VisualTracker – modular pd environment for sequencing events on timeline

Aleš ČERNÝ
Prague, Czech Republic
ales.cerny@gmail.com

Abstract

VisualTracker is linear sequencer built in *pd* allowing users easily integrate their own time based *abstractions* and execute them in compositions graphically visualized on a timeline. Once this *abstraction* (called *module*) is dynamically loaded into environment a shared *connector abstraction* automatically links the user program with essential features of the *VisualTracker environment* and creates its visual representation on *gui*[1] *timeline*[2] to control its timing. Shifting these *module* representations in separate window and grouping them into *tracks* provides a very intuitive and user friendly way to create compositions of various program events and also control their output routing. Fully integrated system for saving *module* data and compositions is implemented.

Keywords

sequencer, gui, timeline, output routing, saving system

1 Introduction

The purpose of this paper is introduction of the *VisualTracker* concept and its approach to potential users and/or *module* developers. The few next chapters explainthe main program parts with the focus on understanding details important for *module* development and their integration into larger program structures. Knowledge of *pd documentation*[3] is assumed in this text.

2 VisualTracker environment *(VTe)*

VTe is global program part responsible for essential features shared by user *modules* loaded as discrete *abstractions*.

2.1 Program location

The main program code of *VTe* is placed and logically sorted in **[pd program]** subpatch. Although this part of *VTe* is meant not to be accessed by regular "production" users it is recommended to explore it to understand the processes and logic of the program. Note any changes in these parts will be lost when upgrading to the next versions of *VisualTracker.*

2.2 User Interface

VTe is built as standard *pd abstraction* with basic *Graph On Parent* providing access to main user parts of the program. Each **[bng]** opens a corresponding *subpatch* window with detailed *gui* or program code providing functionalities of *VTe* and maintaining the *modules*.

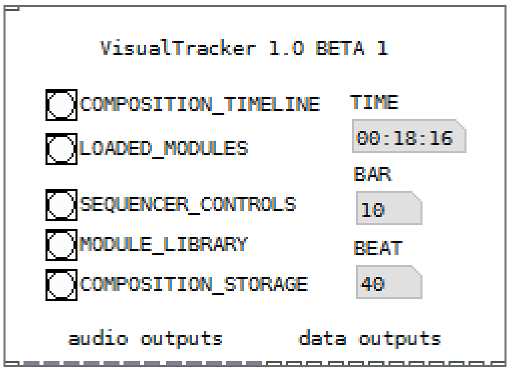


fig. 1 VisualTracker abstraction

2.2.1 Control windows

Sequencer_controls + *Composition_storage* are just simple gui windows with controls triggering other *VTe* program *subpatches* and functions described later. Their content is hopefully self- explanatory.

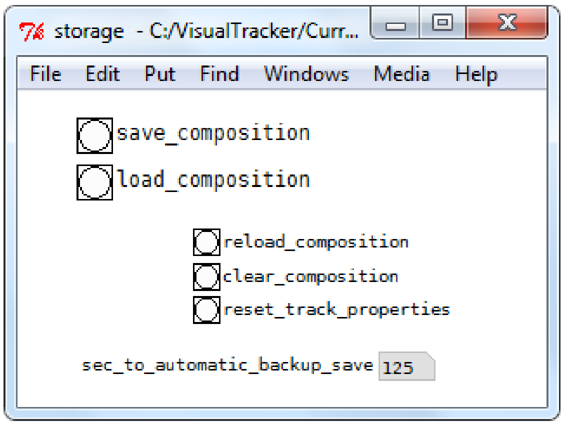
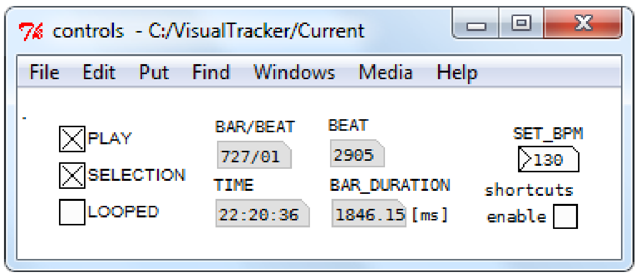


fig. 2 Composition storage window

fig. 3 Sequencer controls window



2.2.2 Module_library window

This window contains *abstraction* **[vt_module_launcher]** located in main *abstraction folder* providing easy one-click loading of listed *modules*. As the *module launcher* is kept outside the main patch it can be easily refilled by desired *modules* or even distributed within separate *module packs* (see 6.2 Module pack)

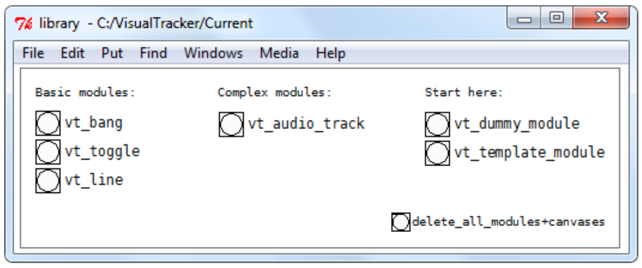


fig. 4 Module library window

2.2.3 Composition_timeline window

Composition_timeline window is the main *VTe* user interface where *module canvases* are placed and composed into the compositions (see 2.4.1 Module canvas position and manipulation). It contains bar grid constructed from **[cnv]** *object*, time scale, simple loop range controls and 36 instances of [**track]** *abstraction* responsible for routing of data and signal of particular *modules*. The window is divided into three sections - track controls & settings (left side), time scale & selection control (upper side) and time grid in the rest of the window (see 2.4 Timeline)

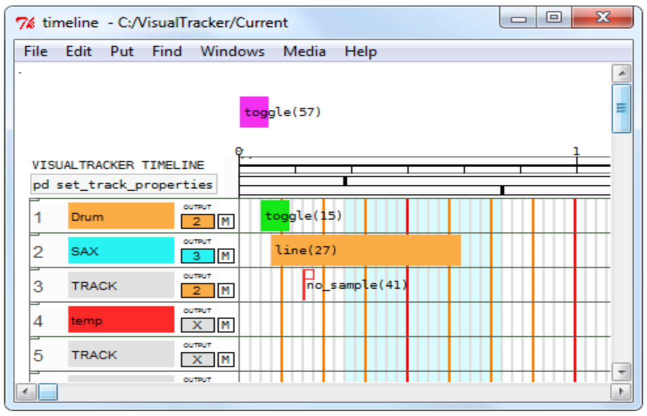


fig. 5 Composition timeline window

2.2.4 Loaded_Modules window

Loaded_Modules window is a container for the functional Module *abstractions* currently integrated (loaded) into *VTe* (see 3 Modules) Each *module* can vary in its size, gui and may contain other sub windows according to its functionality. *Modules* are created in this window in three main ways:

•by simple manual *object box* creation and typing *module abstraction* name (for example **[vt_line]**). *Abstraction* must be located in *VisualTrackler* folder or global *pd* path

•by **[vt_Module_launcher.pd]** *abstraction* triggered from *Module_library* containing predefined Modules (see 6.2 Module pack)

•by *Storage system* (see 2.5 Composition storage)

Modules created from *Module_library* or by *Storage system* are automatically aligned according to their predefined size. Manually created *modules* are aligned after hitting *Reload_composition* in *Composition_storage* window (see 2.5.7 Reload composition).

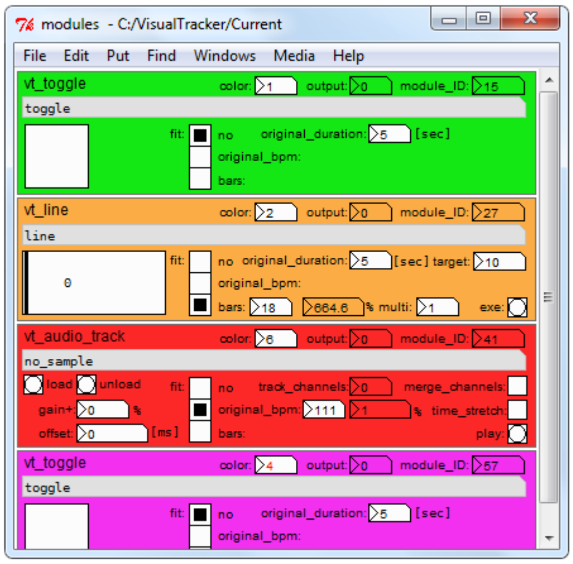

modules - C:/VisualTracker/Current
File Edit Put Find Windows Media Help
vt_toggle color: 1 output: 0 module_ID: 15
toggle
fit: no original_duration: 5 [sec]
original_bpm:
bars:
vt_line color: 2 output: 0 module_ID: 27
line
fit: no original_duration: 5 [sec] target: 10
original_bpm:
bars: 18 664.6 % multi: 1 exe:
vt_audio_track color: 6 output: 0 module_ID: 41
no_sample
load unload fit: no track_channels: 0 merge_channels:
gain+ 0 % original_bpm: 111 1 % time_stretch:
offset: 0 [ms] bars: play:
vt_toggle color: 4 output: 0 module_ID: 57
toggle
fit: no original_duration: 5 [sec]
original_bpm:

fig. 6 Loaded modules window

2.3 Sequencer

Sequencer is obviously core of *VTe*. It is based on **[line]** object which is sequenced using **[div]** and **[change]** objects.

2.3.1 Sequence division

Due to user interface optimization main time sequence unit of *VTe* sequencer and therefore minimal time resolution of particular *module* event is one *bar*[3]. Further division to smaller time parts such as *beats* or even their divisions (used in samplers or piano rolls) should be implemented directly in *Module program* using time related *objects* such as **[line]** or **[delay]** (actually creating *sub-sequencers*). Module events may also be synchronized by global *VTe receives* (see 3.2.1 lobal receives). Default *sub-sequencer* interface is in development (see 4 Future development)

2.3.2 Tempo

Speed of the sequencer is defined by *BPM*[4] controlled in *Sequencer_controls* window.

2.3.3 Selection

The selection feature is controlled by *toggles* in *Sequencer_controls* window and visualized in *Composition_Timeline* window limits the range of main sequencing **[line]** and forces the sequencer to not start from 0.

2.3.4 Sequencer looping

Looping within global time of sequencer is time based - provided by **[delay]** depending on BPM and controlled by two *vsliders* setting the looping points in *Composition_timeline* window

2.3.5 Sequencer Controls

Except the main sequence output sent into modules, *VTe* sequencer also provides control values displayed in *Sequencer_controls* window such as beat counter or time position. Sequencer can be controlled by default keyboard shortcuts (**space** for play/stop, **L** for looping and **S** for selection) after checking the toggle *shortcuts enable*. Shortcuts can be changed in **[pd key_shortcuts]** subpatch (see 2.1.1 Program location)

2.4 Timeline

Timeline is the main *VisualTracker gui* where the *module canvases* (see 3.1.1 Module canvas) are placed and handled.

2.4.1 Module canvas position and manipulation

Module canvases can be moved freely by mouse or keyboard in *pd edit mode* and they automatically snap to the *bars* and *tracks* so the composition always stays neat. The section above the time scale can be used as a swap place for currently unused, though loaded, *modules* respective to their representations. The position of *module canvas* on the timeline determines the time when the *module* program is triggered (horizontal position) and to which output of *VTe* is its output data/audio sent (vertical position).

2.4.2 Composition time (horizontal grid division)

Horizontal grid division is currently based on musical segmentation for 4/4 time signature. It means the basic horizontal unit (gray line distance) is one *bar.* Further visual grouping to *phrases*[5] is indicated by a red line. As the distance between bars is graphically fixed to 8 pixels, the variable dependent on the *BPM* of sequencer is *bar* duration **[vt_bartime].** Actual composition time is visually indicated by time scale above the grid section and speed of the timeline cursor. Note the *bar* is fundamental and minimal unit used by *VTe* (see 3.1.2 Single execution points).

2.4.3 Tracks (vertical grid division)

Thegrid is vertically divided into *tracks*. Properties of each track (horizontal strip) is controlled by *track abstractions* on the left side of the grid. *Track abstraction* can hold its color, name and primarily sets the output routing for *modules* placed into it. 0 means the output is sent nowhere (unconnected outlet)

It is important to understand the visual concept of Module canvases composed into tracks is strictly virtual. No *module* data or audio is actually sent through *track abstractions* - the relation of *Module canvas* position and track output selection leads just into setting routing switch in *connector abstraction* (details of solution using **[route]** and **[mux]** *objects* in each **[track]** *abstraction* in combination with **[demux~]** object in *connector abstraction* is beyond focus of this text).

Track abstractions are individualized by *numboxes* at the far right of the grid. They can be replicated (CTRL+D) and their number is (theoretically) infinite. Number of separate outputs is in current release (1.x) limited to 12. Output of each *track* is set equally for data and audio (you can not send data and audio signals of one *module* to different outputs. If *module* is using stereo output, the first channel is routed to the *track* defined in *track abstraction* and second one to *track* number + 1.

2.5 Composition storage

Obviously no complex software would make sense without the possibility of saving the state of ongoing work and various compositions across workstations or even software versions.

Due to lack of native storage solution in pd, *VisualTracker* is using its own system allowing storage of user data.

2.5.1 Saving main *VisualTracker* abstraction

Although it is necessary to modify core abstraction during the work (*module abstractions* are created in *Loaded_modules* window, *module canvases* are shifted in *Composition_timeline* window) all user data are stored in external storage and *patch* itself should stay unmodified. If the user needs to modify core code it is recommended to hit *clear_composition* **[bng]** in *Composition_sorage* window before saving and check if *Loaded_modules* window is completely empty and *Composition_timeline* window contains no *module canvases*. After regular "production" use of VisualTracker (composing the modules) and saving user data through storage system when asked to save patch ("Do you want to save the changes you made in ….. ?") just click "NO"

2.5.2 Storage system

VTe storage system is based on **[coll]** object[6] using the advantage of its indexed txt rows. Composition is saved by hitting *save_composition* **[bng]** in *Composition_storage* window. These selected data are fed into **[coll]** and then dumped into a text file. Theactual storage program is located in **[pd data_storage]** subpatch. The storage file contains selected global values (BPM, track names, track colors and track outputs) and user defined values of each currently loaded *module*. The save/load process is delivered through the *storage abstraction* **[storage]** respective **[mstorage]** connected to desired value and equipped by correct *abstraction creation arguments* (see tables). There are two approaches to saving data in VTe different for *VTe* and *modules*:

2.5.3 Global values storage

Global value is any data type (number or symbol) used outside the *module abstraction*. Typically it is *BPM* of *VTe* or track name or track color. But it can be also value used completely outside *VisualTracker* in any *patch* opened in current instance of *pd*. The only requirements are to access **[storage]** *abstraction* located in **/abs** folder and user assurance of unique identifier within *parameter syntax* (due to reserved identifiers it is recommended to use identifiers 400 - 500 when saving values outside *VTe*). Values are stored in **[coll]** *object* respective text file in tagged rows and loaded back using routed output of **[storage]** *abstraction*.

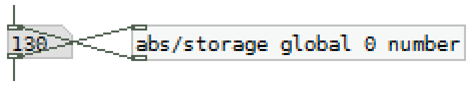
Note the **[storage]** *abstraction* can´t be simply used inside another *abstraction* due its requirement for unique identification. This can be resolved by *abstraction individualization* using *numbox* inlet or variable *abstraction creation aegument* ($1)

2.5.3.1 [storge] abstraction arguments

1.[integer > 0] identifier - unique global number

2.[integer or symbol] actual stored value or symbol (according to switch argument 3)

3.[keyword "number", "symbol" or "end"]data type switch



```
130   abs/storage global 0 number
```

fig. 7 - storage abstraction

2.5.4 Module values storage

Although using similar concept the approach to saving *module values* within *VisualTracker storage system* is completely different. Due to requirement to not modify the main VisualTracker patch (see 2.5.1Saving main VisualTracker abstraction) *modules* are not saved itself, but there is only creation data saved into external file including the file name of particular *module abstraction*. Then during load process all *modules* are re-created from scratch using **[obj(** *message* and stored *module values* are loaded through creation arguments and transferred to certain *numboxes* using $ variables. This solution leads to two very interesting user features:

•*module* can be simply "replicated" (CTRL+D) including its all on the fly changed user values while maintaining its independent *abstraction status* (see 2.5.7 Reload composition)

•**[mstorage]** abstraction can be used for defining the initial values of newly created modules. Desired creation value is hard set as part of **[mstorage]** creation arguments (see syntax table)

2.5.4.1 [mstorge] abstraction arguments

1.[$0] module ID variable

2.[integer > 0] identifier - unique global number

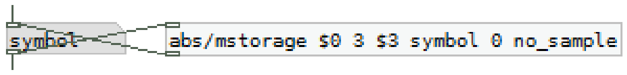
3.[integer or symbol] actual stored value or symbol (according to switch argument 4)

4.[keyword "number", "symbol" or "end"]

data type switch

5.[integer > 0 or 0] initial value if switch is set to "number"- if switch is set to "symbol" this argument must be 0

6.[symbol] initial symbol - relevant only when switch is set to "symbol"

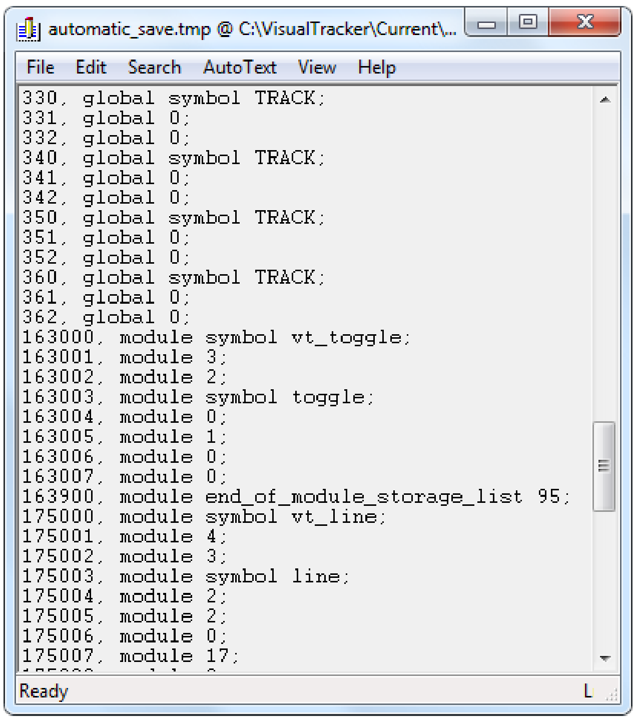


```
symbol
abs/mstorage $0 3 $3 symbol 0 no_sample
```

fig. 8 - mstorage abstraction

2.5.5 External storage file

Each global and module value is stored in a separate row of external text file. Rows are indexed according to value type and *module* affiliation and. *module* values are moreover divided into sections. Each section starts with the name of particular *module* and is complemented by *end identifier* following by module canvas height (see tagged obligatory parts in **vt_template_module.pd**)



automatic_save.tmp @ C:\VisualTracker\Current\...
File Edit Search AutoText View Help
Ready
L

```
330, global symbol TRACK;
331, global 0;
332, global 0;
340, global symbol TRACK;
341, global 0;
342, global 0;
350, global symbol TRACK;
351, global 0;
352, global 0;
360, global symbol TRACK;
361, global 0;
362, global 0;
163000, module symbol vt_toggle;
163001, module 3;
163002, module 2;
163003, module symbol toggle;
163004, module 0;
163005, module 1;
163006, module 0;
163007, module 0;
163900, module end_of_module_storage_list 95;
175000, module symbol vt_line;
175001, module 4;
175002, module 3;
175003, module symbol line;
175004, module 2;
175005, module 2;
175006, module 0;
175007, module 17;
```

fig. 9 - External storage file content

2.5.6 Auto save

Auto save functionality provides automatic saving of running work to **automatic_save.tmp** file located in **/storage** folder. Saving is executed every 300 seconds and indicated in *Composition_storage* window. Interval can be changed in **[pd data_storage]** subpatch.

2.5.7 Reload composition

After hitting corresponding **[bng]** in *Composition_storage* window whole composition is saved to **/storage/reload.tmp** file and immediately loaded back. This operation leads to two main results in *Loaded_modules* window:

- user values connected to **[mstroage]** abstraction are transferred into *creation arguments* of loaded module abstractions and module can be simply replicated (CTRL+D). This is useful when working with module containing a lot of user options (such as sequencers or samplers) and new module instance should use a similar setup.
- manually created modules abstraction are automatically aligned.

3 Modules

Modules are functional parts of *VisualTracker* actually performing program actions triggered by *VTe* sequencer. They are built as discrete abstractions and can contain pd code playing sound samples or video files, triggering midi notes or operating external hardware according to the flexibility of whole pd environment. Note the purpose of *VisualTracker* is not to provide a large variety of terminal *module functions* but rather to develop an open environment prepared for imagination of *module developers*. The aim is to provide an easy way to integrate any program into *VTe* - all you need is to paste your program into *module template* (see 3.4 Module template) and use some predefined sends, receives and other features providing communication with *VTe* and delivering essential features described in this paper. *Modules* are handled (but not stored) in *Loaded_modules* window.

3.1 Visualtracker connector

[visualtracker_connector] *abstraction* located in **/abs** folder is one of obligatory parts of each Module. This *abstraction* is shared by all the *modules* and opening a pd *patch* containing this *abstraction* causes actual “loading” of the *module* into *VTe* (of course if main *VisualTracker* patch is running). Instances of *connector abstraction* are individualized by their *creation arguments* respective they assuming unique *local variable* ($0) of each *module abstraction*.

3.1.1 Module canvas

Module canvas created in *Composition_timeline* window upon each *module loading process* is the most essential feature of *connector* and *VisualTracker* in general. *Module canvas* is a visual representation of *module event* on timeline defining its *execution time*. It is connected to *module program* through its *name reference* and also visualizes some *module properties* such as duration, name or color. *Module canvases* can be freely moved across timeline by mouse (in *pd edit mode*) or can be moved by the *module programs* themselves generating dynamic compositions.

3.1.2 Single execution points

As described above the position of *module canvas* on a timeline defines the time point when the *module program* is triggered. Practically it means once the sequencer positions, counted in bars, equals the position of *module canvas* on timeline corresponding *module* receives the bang **[r $0-EXECUTE]** which is further processed in *module program.*

3.1.3 Multiplication

Multiplication allows continuous repetition of a program event within the one *module canvas* (*module canvas* resizes dynamically according to number of multiplications).

It is important to understand the difference between *multiplication* and *looping*. While *looping* is the repetition of program code inside the *module program* (such as **[phasor]** based sample player) *multiplication* generates multiple execution points triggering the event according the global tempo and eliminating the time inaccuracy in *module event* duration.

Practically it means that if the *module duration* is 3 bars, multiplication is set to 4 and the *module canvas* is placed to position 5 (fifth bar from beginning). Thegiven *module* program is then executed when *VTe* sequencer reaches numbers 5, 9, 13 and 17. If *VTe* speed is set to 80 BPM (duration of one *bar* is 4 *beats* = 3000 ms) then *module program* is executed exactly 15, 27, 39 and 51 seconds from the start of the composition.

3.1.4 Module duration

When the Module program is correctly implemented into *VTe,* its duration can be dynamically changed to synchronize loaded events with one another. This feature, known as time warp or fit in tempo, is performed by the program placed in *connector abstraction* and controlled by predefined *module gui switch*:

3.1.4.1 no fit

This option is default and just performs the *module program* in its original duration. This duration can be hard set by **$0-ORIG_DURATION** variable or calculated by *module program* (for example sample length). Event duration is independent from the VTe global speed and visualized by the length of corresponding *module canvas*. Note the *module canvas* then is VTe speed dependent and its length vary according the current *VTe BPM* showing real duration of event on timeline.

3.1.4.2 fit to bars

Switching to this option the **$0-ORIG_DURATION** is sent for recalculation according current *VTe* speed and selected number of *bars*. The new *event duration* is delivered by **$0-DURATION** variable for further program processing (can be fed for example into **[delay]** or **[line]**). This feature is indispensable when working with audio samples prepared for looping. Only in this option multiplication is supported. *Module canvas* displays module event aligned to bars so as real event time is.

3.1.4.3 fit to bpm

This lfinal option is useful for longer events which are not loop based, but have their own tempo (typically a cappella song versions or soundscape samples) . The length of event is calculated according to the **$0-ORIG_BPM** variable. To fit the event into *VTe* tempo correctly, this feature should be used in combination with starting point alignment.

3.1.5 Creation bang

Due to using **[obj(** function of pd (see 2.5.4 Module values storage) for dynamic module creation, all modules present in *Loaded_modules* window during creation of new module receive *loadbang* when any *module* is loaded. To avoid repetitive *loadbanging* of *modules,receive channel* is implemented which eliminates this behavior by blocking follow-up *bangs*. *Creation bang* receive should be always used instead of **[loadbang]** in *module program*.

3.2 Sends/receives

The sections below summarize *send* and *receive channels* connecting the *module* *program* with *VTe* through *connector abstraction*. Properly named **[send]** and **[receive]** objects are prepared in *module template* and connection can be established just by *cord connections*.

3.2.1 Module receives

List of channels sending data FROM *connector abstraction* TO *user program:*

•$0-CREATIONBANG : Sends bang only once when module instance is created.

•$0-EXECUTE : Sends bang at the beginning of every module event according to the horizontal position of module canvas on time line.

•$0-DURATION : Duration of module in milliseconds according to the switch "fit" (original or fited to current BPM). Rely on global BPM and "bars" value. Multiplication does not affect this value.

•$0-GET_POSITION : Horizontal position of module canvas on timeline.

•$0-GET_TRACK : Vertical position of module canvas on timeline. Could be used for sending data to different outputs according to the particular track settings.

•$0-ID : Number generated uniquely for each instance of any module. Could be used for referring to particular module. Module ID is dependent on unique abstraction variable ($0)and can´t be changed.

•$0-OUTPUT_TRACK : Number of output module output (audio1 or data) is sent to. Audio2 (if presented) is sent to output+1.

3.2.2 Module sends

List of channels sending data FROM *user program* TO *connector abstraction:*

•$0-SET_TRACK : Moves corresponding module canvas to a particular track on timeline (vertical move). Analogical to manual canvas moving. Module audio and data outputs may be affected by track change.

•$0-SET_POSITION : Moves corresponding module canvas to a particular bar on timeline (horizontal move). Analogical to manual canvas moving.

•$0-COLOR : Color of module background and appropriate module canvas. Color is defined by preset - see color table subpatch. Module is created in light gray color by default.

•$0-FIT : Time stretch switch. 0: duration of module event is original, 1: duration is calculated according to set original BPM of event, 2: duration is calculated according to set number of bars and multiplication in conjunction with current bar duration.

•$0-BARS : Defines how many bars the module event will last. Active only when fit switch is on. Duration of module event is stretched to time? x of bars?

•$0-MULTI : Defines multiplication of module event. Active only when fit switch is set to 2. Each next event starts from beginning right after another.

•$0-NAME : Name of each module instance. It appears also in module canvas. Usually typed manually or generated for example from file name. IMPORTANT: do not use spaces !

•$0-ORIG_DURATION : Original duration of module event in milliseconds. Set manually or calculate for example from sample table.

•$0-ORIG_BPM : Original BPM of module. For example BPM of audio loop.

•$0-MODULE_FILENAME : Filename of module without .pd extension. Used for module creation during loading saved composition.

•$0-AUDIO1 : Address module audio output nr. 1 to the audio channel defined in vt track properties (track properties are set in timeline window) according to the module canvas vertical position.

•$0-AUDIO2 : Address module audio output nr. 2.

•$0-DATA : Address module data output (stream of numbers) to the data channel defined in vt track properties (track properties are set in timeline window) according to the module canvas vertical position.

•$0-RESET_CNV_POSITION : Resizes current module canvas to default (create) size (= 0).Usually used when unloading/reset module

•$0-GOP SIZE : Height of Graph On Parent of module in pixels - used during loading for graphical module sorting.

3.2.1 Global receives

List of channels sending data FROM *VTe* TO *user program:*

•vt_seq_stop: Sends bang when sequencer stop button is pressed. Next "play bang" should trigger the module to perform from beginning (STOP is not PAUSE). All modules should implement this immediate stop function

(Sequence number is set to 0 after STOP)

•vt_bpm: Receives global VisualTracker BPM. To initiate the value send bang to vt_init_bpm channel.

•vt_bar: Receives number of current sequencer position counted in bars. This value is used as main trigger for module events.

•vt_beat: Receives number of current sequencer position counted in beats. Subdivision is made directly by sequencer.

•vt_time_sequence: Receives direct stream of sequencer **[line]**

3.3 Initial module values

Except default storage function **[mstorage]** *abstraction* inside *module* deliver the init function. *Creation argument* defines the number/symbol which is loaded into connected numbox/symbol box upon *module* loading. This feature works similarly to *module* re-creation with arguments stored in an external file relying on the fact that arguments which are not present upon loading are 0.

3.4 Module template

For easy user program implementation the main features including the basic gui design and switches described above are prepared in *template module* **vt_template_module.pd**. The best way is to compare further described *basic modules* with *module template* and learn directly form program code.

3.5 Basic Modules

Beside *module template,* there are three *basic modules* included in the basic *VisualTracker* installation which can be used as a base for further module development.

3.5.1 [vt_bang]

Easiest *module* which is performing **[bang]** on time defined by *module canvas*. Duration of event is irrelevant as so as it makes no sense to fit it in tempo or multiply, so additional features are logically missing. Values stored for this *module* are its position on timeline (track, position), name and color.

3.5.2 [vt_toggle]

Module operating **[toggle]**. The duration of the event is relevant already, so corresponding controls are present. Original duration is set by *gui numbox* and recalculated duration is fed into **[delay]** according to the fit switch**.** *Storage abstractions* are added to store additional values.

3.5.3 [vt_line]

Despite its simplicity, this *module* is using all the current features of *VTe*. The output of **[line]** sub-sequencer is prepared for controlling the playback of samples, videos or any other time based programs.

4 Future development

VisualTracker is built with a focus on continuity of development and the possibility to migrate easily to new versions. Therefore the program code and user data are strictly separated and external file storage is implemented (see 2.5 Composition storage).

In addition to regular testing of the current version and incorporation of user feedback, the following ideas are also planned

•Default *sub-sequencer* interface for Module template allowing further sequence division useful for samplers or pianoroll instruments.

•*Composition_timeline* facelift using advanced dynamic features of *data structures*

•*Recording pool* implementation allowing to record the output of VTe and share it across the modules (live sampling)

•Interface and controls for dynamic compositions where *module canvases* are shifted on timeline according to certain schemes (non linear timeline)

•Automation feature for value envelope controls (using arrays or data structures)

5 Documentation

5.1 VisualTracker program

User documentation and current downloads can be found at http://code.google.com/p/visualtracker. As *VisualTracker* is open source software, users are welcome to participate using the community features of the Google site like forums and issue tracking.

5.2 Module development

Detached page devoted to module development is established at http://code.google.com/p/vtmodules/

6 Releases

6.1 Program pack

The zipped program pack includes all main VisualTracker abstractions, module template and basic modules (see 3.5 Basic Modules**).**

6.2 Module pack

The pack of user modules is released independently from the main patch according to the growth of the user base containing complex modules and tools. For each pack, the module launcher is included to integrate modules into *Module_library*. All modules released as part of Module pack are reviewed by VisualTracker developers to ensure their compatibility.

7 Licence

VisualTracker is developed as open source software built in pure data environment (currently *pd-extended 0.42.5*, http://puredata.info/) and under GNU General Public License (http://www.gnu.org/licenses/gpl.html)

8 Conclusion

9 *VisualTracker* is regularly developed and tested for more than a year with the idea of cooperation and open source evolution with hope to attract pd fans and enthusiastic and tempt them to use *VisualTracker* as a platform for their experiments, extend the library of modules and come up with new ideas. The basic functions featured are already seen in proprietary software such as Ableton Live or ACID, but only in open source and 100% readable environment of Pd they can be really explored and used the way they were never used before.

10 Acknowledgements

The author would like to thank to whole pd community freely sharing their ideas, the PURE DATA forum (http://puredata.hurleur.com) contributors and namely to Bérenger Recoules for his enthusiasm regarding this project, his comments and above all the testing and early module development.

11 References

[1] Graphical user interface - http://en.wikipedia.org/wiki/Graphical_user_interface

[2] Timeline - http://en.wikipedia.org/wiki/Timeline

[3] PD documentation - http://www.crca.ucsd.edu/~msp/Pd_documentation/

[4] Measure - http://en.wikipedia.org/wiki/Bar_(music)

[5] Beats per minute - http://en.wikipedia.org/wiki/Tempo

[5] Phrase - http://en.wikipedia.org/wiki/Phrase_(music)

[6] **[coll]** is part of cyclone library