in their hole.

To route audio between modules you need to specify how you want to connect them. There are some limitations to that. First, you cannot connect two outlets together. While this is possible on a real modular synthesizer, this usually makes no sense, and very often leads to hardware damage. Similarly, connecting two inlets together is not quite mindful, it would produce no effect, or - on some rare occasions - odd noises. Therefore connections between inlets to inlets and outlets to outlets are disabled in vModSynth by design.

It is also impossible to connect multiple outlets to a single inlet. This is also reasoned by how real synthesizers work, signals have to be mixed together by a mixer, a multi-connection like such has undefined effects. Connecting one outlet to multiple inlets, however, *does make sense*, but most modular synthesizer studios do not provide branching wires. The .com synthesizers use a “Multiply” module instead, which routes it's input to multiple outlets - and same solution is adapted in vModSynth.

To connect a inlet to an outlet with a wire, first click on an outlet (it will be highlighted in blue as selected), then on an inlet. The reverse order will not work. If there is already a wire connected to that inlet from another outlet, it will be removed. If there is already a wire connected from that outlet to another inlet, it will be removed too.

To disconnect a wire, first click the outlet it's connected to, then click on the inlet.

There is an unlimited number of wires you can use. All wires use a random blueish-greenish color to make it easier to identify them.

It is also possible - unlike in many related software (including *MAX/MSP* studio, *puredata*, *AMS*) to create looped connections. This allows creating feedback-based paths and using other uncommon techniques, just as with a real analog modular synthesizer.

2.1.3 Using knobs and switches

Almost every module has a knob or few. They serve different purposes and modify different parameters, but they share the look and interface.

To move a knob, click and hold on it, then drag the mouse. The knob will update immediately, and will preview you the value it's set to. Movement upwards increases its value, movement downwards lowers it. If you want to fine-tune a knob, move the mouse left-right instead. When you are happy with the value, release the mouse.

Selectors are similar to knobs, buy they have instead a fixed, small number of positions they can be set to. They are controlled exactly the same way as knobs.

To toggle switches, click on them. The result is also applied immediately.

2.1.3.1 Binding knobs to MIDI controllers:

Every single knob or selector can be quickly bound to a MIDI controller, which allows you to drive them by moving a slider or knob on an external MIDI device; external software emitting MIDI control events (like *harmonySEQ*) can be used too.

To bind a knob to MIDI controller, right click on it. A window will appear when you can choose desired channel (use 0 to listen on all channels) and MIDI controller number. Press “Bind” to confirm your choise.

To unbind a knob from a MIDI controller, right click on it and press “Unbind”.

2.1.4 Controlling the display

When you are working on a large set of modules, it is likely that it will not fit in window’s area. In such case, instead of using the slider in the lower part of the window, you can use mouse wheel to quickly browse your setup.

The “zoom-in” and “zoom-out” buttons on the toolbar can be used to scale the interface. This way you can get a closer look on a knob’s value, or get an overview of your path. The same effect can be achieved by scrolling mouse wheel while Control key is held down.

2.1.5 Edit mode

If you are happy with you setup and want to just play with it, you can disable the Edit Mode by toggling the“Edit” button on the toolbar, which should feature a pencil-like icon. This will disable all editing features, like adding/removing and rearranging modules and connecting/disconnecting wires, so that you won’t mess up things accidentally. Knobs and switches can still be moved. To re-enable Edit Mode, press the toolbar button again.

2.2 Modules

vModSynth comes with a number of different modules you can use for your synthesizers. While most of them are obvious to use for anyone with some knowledge on electronic music, it may be worth to explain how to use some of them.

2.2.1 Oscillator

An oscillator module is the heart of every synthesizer. It provides the base pitch for your timbres, or low frequency sine waves for modulation. vModSynth has a single universal Oscillator module which integrates LFO functions, just like the .com synthesizers.

It outputs 5 different wave shapes simultaneously (sine, triangle, saw, ramp and pulse). You can mix them together with a mixer module for a wide range of new shapes.

You can control the oscillation frequency with the selector and knob at the upper part of the panel. The selector chooses the range of frequency, while the knob finetunes it to any value (logarithmic scale is used). Then selector is in LOW mode, the oscillator functions as a LFO, in such case the frequency knob has the range from about 0.01Hz to 8Hz.

The oscillator module can also function as a VCO. Simply use it's frequency inlets to provide it with signal using 1V/octave standard. If any of these inlets is connected, the manual frequency setting is ignored.

Regardless of whether you control the frequency manually or using a signal, you can apply an exponential and/or linear modifier to it using dedicated inlets. Associated knobs specify how much influence over frequency the incoming signal has. This allows you to easily create a vibrato effect. The amount of exponential multiplier is specified in semitones - for vibrato effect you should set it under 0.2.

You are also free to modify the pulse wave's width using the down-most section. You can either set it manually, or have an external signal control it - the amount knob selects how much influence over this setting the incoming signal has.

The hard sync inlet is useful for synchronizing multiple oscillators, or for resetting it phase when a key is pressed. Any raise of signal on the hard sync inlet will reset the phase of the oscillator.

This oscillator output has voltage range of -5 to 5 V. This does not matter much in case of a software synthesizer, but you may be interested in this value for some uncommon uses.

2.2.2 Noise source

This module works as a source of white and pink noise. It can be then further filtered and/or used to control a variable parameter.

2.2.3 Audio output

This is a special module that routes incoming signal to your system's output. It's useful it you want to hear the resulting sound on your speakers or headphones. By default, you should use the mono output, but stereo output is also provided. If any of the stereo output inlets are connected, the mono signal is ignored.

There is a single knob on this module, it lets you easily scale the incoming signal to change it volume. At 100db the -5to5V signal is scaled to max volume of your soundcard (most likely that will be a -5to5V too).

2.2.4 MIDI input

This is a special module that reacts on MIDI note events. Please keep in mind that modular synthesizers are monophonic by design.

The pith outlets output a 1V/octave signal representing the pith of the most recently pressed key. There two outlets output identical signal, they are two just for convenience.

The gate outlet outputs a high signal if there is any key held down, and a low signal otherwise. It's most commonly connected to an envelope generator.

The velocity outlet outputs a 0to5V value representing the velocity of last MIDI note event.

Connecting MIDI device to vModSynth - which has to be done in order for this module to output any data - depends on your system and is outside of the scope of this manual.

2.2.5 Amplifier

This module attenuates two input signals by a constant value (set with “initial gain” knob) or by the value provided by the two control signals (the first of them can be scaled down with the “control #1 level” knob). The control signal is usually the MIDI velocity value, or the outcome of an envelope generator. Sometimes a LFO is used for a tremolo effect.

The selector can switch the function that is applied to the control input. It is most useful to select the sharpness of the envelope.

This module has two outlets, one being the inverted version of the other.

2.2.6 ADSR envelope

ADSR envelope generator is a module that generates the contour of a note. You can define the desired shape of generated envelope with four knobs. A gate-like signal is expected at the input, any raise grater than 0.5V triggers attack, and any decrease greater than 0.5V triggers release.

2.2.7 Filter

This is a universal filter module, that can apply low-pass, high-pass and band-pass filtering to a signal. It mixes the two input signals, for convenience. All filters are applied simultaneously, and outputted by three different outlets.

For all filters you can choose the corner frequency by moving the frequency knob (logarithmic scale is used). If the 1V/octave inlet is connected, the main frequency knob is used to scale the frequency 1/4 to 4 times.

You can also use the bandwidth knob, it applies only to the band-pass filter.

There are inlets for multiplying the bandwidth and frequency of the filter. Corresponding knobs specify how much does the incoming signal affect the parameter.

2.2.8 2-channel mixer

The two input channels are scaled according to knob values, and then mixed together. A constant offset can be added too, as set by the third knob.

This module has two outlets, one outputs the inverse of another.

2.2.9 Multiply

This module does not perform any transformation of the sound. It just routes a single signal to multiple outlets. There are three separate multiplying banks, but if you need more branches of a single signal, you can connect these banks together using the two switches between them.

2.2.10 Panorama/Crossfade

vModSynth does the panning and fading in a single module. The upper part of it is responsible for crossfading, you can set the value manually, or toggle the upper switch to have this value controlled by an external signal, plugged it into the middle inlet. The lower part does panning, and similarly - you can either set the panorama value manually, or toggle the lower switch and use an external source to control it.

Because the same external signal can control both panning and fading, this module is useful also for splitting a single signal into two parts, applying different effects on them, and joining them back together.

2.2.11 Sample and hold

This module remembers the value of input signal at a given moment and keeps constantly outputting it to it's outlet. The input level knob attenuates the incoming signal.

You can use your own gate to trigger sampling - just set the switch to "external" and plug a wire into the gate inlet. Any rise of this signal greater than 0.5V will sample the input signal.

Alternatively, using the sample and hold module in "internal" mode makes use of it's internal LFO to trigger sampling. You can choose it's frequency with the "sample rate" knob (logarithmic scale is used).

2.2.12 Echo

Echo module applies a simple echo effect to the passing signal. The two knobs let you specify delay time (50ms to 7s, logarithmic scale) and the amount of feedback.

2.2.13 Reverb

If your synthesizers sound too flat, you may want to add some smooth reverb to make them sound wider and deeper. This artificial reverbator produces a stereo output, which is great for 3d effects - but if you wish a mono reverb, use just the left output.

The first knob that selects the room size decides how deep the reverb is. Small values sound like a basement, larger as a concert hall, while the largest make an impression of a great cathedral.

The dampening knob specifies whether the walls of the room are solid (1) or smooth (0). The greater the value, the slower the reverb will decay. With dampening set to 1, there is no dampening at all, which means the reverb does not decay and stays infinitely long. While this can be used to produce some amazing effects, keep in mind this is not a stable setup, because the signal can reach any values, and will be soon clipped by your soundcard.

The dry/wet knob selects how much of the original (dry) signal should be outputted, and how much signal with effect applied (wet) should be added.

2.2.14 Overdrive

The overdrive module applies the effect of overdriving an analog amplifier. The gain knob scales the input signal, the level knob chooses the volume of resulting sound. The function switch can select the function of overdriving - it can be either a arctangent or a function clipping to -5to5V.

Part III

Technical Reference and Implementation Details

3 Technical Overview

vModSynth is an application that runs in real time. As with any other synthesizer, it is essential to compute the sound within minimal time period. This is because the latency should be as little as possible (so that software reaction on use input can be heard immediately), and because any pauses when samples are not delivered to the soundcard result in audible noises.

Another challenge is to maintain the extensibility of the application. Implementing new modules should require minimum work, once all core features are provided by main code parts.

To achieve this, all modules are implemented as separate classes. Most of them consist of just three methods - a constructor for initializing, a `dsp()` method which contains algorithms for calculating a single sample in sequence, and a `draw(...)` method, which defines how a module's interface should look like. With these features shared among all modules, managing them is much easier.

3.1 Thread management and synchronization

vModSynth runs two separate threads. One of them is responsible for the GUI, it passes control to GTK and reacts on its signals. The other one interacts with ALSA, and waits for hardware interrupts - which happen both when one of PCM's buffers is empty and new samples can be written (in such case DSP routines are called as many times as required to fill the buffer), or when a MIDI event arrives and has to be processed.

Synchronizing these threads is a bit of a challenge. This is because *GTKmm* is, quoting its documentation: "*not thread-safe, just thread-aware*". Moreover, *sigc++* library, on which *GTKmm* depends on is even thread-unsafe, and unwanted crashes are guaranteed when a signal is handled by a different thread than it originated from. Because of that, it gets tricky to ensure proper communication between GUI and audio threads.

Relying on mutexes to lock critical sections would not be enough in such case. While the audio thread is not sleeping, any GTK signal may not be emitted - as that might confuse *sigc++*. This is solved by *locking the global GDK lock*. Whenever audio calculations are taking place, GDK is locked completely, using `gdk_threads_enter()` and `gdk_threads_leave()` functions. This freezes GUI for the moment where calculations are taking place, but this is far too short period of time to be noticed by the user (and all UI events are scheduled to be processed once the audio routines are finished).

These two functions are marked as deprecated within GDK header files, but the documentation claims they are still working as expected, therefore related compile-time warnings can be safely ignored.

The only case when the audio thread has to trigger GTK events is when a MIDI control message is got and a knob needs to be redrawn, to update it's status. This is achieved with *GTKmm*'s *Dispatcher* object, which captures such signal, and emits it from the GTK thread, once the global GDK lock is disabled.

3.2 DSP routines

vModSynth does not calculate every sample at the time when it should be emitted. That would use much more CPU power, and would be easily affected by system lag, an audible noise might interfere with synthesized sound. Instead, vModSynth waits until *BUFFER_SIZE* (defined in *AlsaDriver.c*, default value: 128) samples are ready to be sent to soundcard and scheduled on its own buffer. With the default sample rate of 44100Hz, this means the audio thread wakes up on a hardware interrupt every $\frac{128}{44100} \times 1000ms \approx 2.9ms$. The internal soundcard's buffer is about the same size, and a 1-sample long delay for every connection between modules applies. This results in overall latency of about 6ms, which is absolutely fine for almost any musical usage.

When compiling vModSynth on a *realtime kernel*, one can safely change the buffer size to a smaller power of two (e.g. 32), and get even smaller latencies. I am unsure if that would be beneficial, though.

Once such hardware interrupt is received, the audio thread reacts immediately by calling `Engine::do_dsp_cycle()` *BUFFER_SIZE* many times. A single `do_dsp_cycle()` asks all modules to process 1 single sound sample, therefore calling it *BUFFER_SIZE* many times results in filling the output buffer (The buffer may stay empty if no Audio Output module is used, but this is desired, this way the application is always synchronized with PCM device).

`Engine::do_dsp_cycle()` is therefore the most important function of vModSynth. On the other hand, because of how the code is organized, it is also one of the simplest functions to be found in vModSynth source.

What it does is indeed simple. First it iterates over all wires that are present on current path, and asks them to pass a single sample of data from the outlet they are attached to to the inlet. This data can be then accessed by the module, using `Inlet::pull_sample()` (and similarly, a module can defer a sample to a wire of it's outlet using `Outlet::push_sample(sample)`). Next, it asks all present modules to perform their own DSP calculations for a single sample. This is done by simply calling module's `dsp()` function. For example, in case of Noise Source module, this call results in randomizing a new sample an pushing it to outlets. In case of Audio Output module, this results in pulling a sample from it's inlet, and scheduling it on vModSynth's output buffer. All modules define their own calculations for a single sample within the `dsp()` method.

All calculations within vModSynth are performed on variables whose type is *double*. This ensures enough resolution for maximum output quality.

3.3 User interface

The user interface is build using *GTK+/GTKmm* and *Cairo Graphics library*.

Tha main window's structure is very simple - it's just a *GtkToolbar*, a *GtkTreeView* for displaying module list (when adding one), and a large *GtkDrawingArea*. The window itself is managed as a separate class (*MainWindow*) while the main DrawingArea's features are implemented within *Cabinet* class.

The *MainWindow* does little except for setting up the UI and reacting on signals. It defers most calls, such as adding new modules or rearranging them, to functions within *Engine* namespace.

The *Cabinet* class, which inherits from *GtkDrawingArea*, has more custom logic added. It implements handlers for several events. The *draw* signal is handled by a custom drawing method. It has simple implementation: it iterates through all modules, and uses their own `draw()` function to let them add themselves to current Cairo drawing context. Afterwards, it does the same for inlets/outlets, knobs, switches and, lastly, wires. Note that all these objects implement a `draw()` function, where they define how to draw such element on a Cairo context. It also reacts on mouse button presses, it determines what was clicked, and processed mouse drags, if it was a knob. The *Cabinet* class also extends *DrawingArea*'s functionality to scrolling detection, so that it can be moved within parent container when the user scrolls the mouse wheel.

Another UI piece is the *ControllerWindow* class. It represents the dialog window for binding knobs to MIDI controllers.

3.4 Modules

All modules are implemented as separate classes. They all inherit from common *Module* class. This class implements common properties, a bunch of helper functions, and some virtual functions which are meant to be overdriven by its descendants.

Properties of a module include:

`std::vectors` of Inlets Outlets, Knobs and Switches These store poiners to all GUI elements that this Module makes use of. This makes it easy to access them from within module's methods.

`add_inlet(...)`, `add_outlet(...)`, ... functions and similar, including `add_knob(...)` , `add_selector(...)` and `add_switch(...)`. These are helper functions that create a new instance of an Inlet/Outlet/... class, register it in the engine (so that the core system knows that they should be drawn etc.), set its parameters, and add it to corresponding vector.

`draw_*` functions are meant to provide a certain drawing functionality, e.g. `draw_background(...)` draws the gray panel over a Cairo Context passed as an argument, which stands for the main body of the module.

`highlight` boolean variable Determines whether this module is selected and should appear highlighted or not.

`xoffset` Number of pixels this module has to be drown away from the left border. This value is responsible for proper alignment of a module, as it is applied to all its drawing functions.

`is_point_within` is used to tell whether a point of given coordinates belongs to the internal area of this module. This is used by the *Cabinet* to determine which module was clicked.

3.4.1 New module template

A new custom module should implement a constructor, a `dsp()` function, and a `draw(...)` algorithm.

The constructor would normally initialize internal module values, and add UI elements to the module - for example, using `add_outlet(10,25)` creates a new outlet which will be drawn at coordinates 10,25 within this module. N-th outlet can be then accessed using `*outlets[N]`, and this it is usually convenient for the DSP routine to output resulting sample with `outlets[N]->push_sample(q);`. Similarly, a module can use inlets, switches and knobs, so that they are easily accessed by the DSP function (e.g. `knobs[5]->get_value();`).

The `dsp()` function is expected to pull one sample from all inlets, process them, generate new output samples, and push one at a time to an outlet. Of course, no expensive unneeded computations should take place within this function, as it will be called very frequently, and small changes can affect CPU usage in a significant manner.

The `draw(cr)` function is called with a pointer to a Cairo Context, and it is expected to draw module's content (except for inlets, outlets and other UI elements) onto the context, using methods provided by Cairo.

3.5 UI elements

There is a number of classes that represent a common UI element, and implement it's functionality so that all modules can make use of it.

All UI elements have a `draw()` function, which implements drawing this particular element onto a Cairo context. Some also implement a `redraw()` function, which schedules redrawing of the Cabinet, trimming the refreshed area to the area of the UI element (trimming saves time needed to redraw), so that it gets properly redrawn with a new value.

3.5.1 Knob

It represents a value from a given range, with big accuracy. It can be either controlled by dragging - in such case it is driven by the *Cabinet*, or by a MIDI controller. It has a sophisticated drawing algorithm, to resemble knobs of .com synthesizers. It's `get_value()` returns the current value it's set to, and therefore is commonly used within DSP routines. A knob also remembers the parameters of MIDI controller it's bound to, so that when a MIDI control event is received, simple iteration over all knobs can find the one that should be moved.

3.5.1.1 Selector It works almost like a knob, yet it has a fixed number of positions. Therefore it inherit most functions from a *Knob*, but always casts it's internal value to an integer number, and has a different drawing routine.

3.5.2 Switches

Similar in use to *Knobs* and *Selectors*, this UI element is used when a binary toggle is needed. `get_value()` returns a boolean variable.

3.5.3 Inlet/Outlet

These classes implement a basic functionality of an *Inlet/Outlet*. Such element stores the pointer to the wire that is plugged into it (NULL if there is no connection). They can also *pull/push* a sample, which represents fetching it from / passing it to the connected wire.

3.5.4 Wires

These are created when a connection is defined. They always remember their associated *Inlet* and *Outlet*. A *Wire* also remembers the samples that are passing through it, and can pass the data from corresponding inlet to outlet with `pass_sample()`, which is crucial for synchronizing wires with modules (see ***DSP routines*** for more information).

4 Code Units

The source code is divided into a number of files.

4.1 Main files

4.1.1 main.cpp

This file contains the *main()* function of the program. It initializes GTK, starts the audio thread, processes command-line arguments and constructs a new instance of a *MainWindow* to be displayed as the application launches. Global variable `quit_threads` is used to notify the audio thread that it should now exit gracefully, after the application is closed.

4.1.2 AlsaDriver{.cpp,.h}

These files implement all audio output and MIDI input features. `thread_main()` is where the audio thread starts execution. It initializes both PCM and ALSA SEQ interfaces to interact with ALSA. The thread sleeps looped waiting for hardware interrupts, and recognizes them. If this interrupt was caused by an incoming MIDI note, `midi_input()` is called. If it was caused by an empty ALSA buffer being ready to be refilled, `playback()` gets run.

Global variables `last_note_pitch` and `last_note_velocity` represent a table of channels, and are used to remember what was the last note event that appeared on that particular channel (0 represents all channels).

`playback()` runs DSP routines to produce required number of samples. It is expected that while this function runs, `sound_buffer` will be filled in, which is done by e.g. the Audio Output module - it calls `add_sample(l,r)` which stores given stereo samples onto the currently calculated position in the `sound_buffer`.

`get_last_note_*` are convenient functions (used by MIDI input module) which return the data of the most recently received MIDI note.

4.1.3 Engine{.cpp,.h}

All functions within these files are encapsulated in *Engine* namespace, for clarity. There is a number of functions there, and they serve different purposes, yet they are are a part of application's core.

The `do_dsp_cycle()` function is used to synchronously compute one sample in by all modules.

`[un]register_*()` functions are used to add or remove a UI element to the global list of all present knobs/inlets/etc. This way the *Cabinet* knows what to draw.

`[un]select_*()` functions are used to mark items as selected (highlight them). Also, it recognizes the sequence of connections, and `[un]connects()` inlets and outlets (which creates/destroys a new instance of *Wire*) together. There is also an `unselect()` function, which is called to reset selection.

`move_selected_module_*()` are used to rearrange modules. These functions swap adjacent modules by changing their *xoffset* property.

`create_and_append_module(ID) remove_module(pointer) and remove_selected_module()` are responsible for creating new instances of desired modules, and for gracefully removing them.

There are also `zoom_in/zoom_out()` functions that are used to scale the GUI. *Cabinet* respects the GUI scale when performing it's drawings and click detections.

4.1.4 MainWindow{.cpp,.h}

These files contain the implementation of *MainWindow* class, which represents the main GUI piece (see: ***User Interface***).

4.1.5 Cabinet{.cpp,.h}

Cabinet class, which inherits after *GtkDrawingArea*, is the main element of the UI interface, and is used to fill the are of the *MainWindow*. It has custom drawing algorithms that are responsible for displaying the modules, wires, etc. (see: ***User Interface***)

4.1.6 ControllerWindow{.cpp,.h}

ControllerWindow class is just a *GtkDialog* with some custom widgets. It is used to choose the MIDI controller/channel and is displayed after right-clicking a knob.

4.1.7 Module{.cpp,.h}

This is where the main *Module* class is implemented. All particular modules inherit from this class. Both `draw(...)` and `dsp()` are pure virtual functions, and therefore no instances of *Module* itself can be constructed. More details about *Module* class can be found in ***Modules*** section.

4.1.8 ModuleList{.cpp,.h}

The purpose of this file is to define the `ModuleList` *enum*, and implement `create_new_module_from_ID` which constructs a new module and returns a pointer to it. This way *ModuleList.cpp* is the only file that *#includes* files from the `./modules` directory, and all the rest of code is fine using the *Module* prototype class.

4.1.9 Outlet{.cpp,.h}, Inlet{.cpp,.h}, Knob{.cpp,.h}, Selector{.cpp,.h}, Switch{.cpp,.h}, Wire{.cpp,.h}

These files implement the UI elements, as described in the ***UI elements*** section. Please note that a *Selector* inherits from a *Knob*, since in fact they do share most logic.

4.2 Module files

These reside in ./modules directory. Except for *algorithms{.cpp,.h}* (which contains DSP algorithms that are shared by many modules), all these files implement a single class, which represents a certain module. Only a few of them require some explanations:

v1001: Audio Output It uses `add_sample(l,r)` from *AlsaDriver* to store samples on vModSynth's output buffer.

v1005: MIDI input It uses `get_last_note_pitch(channel), get_last_note_velocity(channel)` from *AlsaDriver* to get information about most recent MIDI note event.

v101: Noise source The pink noise is the same white noise, yet filtered out. Filter parameters were originally designed by Paul Kellet.

v230: ADSR envelope generator The `phase` value represents the stage of envelope that is currently being outputted. 0 - no envelope, 1 - attack, 2 - decay, 3 - sustain, 4 - release.

v300: Filter This module features a one-pole low-pass filter, a one-pole one-zero high-pass filer, and a two-pole band-pass filter. The filtering algorithms were designed with help of *The Theory and Technique of Electronic Music* by Miller Puckette.

v703: Reverb This reverb design consists of three sequential quick-echoes, and four-channel looped reverbator body.

5 Tests

Testing vModSynth requires precision as it is not just correctness that is tested. A software synthesizer needs to use as little computing power as possible, and should never hog the host system unnecessarily. The reaction on input should be as quick as possible, and there should be no audible quirks or noises.

vModSynth was extensively tested on three different machines with different hardware specifications. Operating systems involved include *Ubuntu* 12.04, 12.10 and 13.10 development builds, as well as *UbuntuStudio* 12.10 with a *low-latency* kernel. The tests were taking place during development, to identify bugs related to new features as soon as possible, and a major testing was done once everything was mostly ready to be released, to ensure stability of the overall product.

GUI tests included using the UI elements in unusual manner, interfering the UI with third-party applications, steering the knobs and modules to reach corner values. Repeated addition and removal of random modules in huge quantities, maintaining huge number of connections, proper reordering of great number of different modules. Binding a big number of knobs to controllers, steering them all simultaneously (including MIDI controllers manual mouse movement).

All modules have undergone heavy correctness tests, which were performed by recording the resulting soundwave (which was captured by a virtual soundcard), and manual analysis of outputted samples. Most modules were also tested for corner cases, if appropriate (e.g. peak frequency gain for band-pass filter, constant signal as an input to a reverbator). The only minor issues that were found are caused by sampling resolution, which can be excused because of digital sound synthesis nature, and are by no means audible to humans.

Endurance tests (setups running constantly for several hours, or repeated addition and removal of connections/modules, or long-term parameter change) were also performed, and no problems were found.

During most of these tests, CPU usage of all threads was measured with third-party tools. Depending on hardware specs used, vModSynth can eat up a big deal of CPU when a very complex path is designed, but it never reaches critical values. Comparing to other established synthesizer software, vModSynth is not outstanding in terms of performance, but it does not use significantly more resources.

MIDI tests involved playing vModSynth with a external MIDI keyboard, connecting a software sequencer (*Rosegarden*, *harmonySEQ*) to vModSynth, and also manually applying corner cases, by sending fake, incorrect MIDI messages to vModSynth.

Along the explained above tests, I have spend a significant amount of time testing vModSynth in a *real usage case* - as a fan of modular synthesizers it was a great pleasure to me to spend a long time with it, designing synthesizers and playing them on my external keyboard. This was also useful for finding issues which might have been found only by trying to be productive while using this piece of software (including inconsistent UI, unexpected or illogical module behavior).