Marian Weger, BSc

Kollabs/DS - a state-saving system with scene morphing functionality for PureData

Project Report

Graz University of Music and Performing Arts

Institute of Electronic Music and Acoustics

Supervisor: DI IOhannes m Zmoelnig

Graz, September 2014

kunst uni graz

institute of electronic music and acoustics

iem

Abstract

Nowadays, most professional music production software or hardware equipment implies the ability to store and recall presets of all its settings. Also cross-fading between different scenes is now widely-used, especially in stage lighting. However, in theater productions, shows or installations, it is often required to create custom software solutions for sound and video processing or for controlling special hardware. To enhance the possibilities of the graphical data-flow programming environment *Pure Data* in such applications, there arises the need for an appropriate state-saving system.

This work documents the design and evaluation process of *Kollabs/DS*, a scene-based state-saving solution for *Pd*, featuring versatile transition features and play-lists. Through bidirectional communication via *OSC*, *MIDI*, etc., it can also be used to control other hard- or software, such as synthesizers or digital mixing consoles.

Kollabs/DS is based entirely on *Pd-Vanilla* abstractions and can therefore easily be used on any platform and operating system, which *Pd* supports. It is part of the *Kollabs* library, which aims to become a complete solution for not only state-saving, but also general data management and communication.

Kurzfassung

Professionelle Software oder Hardware Equipment für Musikproduktionen beinhaltet heutzutage meist eine Möglichkeit zum Speichern und Wiederherstellen sämtlicher Einstellungen. Auch das Überblenden zwischen verschiedenen Szenen ist mittlerweile weit verbreitet, besonders bei Bühnenbeleuchtung. Nichtsdestotrotz ist es bei Theaterproduktionen, Shows und Installationen oft nötig, spezielle Software zu entwickeln, um besonderen Wünschen in Audio- und Videoverarbeitung oder der Steuerung spezieller Hardware nachzugehen. Um die Möglichkeiten der graphischen Datenfluss-Programmierumgebung *Pure Data* für solche Anwendungsfälle zu erweitern, ist auch hierfür ein geeignetes Speicher-System notwendig.

Diese Arbeit beschreibt den Entwicklungs- und Evaluierungsprozess von *Kollabs/DS*, einer szenen-basierten Speicherlösung für *Pd*, die auch vielfältige Möglichkeiten für Playlists und Überblendungen zwischen einzelnen Szenen bietet. Durch bidirektionale Kommunikation über *OSC*, *MIDI*, etc. kann das System auch genutzt werden, um externe Hard- oder Software, wie z.B. Digitalmischpulte, zu steuern.

Kollabs/DS basiert ausschließlich auf *Pd-Vanilla* Abstraktionen und kann daher auf jeder Plattform genutzt werden, die von *Pd* unterstützt wird. Das System ist Teil der *Kollabs* Library, mit der versucht wird, eine einheitliche Lösung, nicht nur für Szenenspeicher, sondern auch für generelles Daten-Management und Kommunikation, zu schaffen.

Contents

Part I.

Development

1. The problem with state-saving in PureData

As *Pure Data* (short: *Pd*) is no ready-to-use program, but rather a versatile graphical data-flow programming language, it does not provide an applicable system for saving the state of variables and tables. A *Pd*-patch can easily store the algorithm and one initial state for all the variables, but out of the box, it is not possible to store different variations of these.

There is rudimentary state-saving built into tables and some of the graphical objects, but these store only the current status of the patch. There is no possibility to recall the data during a running patch, or to store different scenes. However, for installations, shows or theater productions, there is the need of storing and recalling different static scenes during the performance, and even fading/morphing between them.

This approach has been implemented in many commercial products such as lighting and sound mixing consoles and synthesizers, both hardware and also software. There are also several solutions available for *Pd*, which implement the most basic functions. These shall also be examined in terms of sufficiency in the next chapters.

2. What a state-saving system should provide

2.1. What needs to be stored

Stored should be preferably anything that can be sent as messages through `send` and `receive`. *Pd* distinguishes between numbers (all numbers are treated as floating point numbers) and symbols. There are also lists, which can be built out of arbitrary combinations of numbers and symbols.

Additionally, tables should be handled by the state-saving system. In fact, they can be seen as lists of numbers too, but need to be treated differently.

2.2. Scene transitions

For shows and installations, there is the need to not only store different scenes, but also morph between them. This means, there must be some kind of smooth transition from the values in one scene to the values in another scene. To be prepared for all possible situations, an individually adjustable transition for every variable and every scene is considered necessary.

However, smooth transitions are only meaningful for numbers, and not for symbols. A possibility to morph between lists of numbers, which also includes tables, would be a nice feature, but not a primary goal, as small lists can always be split into single numbers and stored individually.

2.3. Re-structuring scene-sets

Another goal would be to edit the list of scenes, how it is common in file browsers and other computer software. The most basic functions include copy, paste, cut and delete individual scenes. When pasting scenes, two cases should be distinguished: Pasting in between two scenes and pasting by overwriting one existing scene. Spreadsheet software, like *Microsoft Excel*[1] or *LibreOffice Calc*[2] should act as a guideline for this behavior.

2.4. Sequencing

By now, most Modern Digital Audio Workstations (DAWs) and mixing consoles provide time-line-editing of their own parameters, and also of external controllers via MIDI. As the MIDI specification is a relatively old standard, which uses only limited resolution, some protocols, like the Mackie protocol, were developed, which build on top of MIDI, but are able to use better resolution for critical parameters, such as volume control. Anyway, these have to deal with the extremely limited bandwidth of MIDI.

In the meantime, some companies rely on their own proprietary protocols, which use for example common Ethernet cables to connect control surfaces to the DSP unit or PC. However, such protocols, like *EuCon*[3], are no open standard and can therefore only be used under some restrictions.

Some DAWs can handle OSC messages, which is sometimes seen to be the successor of MIDI, but as an OSC message can contain various types of data, the handling is a bit more complicated. The DAW software *REAPER*[4] supports OSC and can even use its time-line editing view for such data, but for a live setup, most DAWs or MIDI-sequencers do not seem to be the perfect solution.

[1] *Excel* is a registered trademark of *Microsoft Inc.* and part of the *Microsoft Office* suite.

[2] *Calc* is part of the free and open source office suite *LibreOffice*, developed by *The Document Foundation*.

[3] *EuCon* protocol: http://euphonix.avid.com/pro//music/eucon.php

[4] *REAPER* is a digital audio workstation by *Cockos Inc.*: http://reaper.fm/

As there should not be a limitation in the type of variables, which are stored (see 2.1), a graphical time-line editing, like in a DAW or MIDI-sequencer, would be a rather difficult task. But for shows, which require an exact time-schedule, there is the need for pre-programmed scenes, which are triggered automatically in some sort of play-list.

The most promising work-flow would be the way how modern lighting consoles operate. They provide storable cues, which can each contain a small sequence, which can individually be edited in a time-line view. This way, the best of the two worlds, cue lists and sequences, are combined.

3. Available solutions

3.1. State-saving in modern lighting consoles

Because of their versatile state-saving and transition functionality, modern stage lighting consoles act as a benchmark for *Kollabs/DS*. Even very simple lighting controllers have the ability to blend smoothly between two independent scenes. Fig. 3.1 shows such a manual two-scene preset board, with control for two states of 12 individual lights (two layers of 12 faders each) and master faders to mix between these two.

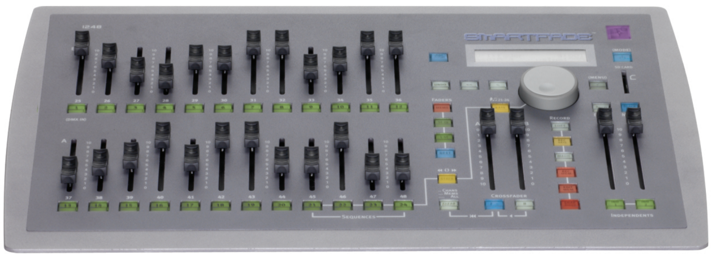


Figure 3.1.: The *SmartFade Console* by *Electronic Theatre Controls (ETC)*: `http://www.etcconnect.com/`

Such basic boards are often used for small venues. Anyway, for bigger productions, with lots of complex lighting scenes, computerized consoles have become standard. In these, the amount of available scenes is almost infinite, with the drawback of being less intuitive in operation. In state-of-the-art consoles, such as the *GrandMA* series by *MA Lighting* (see Fig. 3.2), complex scenes can also contain movements, which are edited in a time-line-view known from sequencers. The scenes can be triggered by hand or scheduled to be played back automatically. This way, a whole show can be pre-produced and played back identically in every performance, minimizing the chance for human failure.

Figure 3.2.: The *GrandMA2* console by *MA Lighting*: http://www.malighting.com/

More information on stage lighting can be obtained online in the *Stage Lighting Primer* by Salzberg and Kupferman, 2013.

3.2. State-saving in *Pure Data*

There are already many different state-saving solutions available for *Pure Data*. In this chapter, some of the most popular will be examined in terms of functionality and usability.

3.2.1. sssad

A very popular system is *sssad* (see Barknecht, 2008), the *Stupidsupersimplistic State Saving ADVANCED*. It has been used by many people in different projects, and has proved to be very stable and efficient. As it is built entirely in *Pd-Vanilla*, it does not need any external libraries and runs on all operating systems, for which *Pd* is available - even *Android*[1] and *iOS*[2], through *libpd*[3]. However, it provides only basic scene-based state-saving with no transitions, scheduling or editing features.

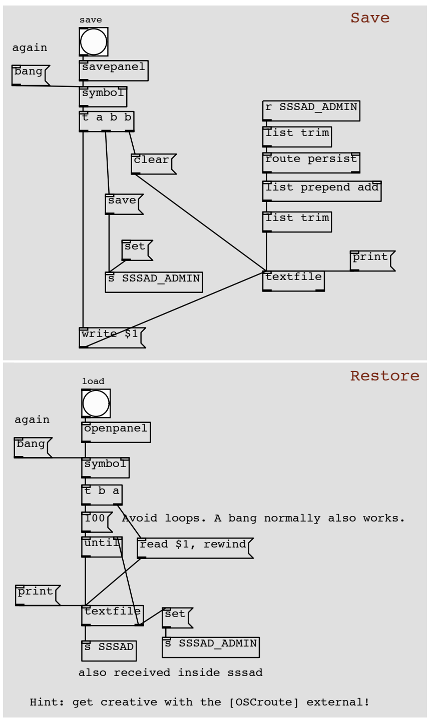


Figure 3.3.: Saving and recalling in *sssad*.

[1] *Android* is a trademark of *Google Inc.*: http://www.android.com/

[2] *iOS* is a registered trademark of *Cisco* and used under license by *Apple Inc.*: http://www.apple.com/ios/

[3] *libpd*: https://github.com/libpd

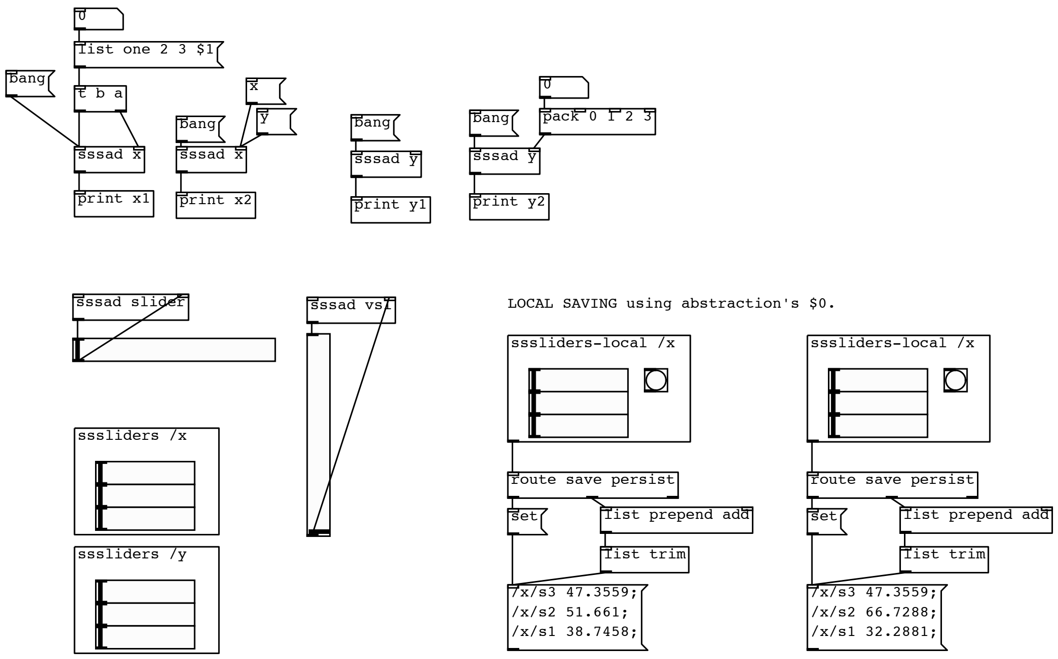


Figure 3.4.: The help patch for *sssad*.

3.2.2. save.me.mmb

The state-saving system from the *mmb* library (see Moser-Booth, 2011) provides similar functionality as *sssad*, but seems to be a bit less spread in the *Pd*-community, and has therefore not been tested as extensively. Other than *sssad*, it depends on several externals from the *Pd-extended* distribution and can not be used with the pure *Pd-Vanilla* version. There is also no possibility for creating scene transitions.

3.2.3. preset hub

The *Pd-L2Ork* distribution (see Bukvic, n.d.) provides a promising state-saving system, called *preset_hub*. It is highly integrated into *Pd-L2Ork* and therefore can not be used in the pure *Pd-Vanilla* version. This could make it difficult to use it on uncommon operating systems. Out-of-the-box, it lacks the ability to morph

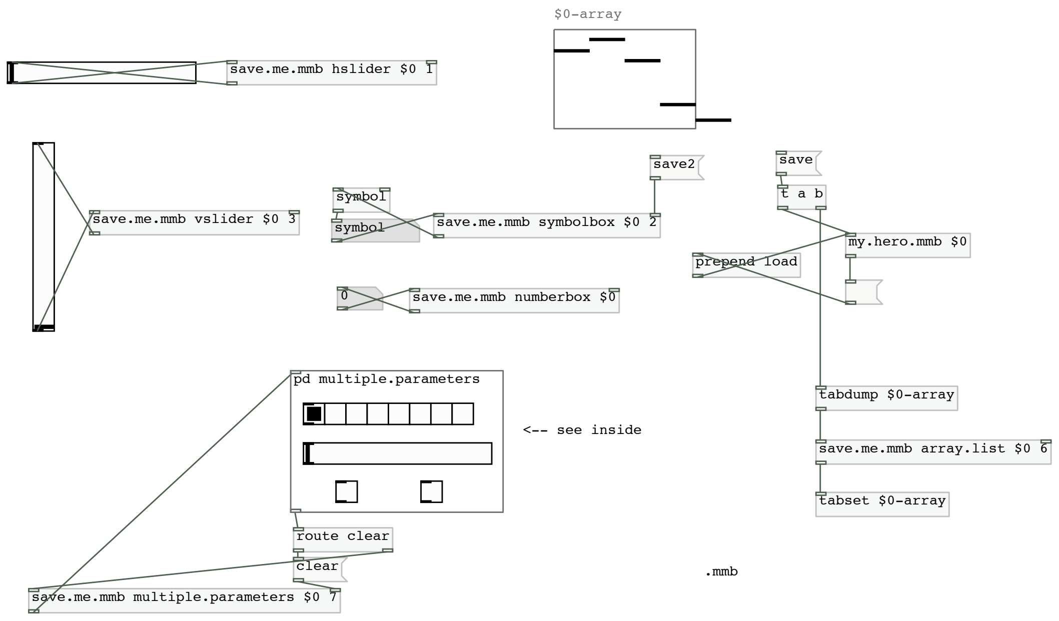


Figure 3.5.: The help patch for *save.me.mmb*.

between scenes, but has a nice feature of loading mixtures of different scenes: there can be given specific amounts in percent for every scene.

4. The design of Kollabs/DS

As Pd is lacking a suitable solution for morphing between scenes, there is the need of a whole new state-saving system. This chapter describes the design of Kollabs/DS.

4.1. Surveillance of the variable states

To save the state of variables, and write (recall) values back to them, they somehow need to be addressed.

In control flow programming languages, such as *C* or *SuperCollider*, it is common to declare and initialize variables like this:

```
int a = 1;
```

and then use them anywhere else in the program with the aid of their unique name:

```
int b = a + 2;
```

As *Pd* has a data-flow design, a more ore less equal result can only be achieved with some workarounds (See Fig. 4.1). There is no practicable possibility to define variables directly, but only the resulting data-flow can be given a unique name. The possibility of using any number of `send` and `receive` objects with the same name to organize the data-flow, leads to the solution of how this stream of data can be surveyed.

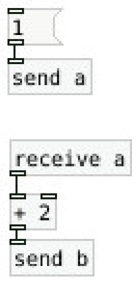


Figure 4.1.

It makes only sense to save variables, which can be controlled by some kind of graphical user interface (GUI) or hardware controller, so it would be sufficient to just observe the (bidirectional) data-stream between GUI and the actual signal processing (DSP). To be sure, that there is always such a data stream available, GUI and DSP should be strictly separated. The simplified communication structure can be seen in Fig. 4.2.

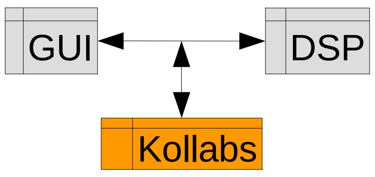


Figure 4.2.: Communication structure of *Kollabs/DS*

The patch shown in Fig. 4.1 already implies, how *Kollabs/DS* can hook into the patch to get the data streams of the individual variables. The data-flow between two objects in *Pd* can be cut and sent "wireless" with the `send`/`receive` objects, as shown in Fig. 9.1.

4.2. Multiple instances

To have multiple instances working properly side-by-side, it must be possible to distinguish between them. They could generate a unique identifier by themselves, but to address them, it seems convenient to just give simple and memorizable names. It has been decided, that the first creation argument of the main abstractions of the state-saving system is always this so-called domain. Each variable can then be assigned to one of them.

The domain name should not be written to the saved file, so that one domain can load the file of another.

The possibility of having multiple domains in one patch can be useful, for example when some settings deal with the technical environment of a show, and others with the actual artistic content. To not mix them up, they can then be saved to different domains and saved/recalled individually.

4.3. Scene transitions

For the scene transitions, some important decisions have to be made. There is one big problem: In a defined sequence of scenes, there is always one transition between two successive scenes. But if scenes are played back in undefined order, also the transitions can not easily be defined. The number of possible transitions would then be the factorial of the number of scenes.

To simplify usage and also implementation, the scene transitions are limited to one per scene. Most important for me is the transition **to** a scene, so this should be the one editable. This means, each scene can have only one transition, which leads to itself, from whatever origin.

4.4. File format

The file format of the saved data is also a critical point, which needs to be discussed carefully. As all the saved data can be perfectly described in plain text format, it is obvious, that it should be saved as a text file.

Sometimes it might be handy to edit some data by hand, so it would be nice to have the data format as human-readable as possible. It seems convenient to use a new line for each variable, with a blank space for separation between the variable name and the corresponding data.

To have one show in a single file, multiple scenes of one domain should fit into. To separate between different scenes and also types of data, some unique header lines must be inserted.

It is also necessary to store the scene transition settings in the text file. These settings are individual for each scene, so they must be saved accordingly. To distinguish between transition settings and the actual scene data, there must be additional header lines for those two parts. The transition data should better be placed at the beginning of each scene, as these variables must be read first in any case.

4.5. Usability

Despite its complexity, using this state-saving system must remain simple and intuitive. A variable should be registered by only one simple abstraction, giving both the storage domain and variable name as creation arguments. Also the main logic must be put in one single abstraction, with a separated corresponding GUI, which can optionally be omitted. For most users, this will be sufficient. For more complex features, such as scene transitions, additional modules can be added, which link automatically in the system.

5. The implementation of Kollabs/DS

This chapter describes, how the core of *Kollabs/DS* is implemented.

5.1. Basic functionality

5.1.1. Surveillance of the variable state

The variables are surveyed by using their unique send- and receive-name (see 9). Inside the according register `ds_reg`, the current value of the variable is always stored by a `list` object (See Fig. 5.1).

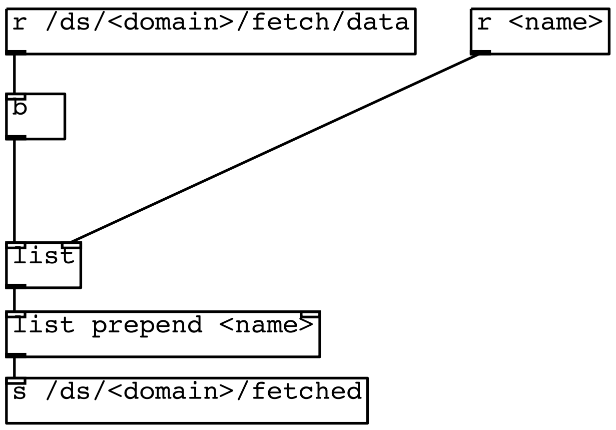


Figure 5.1.: The principle for surveying and fetching variables inside `ds_reg`.

5.1.2. Storing/recalling scenes to/from memory

The individual scenes are stored in a dynamically created slot, consisting mainly of one `textfile` object. To store a scene to memory, all registered variables of a specific domain are fetched from the registers `ds_reg` to the main storage logic `ds_logic` by sending a trigger signal to `/ds/<domain>/fetch/data` (See Fig. 5.1). For each variable, its name and value are then sent to `/ds/<domain>/fetched` and then forwarded to the according `textfile` object, where each variable is stored in one individual line of the buffer.

To recall the scenes from memory, the buffer inside the according `textfile` of the scene is emitted line by line and the values are routed to their send/receive name (See Fig. 5.2).

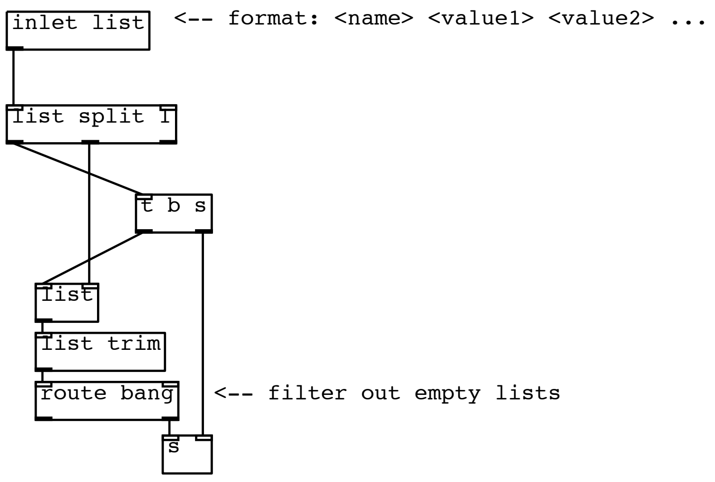

inlet list
<-- format: <name> <value1> <value2> ...
list split 1
t b s
list
list trim
route bang
<-- filter out empty lists
s

Figure 5.2.: The principle of routing stored values back to the variables (recalling).

5.1.3. Saving/loading scene-sets to/from disk

Whole scene-sets are saved to disk by multiplexing all the single scene buffers to one "master" `textfile` object, whose contents are then saved to a plain text file on the hard drive. The scenes are separated by a header line:

```
======================= SCENE <nr> =======================.
```

The scene-sets can be loaded back from disk into the main buffer and are then routed to their individual scene-buffers according to their header.

5.2. Scene management

5.2.1. Scene-set editing (copy, paste, etc.)

To have the ability to respond fast to changes in the scene order, or built a new scene on top of an already existing one, there is the need of basic scene-editing features, such as copy, paste, cut, etc. There is one scene-buffer extra, which serves as a clipboard. Scene contents can be copied to it or forwarded to another buffer.

If a scene gets completely deleted, the contents of the subsequent buffers are routed one-by-one to the preceding buffer. This obviously very inefficient action is used, because it is relatively easy to implement and the affected editing actions are usually not used during a running show. The simple changing of the scene numbers for the buffers would also introduce a higher risk of addressing errors.

5.2.2. Scheduling and naming scenes

By giving each scene a duration, the scenes can be scheduled in a play-list-like behavior. A simple timer triggers the next scene, if the according duration has passed. There are two clock sources available:

The first source is a `metro` object, whose rate (= precision) can be set in milliseconds. This clock source relies on the CPU time, which means it pauses if the CPU has no resources available to calculate it. This could result in imprecise timing if there is heavy CPU load.

Alternatively, a `realtime` object can be placed after `metro` to improve the timing precision. `realtime` measures time according to time calculated by the operating system.

The scene duration and also a scene name can be stored for each scene individually. To do so, each scene is divided into different sections by specific headers, just like the scenes separated when saved. Each scene consists of a properties section for duration and name, and a data section for the actual data.

5.3. Scene transitions

In addition to the properties and data section in the individual scenes, there is a settings section, which contains all the settings for scene transitions. To be able to introduce transitions for each variable, they must not be routed directly to their send/receive name on every recall, but first pass through more complex data processing.

When a scene gets recalled, it is important that the transition settings are recalled before the actual data. The data is then routed back in the individual register abstractions `ds_reg` for each variable, to apply individual transitions there.

5.3.1. Morph/Fade

A linear ramp from zero to one is generated by a `line` object with the according fade time. To enable curved fades the linear ramp is shaped by a small abstraction, which is called `fadecurve`. It basically takes two parameters: *shape* and *weight*.

The influence of *shape* on the fade curve is described mathematically in the following table:

shape	f(x)
≥ 2	$x^{abs(shape)}$
$1 < shape < 2$	$\frac{1}{2}(1 - \cos(\frac{\pi}{2}x) + x^{abs(shape)})$
1	$1 - \cos(\frac{\pi}{2}x)$
$0 < shape < 1$	$\frac{1}{2}(\sin^2(\frac{\pi}{2}x) + 1 - \cos(\frac{\pi}{2}x))$
0	$\sin^2(\frac{\pi}{2}x)$
$-1 < shape < 0$	$\frac{1}{2}[\cos(\frac{\pi}{2}(1-x)) + \sin^2(\frac{\pi}{2}x)]$
-1	$\cos(\frac{\pi}{2}(1-x))$
$-2 < shape < -1$	$\frac{1}{2}[1 - (1-x)^{abs(shape)} + \cos(\frac{\pi}{2}(1-x))]$
≤ -2	$1 - (1-x)^{abs(shape)}$

This rather complex description is explained more simple in 12.6.2 Morph. *Weight* additionally blends between the selected shape and a linear ramp.

The final shaped ramp (from zero to one) is then applied to the variable (from old value to new value). As this type of fading does only make sense for numbers, it is only applied to values of type "float" in *Pd*.

5.3.2. Delay

A delay is implemented by buffering the variable in a `list`, which is then triggered to output after a certain time has passed.

For a constant delay, a simple `delay` object is used (See Fig. 5.3). To chain a recall to another one, which means recall only after another variable has finished its transition, an additional external trigger input is introduced (See Fig. 5.4). Each variable creates a trigger signal, when its transition has finished.

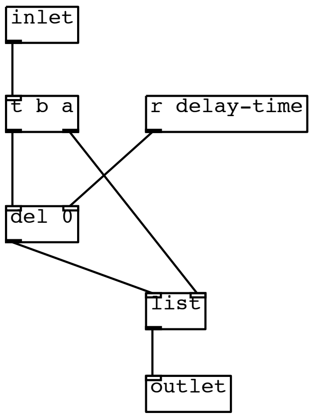


Figure 5.3.: The principle of delaying recalls for individual variables.

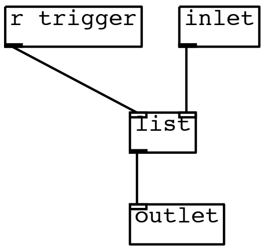


Figure 5.4.: The principle of triggering a delayed recall from an external trigger source.

5.3.3. What if last transition is still active?

If a scene is recalled during an active transition, there are two ways of handling this problem:

The system could on the first hand just block all new commands, until the transition has finished, or on the other hand stop the current transition and start the second transition from that point. Both variants are implemented and switchable per variable and scene or even globally.

5.3.4. Dynamic creation of transition functionality

As these transition features make the *Pd*-patch big and slow, they are dynamically loaded only when needed. To enable scene transitions, a `ds_transition` module must be created. Only if it exists, which means the bidirectional communication with the individual registers succeeds, the corresponding modules are loaded inside the registers.

Part II.

User Manual

6. What is Kollabs/DS

Kollabs/DS is a set abstractions aiming on scene-based state-saving for the data-flow programming language *Pure Data* (*Pd*). It offers some advanced scene transition features for each variable and scene, such as morphing, delaying, etc. and also basic time-line editing for programming complete shows.

It is part of the more general abstraction library *Kollabs*, which also provides solutions for MIDI[1] and OSC[2] communication.

[1]Musical Instrument Digital Interface: `http://www.midi.org`
[2]Open Sound Control: `http://opensoundcontrol.org`

7. Requirements

Kollabs/DS relies only on pure *Pd-Vanilla*[1], which means it does not need any external libraries.

[1] *Pd-Vanilla* is the most basic version of *Pd*, mainly written by Miller Puckette: http://msp.ucsd.edu/software.html

8. How to add *Kollabs* to *Pd*

Kollabs can be added to *Pd* in multiple ways.

8.1. Inside a patch

Open the patch in which you want to use *Kollabs* and add the following object:

```
declare -path /path/to/kollabs
```

where `"/path/to/kollabs"` needs to be replaced by the absolute or relative path to the `"kollabs"` folder. This is only applied after saving and re-opening the patch.

8.2. Through arguments for *Pd*

The path can also be given directly to *Pd* by starting the program with additional arguments from a terminal. For *UNIX*[1]-like operating systems, open a terminal and run the command `"/path/to/pd -path /path/to/kollabs"`.

`"/path/to/pd"` needs to be replaced by the absolute path of the *Pd* program. `"/path/to/kollabs"` needs to be replaced by the absolute path of the `"kollabs"` folder.

[1]This includes *Mac OS* and *Linux*: `http://en.wikipedia.org/wiki/Unix`. *UNIX* is a registered trademark of *The Open Group*. *Mac OS* is a registered trademark of *Apple Inc.*

8.3. Inside *Pd*

An easy way to add a path is to use the Graphical User Interface (GUI) of *Pd*: Click on *Preferences*→*Path...* in the menu and add the absolute path through a dialog.

9. General patching principles

Patches for *Pd* can be programmed in many ways. Anyway, *Kollabs* is based on some basic patching principles that are explained below.

9.1. Working with sends and receives

One major principle, which should be considered is to separate the GUI controls from the actual data processing. In *Pd*, this is done by using sends and receives instead of line connections (See Figure 9.1).

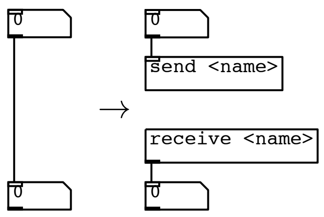
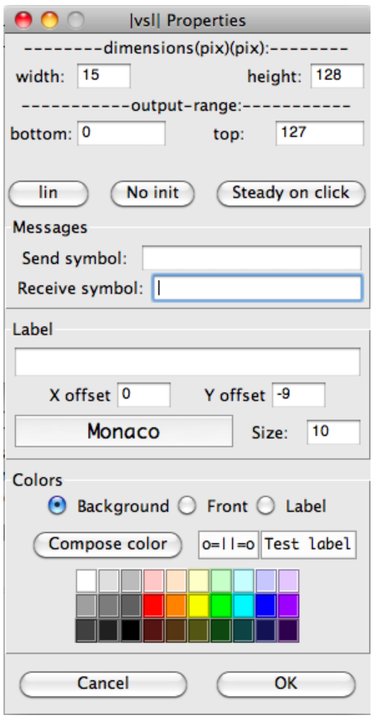


Figure 9.1.: Usage of sends and receives instead of line connections in *Pure Data*.

If a GUI parameter should communicate bidirectional, which means it receives and sends at the same time, a feedback prevention is necessary to guard against endless loops. The following objects have built-in feedback prevention:

◯ *Bang*, ☒ *Toggle*, ▷1 *Numberbox2*, ▮ *Hslider*/*Vslider* and ■□ *Hradio*/*Vradio*.

Sends and receives can be set directly in their properties dialog (*right-click* → *Properties*) under *Send symbol* and *Receive symbol* (See Figure 9.2). *Kollabs* relies on the concept of GUI objects with identical *Send symbol* and *Receive symbol*. In the following, such pairs of `send` and `receive` with the same name are treated as variables.



|vsl| Properties

--------dimensions(pix)(pix):--------

width: 15 height: 128

-----------output-range:-----------

bottom: 0 top: 127

lin | No init | Steady on click

Messages

Send symbol:

Receive symbol:

Label

X offset 0 Y offset -9

Monaco Size: 10

Colors

(●) Background ( ) Front ( ) Label

Compose color | o=||=o | Test label

Cancel | OK

Figure 9.2.: Properties dialog of a GUI element in *Pure Data*

9.2. Naming guidelines

There is no special naming syntax needed for variables in *Kollabs*, but a beginning with "/" is highly recommended to conform with the OSC standard. It is also proposed to use variable names consisting only of lowercase letters and slashes, in the form of "`/<category>/<subcategory>/.../<name>`". Anyway, in general, variables can have any name that PD allows.

10. Syntax

10.1. The different storage layers

Kollabs uses five different storage layers, which are recalled in the following order: *global, properties, settings, tables, data*. Each of these layers contains a different part:

Global The *global* layer saves all data, that should stay the same for all scenes.

The other four layers contain data, that changes from scene to scene:

Properties The *properties* of the scene contain only two variables: Its name and duration.

Settings Transition settings for the individual variables are located in the *settings* layer.

Now comes the actual data:

Tables Tables can be found in the *tables* layer, before the other data.

Data The actual data of the individual variables is located in the *data* layer.

10.2. File layout

A scene-set is saved to a simple text file with the following structure (see listing 10.1). There is one section for the *global* variables, followed by the individual *scene* sections, with each containing the four layers *properties, settings, tables* and *data*.

Listing 10.1: File Layout

```
================================ GLOBAL ================================
<variable_name_1> <value>
<variable_name_2> <value>
...

================================ SCENE <nr> ================================
# PROPERTIES
/name <value>
/duration <value>
...

# SETTINGS
<setting_name_1> <value>
<setting_name_2> <value>
...

# TABLES
<table_name_1> <value1> <value2> ...
<table_name_2> <value1> <value2> ...
...

# DATA
<variable_name_1> <value1> (<value2> <value3> ...)
<variable_name_2> <value1> (<value2> <value3> ...)
...
```

10.3. Creation arguments and flags

In general, objects inside *Pd* can take creation arguments, which have to be set at their exact position:

```
<object> <arg1> <arg2> ...
```

10. Syntax

In *Kollabs*, only the few mandatory creation arguments need to be given in the right order. All following optional arguments are called *flags* and can be given in any arbitrary order:

```
<object> <arg1> <arg2> ... <flag1> <flag2> ...
```

For most abstractions of *Kollabs/DS*, the first creation argument is always the *domain*. This is a unique identifier to define, which abstractions belong together to the same instance of *Kollabs/DS*:

```
<object> <domain> (<arg2> ...) (<flag1> <flag2> ...)
```

The creation arguments as well as the flags for the individual objects of *Kollabs/DS* are explained further in the following file reference.

11. File reference

This library contains plenty of abstractions, with many of them being solely used inside others and are irrelevant for the end-user. This chapter explains only the most important ones, that cover all functionality being discussed in this document.

11.1. Main abstractions

All the data processing is happening in these main abstractions:

`ds_logic`	Provides the basic functionality.
`ds_transition`	Needed for scene transitions.
`ds_scheduler`	Play-list view. Needed for scheduling scenes.
`ds_reg`	Needed for registering variables to the system.
`ds_reg_global`	Needed for registering global variables to the system.

11.1.1. ds_logic

`ds_logic` is the main abstraction, which creates a storage domain.

Creation arguments

There is only one creation argument:

```
ds_logic <domain>
```

domain The first creation argument defines the storage domain, to which variables can get registered to. It is possible to create multiple domains with different names by adding additional instances of `ds_logic`, but it is very important that each domain exists only once, which means there are not two instances with the same identifier.

Flags

There are two optional flags, that can be given in any order after the domain. Both will bypass the transition features:

simple The data-stream is directly routed to its (assumed) receivers, without further processing through the registers (all register settings will be bypassed). This is useful, if only very basic state saving is required. This way, even variables, that are not registered, can be recalled.

nodispatch If set, the data will not get dispatched on a recall. Nevertheless, at every recall, the whole data-stream (variable names and values) gets dumped to the first outlet. Get creative!

11.1.2. ds_transition

`ds_transition` provides scene-transition functionality to the registers. To enable scene transitions for a storage domain, there must be created one `ds_transition` abstraction with the same identifier.

Creation arguments

`ds_transition` takes only one creation argument:

`ds_transition <domain>`

domain This assigns the abstraction to a specific storage domain.

Flags

none.

11.1.3. ds_scheduler

By using `ds_scheduler`, a play-list of scenes can be managed. Scenes can be given a specific duration, after which the next scene will be recalled automatically. `ds_scheduler` creates the whole needed logic for the time-line functions.

Creation arguments

`ds_scheduler` takes only one creation argument:

`ds_scheduler <domain>`

domain This assigns the abstraction to a specific storage domain.

Flags

none.

11.1.4. ds_reg

`ds_reg` manages the surveillance of the individual variables and assigns them to a storage domain:

`ds_reg <domain> <name>` registers a variable `<name>` to domain `<domain>`.

The state of the registered variable will then get tracked and can be stored in different scenes.

Creation arguments

domain This defines the storage domain, to which the variables get registered.

name The second creation argument specifies the name of the variable, which is to be registered.

Flags

The following optional *flags* can be given after the creation arguments in any order:

nomorph The *nomorph* flag deactivates morphing for this register permanently. If set, the corresponding parameters in the transition dialog will be grayed out. They are still visible, but will not be stored anymore.
For some types of variables, it is never desired to morph between two states (for example lists, symbols, toggles, etc.). To save computing power and keep file sizes small, the morphing functionality should be turned off for such variables.

delay <time/s> The delay settings can be set permanently by creation arguments: The *delay* flag followed by a delay time in seconds will set the delay time permanently. It is then not possible anymore to change the delay in the register settings dialog. The according GUI controls will be grayed out. They are still visible, but will no more be saved with the storage.
EXAMPLE: `ds_reg main variable1 delay 5` will register *variable1* to domain *main* and delay all its recalls by five seconds.

nodelay turns off the delay permanently. It is not possible anymore to change the delay in the register settings dialog. This has the same effect as `"delay 0"`. The according GUI controls will be grayed out. They are still visible, but will no more be saved with the storage.

slave turns on slave mode permanently. It is not possible anymore to change it in the register settings dialog. The according GUI controls will get grayed out and will not be saved anymore with the scene.

noslave turns off slave mode permanently. It is not possible anymore to change it in the register settings dialog. The according GUI controls will get grayed out and will not be saved anymore with the scene.

occupy turns on *occupy* permanently. It is not possible anymore to change it in the register settings dialog. The according GUI controls will get grayed out and will not be saved anymore with the scene.

nooccupy turns off *occupy* permanently. It is not possible anymore to change it in the register settings dialog. The according GUI controls will get grayed out and will not be saved anymore with the scene.

direct *direct* = *nomorph* + *nodelay* + *noslave* + *nooccupy*. The *direct* flag is a shortcut for *nomorph*, *nodelay*, *noslave* and *nooccupy* combined. This means, most of the transition features will get bypassed permanently.

data_prepend <symbol> Prepend any symbol to the recalled data.
EXAMPLE: `ds_reg main variable1 data_prepend set` will register *variable1* to domain *main* and prepend the symbol *set* to the recall data. This means, if the data was `"0 8 15"`, it will become `"set 0 8 15"` when recalled. This is especially useful when GUI elements without feedback prevention are used (`symbol` *Symbol*, `0` *Number*).

data_append <symbol> Append any symbol to the recalled data.
EXAMPLE: `ds_reg main variable1 data_append foo` will register *variable1* to domain *main* and append the symbol *foo* to the recall data. This means, if the data was `"0 8 15"`, it will become `"0 8 15 foo"` when recalled.

name_prepend <symbol> Prepend any symbol to the variable name for recalls.
EXAMPLE: `ds_reg main variable1 name_prepend foo/` will register *variable1* to domain *main* and prepend the symbol `"foo/"` to the variable name for recalls. This means, the state of *variable1* will get recalled to `"foo/variable1"`.

name_append <symbol> Append any symbol to the variable name for recalls. EXAMPLE: `ds_reg main variable1 name_append /foo` will register *variable1* to domain *main* and append the symbol `"/foo"` to the variable name for recalls. This means, the state of *variable1* will get recalled to `"variable1/foo"`.

change By default, saved variables always get dispatched, even if the recalled value does not differ from the current value. To save some computing power, the change flag can be set to load only the differences on every recall. This function is similar to the `change` object in *Pd*.

nochange Explicitly dispatch all values, even if the do not differ from the current value. This flag is redundant, as it does not change the default behavior of *Kollabs/DS*.

table This flag must be set, if the registered variable is a table.

resize If the variable is a table and the *resize* flag is set, it will get resized automatically to the new size on every recall.

noresize If the variable is a table and the *noresize* flag is set, it will never get resized automatically. This flag is redundant, as this is the default behavior of *Kollabs/DS*.

11.1.5. ds_reg_global

With `ds_reg_global`, variables can be registered to the *global* layer of a scene-set (See 10.1). For more information on global variables, see 12.2.3.

`ds_reg_global <domain> <name>` registers a global variable `<name>` to domain `<domain>`. Its status will be saved with the scene-set and recalled when the scene-set is loaded.

Creation arguments

domain This defines the storage domain, to which the variable gets registered.

name The second creation argument specifies the name of the variable, that should be registered.

Flags

none.

11.2. Graphical User Interface

Some of the main abstractions have a corresponding Graphical User Interface (GUI) to access the controls and settings self-explanatory. In general, the object name of the GUI is constructed by appending `"_gui"` to the abstraction name. They all have only one creation argument, which is the domain:

`<object>_gui <domain>` creates the corresponding GUI for `<object> <domain>`.

The GUI objects follow the concept of identical send and receive names, which is explained in 9.1. The address for the sends and receives always begins with the header `"/ds/<domain>"`, to which an extension in the form of `"/name/subname"` is added. This leads to the full send/receive name `"/ds/<domain>/name/subname"`.

Of course, it is possible to create more GUIs for the same object and domain. Anyway, some dialog windows will only open once.

11.2.1. ds_gui

`ds_gui` (see Figure 11.1) provides the GUI for `ds_logic`. It contains all the controls for the basic scene-based state-saving functionality.

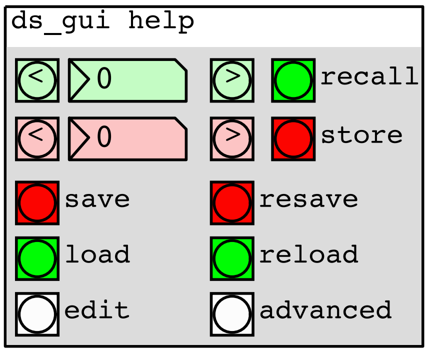


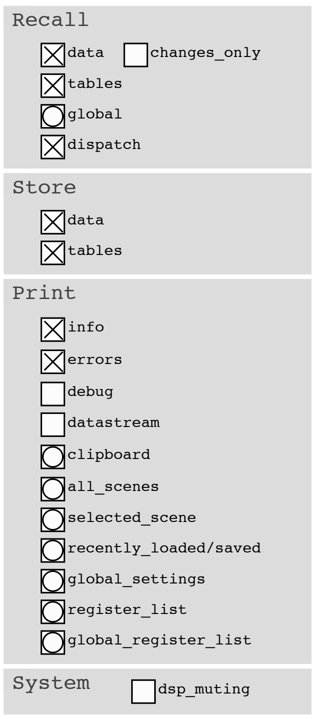
Figure 11.1.: GUI for `ds_logic` of domain *help*.

Main controls

GUI element	Address extension `/ds/<domain>...`	Description
<	`/scene/current/prev`	Decrement current scene by 1.
▷9	`/scene/current`	Set current scene.
>	`/scene/current/next`	Increment current scene by 1.
🟢 *recall*	`/recall`	Recall current scene from memory.
<	`/scene/selected/prev`	Decrement selected scene by 1.
▷26	`/scene/selected`	Set selected scene.
>	`/scene/selected/next`	Increment selected scene by 1.
🔴 *store*	`/store`	Store current scene to selected scene in memory.
🔴 *save*	`/save`	Open save dialog: Save the whole scene-set to disk.
🔴 *resave*	`/resave`	Save the whole scene-set to last specified filename on disk.
🟢 *load*	`/load`	Open load dialog: Load a scene-set from disk.
🟢 *reload*	`/reload`	Load the last specified filename.
○ *advanced*	`/advanced/vis`	Open the advanced settings dialog (See 11.2.1 Advanced controls).
○ *edit*	`/edit/vis`	Open the file editing dialog (See 11.2.1 Edit).

Advanced controls

The advanced controls (see Figure 11.2) are opened by clicking on ○ *advanced* in `ds_gui` (See 11.2.1 Main controls).



Recall
☒ data ☐ changes_only
☒ tables
◯ global
☒ dispatch

Store
☒ data
☒ tables

Print
☒ info
☒ errors
☐ debug
☐ datastream
◯ clipboard
◯ all_scenes
◯ selected_scene
◯ recently_loaded/saved
◯ global_settings
◯ register_list
◯ global_register_list

System ☐ dsp_muting

Figure 11.2.: Advanced controls for `ds_gui`.

Recall		
GUI element	Address extension `/ds/<domain>...`	Description
☒ *data*	`/recall/data/state`	If turned off, no data gets recalled.
☐ *changes only*	`/change`	If turned on, only changed variables will get dispatched.
☒ *tables*	`/recall/tables/state`	If turned off, no tables get recalled.
◯ *global*	`/recall/global`	Click to recall the global variables.
☒ *dispatch*	`/dispatch`	If turned off, nothing gets dispatched.

Store		
GUI element	**Address extension** `/ds/<domain>...`	**Description**
☒ *data*	`/store/data/state`	If turned off, no data gets stored.
☒ *tables*	`/store/tables/state`	If turned off, no tables get stored.

Print		
GUI element	**Address extension** `/ds/<domain>...`	**Description**
☒ *info*	`/print/info/state`	Print informational messages (i.e. store, save, load, etc.).
☒ *errors*	`/print/errors/state`	Print error messages.
☐ *debug*	`/print/debug/state`	Print debug messages.
☐ *data-stream*	`/print/datastream/state`	Print the whole datastream on every recall.
◯ *clipboard*	`/print/buffer`	Print the contents of the clipboard.
◯ *all scenes*	`/print/scenes`	Print the contents of all scenes in memory.
◯ *selected scene*	`/scene/selected/print`	Print the contents of the currently selected scene.
◯ *recently loaded/saved*	`/print/main`	Print the last loaded/saved data.
◯ *global*	`/print/global`	Print the current state of the global settings.
◯ *register list*	`/reg/print/dialog`	Open a list of all registered variables to print them individually.
◯ *global register list*	`/reg/global/print/dialog`	Open a list of all registered global variables to print them individually.

System		
GUI element	**Address extension** `/ds/<domain>...`	**Description**
☐ *DSP muting*	`/dsp/mute`	If activated, the DSP gets always switched off during dynamic patching within *Kollabs*.

Edit

A click on the ◯ *edit* button in `ds_gui` (see 11.2.1 Main controls) opens the edit menu (See Figure 11.3). This dialog provides some file operations on the preset file. It is possible to undo these operations by reloading the preset file from the hard drive. To be able to use the destructive edit actions, it is always necessary to unlock them.

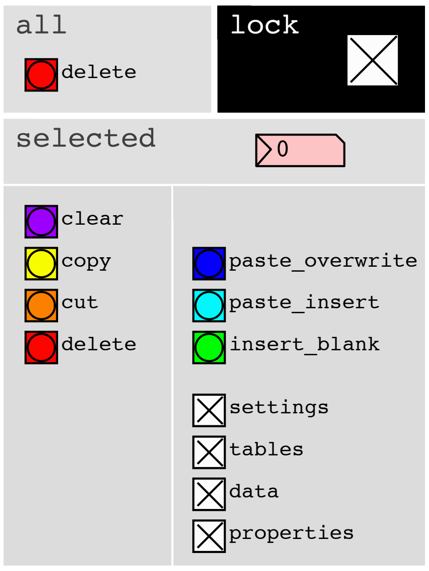


Figure 11.3.: Edit menu for `ds_gui`.

With exception of the delete function for the whole buffer in memory, all the editing actions aim on the currently selected scene, which can also be set in this dialog. It is also possible to specify, which storage layers should get pasted.

GUI element	**Address extension** `/ds/<domain>...`	**Description**
☒ *lock*	`/edit/lock`	Unlock the edit actions.
🟥 *delete all*	`/edit/clear`	Clear the whole memory.
🟪 *clear*	`/edit/scene/clear`	Clear contents of the selected scene.
🟧 *cut*	`/edit/scene/cut`	Cut selected scene to clipboard.
🟨 *copy*	`/edit/scene/copy`	Copy selected scene to clipboard.
🟥 *delete*	`/edit/scene/delete`	Delete selected scene.
🟦 *paste overwrite*	`/edit/scene/paste-overwrite`	Paste scene replacing the selected scene.
🟩 *insert blank*	`/edit/scene/insert`	Insert blank scene before the selected scene.
🟦 *paste insert*	`/edit/scene/paste-insert`	Insert scene from clipboard before the selected scene.
☒ *settings*	`/edit/scene/paste/settings`	Paste transition settings.
☒ *tables*	`/edit/scene/paste/tables`	Paste tables.
☒ *data*	`/edit/scene/paste/data`	Paste data.
☒ *properties*	`/edit/scene/paste/properties`	Paste scene properties.

11.2.2. ds_transition_gui

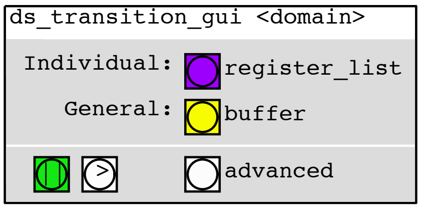


Figure 11.4.: GUI for ds_transition of domain <domain>.

ds_transition_gui (see Figure 11.4) provides the GUI for ds_transition. It gives access to all controls needed for editing scene transitions for the individual variables.

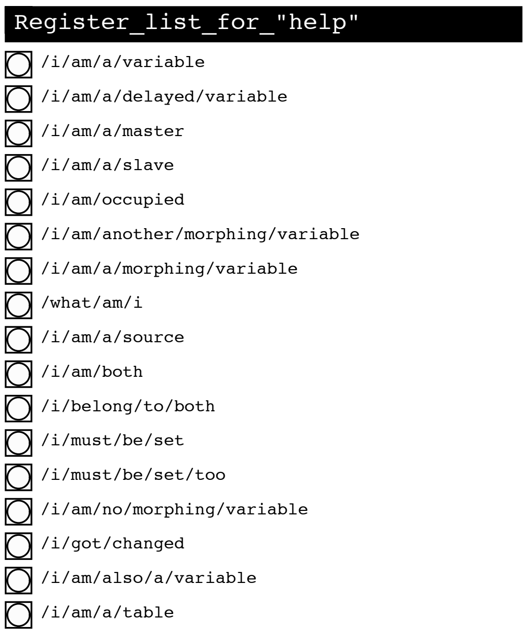
Main controls

GUI element	Address extension `/ds/<domain>...`	Description
🟪 *register list*	`/reg/dialog`	This button opens a list of all variables that are registered with this domain. Inside, click on a variable to get to its scene transition dialog (See 11.2.2 Register list).
🟨 *buffer*	`/reg/settings/buffer`	The register settings buffer acts as a clipboard for scene transition settings. The individual registers can copy their settings to it or take the contents of the buffer. The buffer can also be edited by hand.
🟩 *stop*	`/transition/stop`	Stop the current transition for the whole domain. Turns green (🟩) when a transition is finished, and orange (🟧) when an unfinished transition has been stopped.
⬜ *resume*	`/transition/resume`	Resume a stopped transition. Turns red (🟥) during an active transition.
⬜ *advanced*	`/transition/advanced/vis`	Open the advanced settings dialog (See 11.2.2 Advanced controls).

Register list / Scene transition dialog

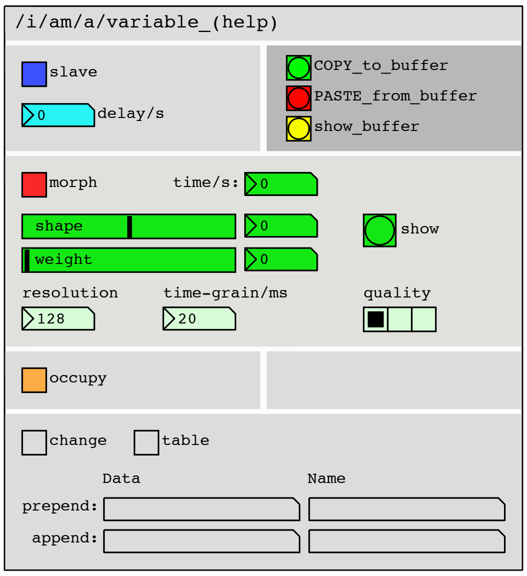
A click on the ● *register list* button opens a list of all variables, that are registered to this domain (See Figure 11.5).

Click on the button (◯) beneath the desired variable name to open its individual scene transition dialog (See Figure 11.6). The parameters in there are stored separately for each variable in every scene. They are saved in the *settings* layer of the scene-set.



```
Register_list_for_"help"
◯ /i/am/a/variable
◯ /i/am/a/delayed/variable
◯ /i/am/a/master
◯ /i/am/a/slave
◯ /i/am/occupied
◯ /i/am/another/morphing/variable
◯ /i/am/a/morphing/variable
◯ /what/am/i
◯ /i/am/a/source
◯ /i/am/both
◯ /i/belong/to/both
◯ /i/must/be/set
◯ /i/must/be/set/too
◯ /i/am/no/morphing/variable
◯ /i/got/changed
◯ /i/am/also/a/variable
◯ /i/am/a/table
```

Figure 11.5.: The register list for domain *help*.



```
/i/am/a/variable_(help)
slave                     COPY_to_buffer
0 delay/s                 PASTE_from_buffer
                          show_buffer
morph        time/s: 0
shape              0      show
weight             0
resolution   time-grain/ms    quality
128          20
occupy
change   table
           Data                 Name
prepend:
 append:
```

Figure 11.6.: Scene transition dialog for variable "/i/am/a/variable" of domain *help*

Scene transition dialog		
GUI element	**Address extension** `/ds/<domain>...`	**Description**
🟦 *slave*	`<name>/slave`	Start recalling only after another (master-) variable has completed its transition. (See 12.6.2 Slave).
▷0 *delay/s*	`<name>/delay`	To delay the recall of one variable, a time in seconds can be entered. (See 12.6.2 Delay).
🟥 *morph*	`<name>/morph`	Turn on *morph* to get a smooth fade from the current value to the value that is saved in the current scene. (See 12.6.2 Morph).
▷0 *time/s*	`<name>/morph/time`	Set the fade time in seconds. (See 12.6.2 Morph).
▷20 *time-grain/ms*	`<name>/morph/timegrain`	Set the data-rate of the fade in milliseconds. (See 12.6.2 Morph).
🟢 *show*	`<name>/morph/curve/vis`	Show the current appearance of the fade curve. (See 12.6.2 Morph).
▷0 *shape*	`<name>/morph/curve/shape`	*Shape* lets you chose a shape for the curve. (See 12.6.2 Morph).
▷0 *weight*	`<name>/morph/curve/weight`	*Weight* is a mix factor between a linear fade and the chosen shape. (See 12.6.2 Morph).
▷128 *resolution*	`<name>/morph/curve/resolution`	Specify the resolution of the fade curve. (See 12.6.2 Morph).
■□□ *quality*	`<name>/morph/curve/quality`	Choose the quality (interpolation type) in which the fade curve will be applied: none (default), linear, 4-point polynomial. (See 12.6.2 Morph).
🟧 *occupy*	`<name>/occupy`	If *occupy* is set, the variable will not respond to another scene recall while it is transitioning. (See 12.6.2 Occupy).

Copy&Paste (See 12.6.2 Copy transitions)		
GUI element	**Address extension** `/ds/<domain>...`	**Description**
🟢 *copy to buffer*	`<name>/settings/copy`	Copy all the settings from above to the register settings buffer.
🔴 *paste from buffer*	`<name>/settings/paste`	Paste the register settings buffer to this register.
🟡 *show buffer*	`/reg/settings/buffer`	Show the buffer window. (See 11.2.2 Buffer).

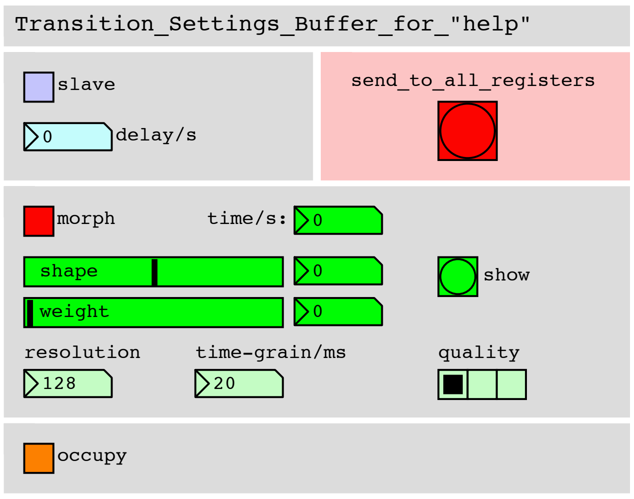
The following GUI elements only appear in this window for monitoring reasons. They need to be set as flags for the individual registers (see 11.1.4):

Flag monitoring		
GUI element	**Address extension** `/ds/<domain>...`	**Description**
☐ *change*	`<name>/change`	Shows, if the *change* flag is set. The message will then only get recalled if the new value differs from the old value. Even if the stored variable is a list or table.
☐ *table*	`<name>/table`	Shows if the *table* flag is set, i.e. the variable is a table.
▭ *data prepend*	`<name>/data_prepend`	Show if a symbol is prepended to the recalled data.
▭ *data append*	`<name>/data_append`	Show if a symbol is appended to the recalled data.
▭ *name prepend*	`<name>/name_prepend`	Show if a symbol is prepended to the variable name for recalls.
▭ *name append*	`<name>/name_append`	Show if a symbol is appended to the variable name for recalls.

Buffer

A click on the 🟡 *buffer* button opens a dialog for editing scene transitions for all registers together (See Figure 11.7).

The controls of the buffer are the same as in the individual registers. You can copy from the buffer to single registers or vice-versa in the individual scene transition dialogs.


Transition_Settings_Buffer_for_"help"

slave

delay/s 0

morph time/s: 0

shape 0

weight 0

resolution 128

time-grain/ms 20

Figure 11.7.: The scene transition buffer for domain *help*.

GUI element	Address extension `/ds/<domain>...`	Description
🟦 *slave*	`/reg/settings/buffer/slave`	See 11.2.2 Scene transition dialog
▷0 *delay/s*	`/reg/settings/buffer/delay`	See 11.2.2 Scene transition dialog
🟥 *morph*	`/reg/settings/buffer/morph`	See 11.2.2 Scene transition dialog
▷0 *time/s*	`/reg/settings/buffer/morph/time`	See 11.2.2 Scene transition dialog
▷20 *time-grain/ms*	`/reg/settings/buffer/morph/timegrain`	See 11.2.2 Scene transition dialog
🟢 *show*	`/reg/settings/buffer/morph/curve/vis`	See 11.2.2 Scene transition dialog
▷0 *shape*	`/reg/settings/buffer/morph/curve/shape`	See 11.2.2 Scene transition dialog
▷0 *weight*	`/reg/settings/buffer/morph/curve/weight`	See 11.2.2 Scene transition dialog
▷128 *resolu-tion*	`/reg/settings/buffer/morph/curve/resolution`	See 11.2.2 Scene transition dialog
■□□ *quality*	`/reg/settings/buffer/morph/curve/quality`	See 11.2.2 Scene transition dialog
🟧 *occupy*	`/reg/settings/buffer/occupy`	See 11.2.2 Scene transition dialog

GUI element	Address extension `/ds/<domain>...`	Description
🔴 *send to all registers*	`/reg/settings/buffer/paste/all`	Paste the buffer settings from above to all registers of the current domain. All variables will then have the same transition.

Advanced controls

In the advanced controls dialog (see Figure 11.8), some or all transition settings can be switched off globally for all variables. It opens by clicking on ○ *advanced* in `ds_transition_gui`.

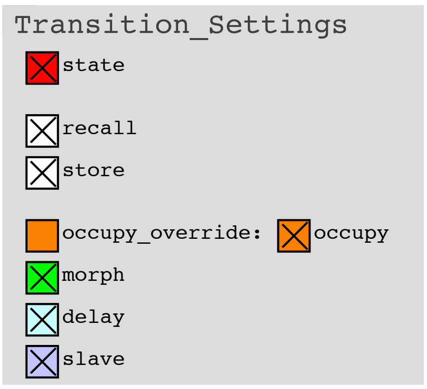
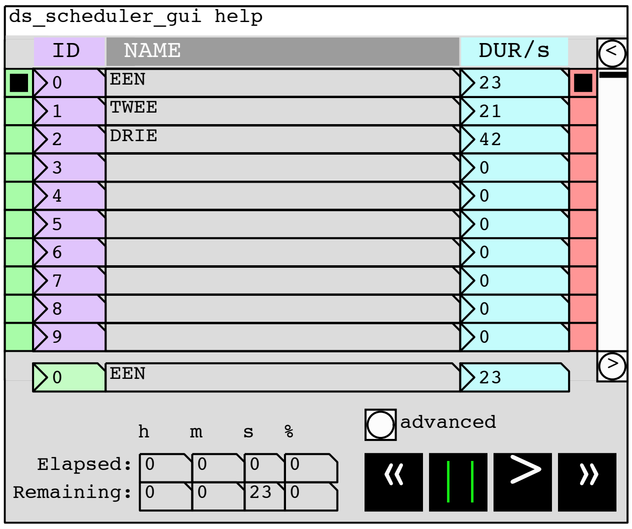


Figure 11.8.: The advanced controls for `ds_transition_gui`.

GUI element	Address extension /ds/<domain>...	Description
🟥 *state*	/transition/state	Turn scene transitions on and off globally.
☐ *recall*	/recall/settings/state	Recall transition settings on a scene recall.
☐ *store*	/store/settings/state	Store transition settings if a scene gets stored.
🟧 *occupy override*	/occupy/override	Override the individual state of *occupy* globally.
🟧 *occupy*	/occupy	Set *occupy* status when *occupy override* is active
🟩 *morph*	/morph	Turn parameter morphing on and off globally.
☐ *delay*	/delay	Turn delays of parameters on and off globally.
🟦 *slave*	/slave	Turn *slave* on and off globally.

11.2.3. ds_scheduler_gui



ds_scheduler_gui help

ID	NAME	DUR/s
0	EEN	23
1	TWEE	21
2	DRIE	42
3		0
4		0
5		0
6		0
7		0
8		0
9		0
0	EEN	23

advanced

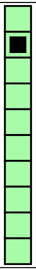
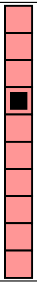
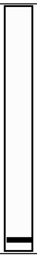
	h	m	s	%
Elapsed:	0	0	0	0
Remaining:	0	0	23	0

Figure 11.9.: [ds_scheduler_gui].

ds_scheduler_gui (see Figure 11.9) provides the GUI for ds_scheduler. It contains all the controls for time-line-editing and scheduling scenes in a play-list view.

Play-list

The play-list view shows all stored scenes and provides controls to browse through them. There are additional controls to set an individual duration and name for each scene.

GUI element	Address extension `/ds/<domain>...`	Description
	`/scene/current`	Marks the current scene (= cursor). Click inside to recall a specific scene.
4 *ID*	– – –	Shows the scene numbers.
NAME	– – –	Shows the scene names. Type in a scene name to set one.
3 *DUR/s*	– – –	Shows the scene durations in seconds. Type in a new duration to set one.
	`/scene/selected`	Marks the selected scene. Click inside to change it. Turns grey during an active transition.
previous	– – –	Scroll play-list backward by one slot.
	– – –	Scroll through play-list.
next	– – –	Scroll play-list forward by one slot.

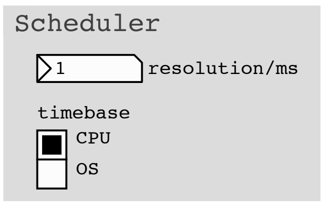
Transport

The transport section of `ds_scheduler_gui` provides controls for playback of the current scene-set.

GUI element	Address extension `/ds/<domain>...`	Description
▷9	`/scene/current`	Current scene.
▭ *NAME*	`/scene/current/name`	Name of the current scene. Type in a name to set it.
▷3 *DUR/s*	`/scene/current/duration`	Duration of the current scene in seconds. Type in a new duration to set it.
◯ *advanced*	`/scheduler/advanced/vis`	Open the advanced settings dialog. (See 11.2.3 Advanced).
« *backward*	`/backward`	Get to the previous scene.
‖ *pause*	`/pause`	Pause playback of the play-list. Colors show status of transition when paused: active, finished.
> *play*	`/play`	Start playback of the play-list. Colors show status of transition during playback: active, finished.
» *forward*	`/forward`	Get to the next scene.
0 *Elapsed h*	`/timer/elapsed/h`	Elapsed playback time of current scene: hours
0 *Elapsed m*	`/timer/elapsed/m`	Elapsed playback time of current scene: minutes
0 *Elapsed s*	`/timer/elapsed/s`	Elapsed playback time of current scene: seconds
0 *Elapsed %*	`/timer/elapsed/cs`	Elapsed playback time of current scene: centiseconds
0 *Remaining h*	`/timer/remaining/h`	Remaining playback time of current scene: hours
0 *Remaining m*	`/timer/remaining/m`	Remaining playback time of current scene: minutes
0 *Remaining s*	`/timer/remaining/s`	Remaining playback time of current scene: seconds
0 *Remaining %*	`/timer/remaining/cs`	Remaining playback time of current scene: centiseconds

Advanced

The advanced dialog (see Figure 11.10) is opened, when clicking on ◯ *advanced* in `ds_scheduler_gui`. There are some settings for the playback engine of `ds_scheduler`.



Scheduler

1 resolution/ms

timebase

■ CPU

□ OS

Figure 11.10.: The advanced controls for `ds_scheduler_gui`.

GUI element	Address extension /ds/<domain>...	Description
▷1 *resolution/ms*	/resolution	Set the time resolution of the playback engine in milliseconds.
■□ *timebase: CPU / OS*	/timebase	Set the timebase of the playback engine. **(0)** ***CPU***: Rely only on the CPU time, created by a `metro` object. This means, it will differ from the actual passed time, depending on the CPU load. **(1)** ***OS***: Rely only on the time calculated by the operating system through the `realtime` object.

12. Using Kollabs/DS

This chapter shows, how *Kollabs/DS* is used. Unfortunately, not all use-cases can be covered here, so some advanced options need to be figured out by looking into the file reference in Chapter 11.

12.1. Creating a storage domain

A storage domain is created by adding a `ds_logic` module to the patch. The first creation argument defines the name of the domain: `ds_logic <domain>`. Additionally there are some *flags* (see 10.3), that can be set starting with the second creation argument (See 11.1.1 for more information).

12.2. Registering variables

For each variable that should be registered to a storage domain, an individual register needs to be created.

12.2.1. General

New variables can be registered to a storage domain with the `ds_reg` module. Variables can be any type of messages (*list*, *float*, *symbol*), or even tables. Nevertheless, the morphing features are only available for *float* values.

EXAMPLE: `ds_reg foo bar` registers variable *bar* to domain *foo* (See 11.1.4).

In this context, variable *bar* can be created either through a pair of `send bar` and `receive bar` or by a GUI object with identical send and receive name (See 9.1).

12.2.2. Tables

To register a table to the storage, the *table* flag needs to be added to `ds_reg` after the two creation arguments: `ds_reg <domain> <name> table`. If the table should be resized automatically to the new size, an additional *resize* flag can be set. In contrast, *noresize* specifies, that the table should never be resized automatically, which is the default behavior anyway.

EXAMPLE: `table baz` creates a table. `ds_reg foo baz table noresize` creates a corresponding register with automatic resizing explicitly disabled. The order of the two flags *table* and *noresize* does not matter.

For tables, the morphing functionality is deactivated, but other scene transition features like *delay* or *slave* can be used.

12.2.3. Global variables

There is also the possibility to register a global variable. This means, its state stays the same for all scenes and is saved and loaded with the scene-set. Global variables can only be of type float.

There is a special register object for global variables: `ds_reg_global`. It works just like `ds_reg`, but there are no flags available for `ds_reg_global`.

Global variables can be accessed just like normal variables through sends and receives, but in addition, there is the possibility to communicate with them through the `value` object (See Figures 12.1 and 12.2).

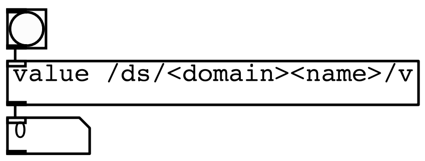


Figure 12.1.: Get the status of a global variable through the `value` object.

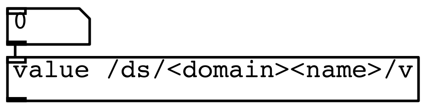


Figure 12.2.: Change the state of a global variable through the `value` object.

12.3. Store/Recall, Save/Load

The most important thing in *Kollabs/DS* is saving scene-sets. For this, there are some semantics to be explained. Single scenes can be *stored* to memory and *recalled* from there. The memory is non-permanent, which means it will be lost, if the patch is closed. Therefore the whole scene-set in memory must be *saved* to a permanent text file on the hard drive, from where it can also be *loaded* again.

12.3.1. Store

To store the current status of all registered variables to a scene, select a target scene in `ds_gui` (see Figure 11.1), either directly ▷26 or through < *decrement* / > *increment*.

Then ● *store* the scene to memory.

12.3.2. Recall

A specific scene can then be recalled from memory by either entering its scene-number ▷9 or through the < *decrement* and > *increment* buttons. If the current scene number gets changed, the according is recalled automatically. The current scene can be recalled again by clicking on 🟩 *recall*.

12.3.3. Save

The whole scene-set with all scenes and global variables can be saved to a text file on the hard drive. 🟥 *save* opens a save dialog, where a path and filename can be specified. If the scene-set has already been saved or loaded during the current session, it can be re-saved to the last specified path via 🟥 *resave*.

12.3.4. Load

To load a previously saved scene-set from the hard disk to memory, click on the 🟩 *load* button to enter the load dialog. If the scene-set has already been saved or loaded during the current session, the last specified path can be re-loaded via 🟩 *reload*.

12.4. Advanced scene editing

The ○ *edit* button in `ds_gui` opens a dialog for advanced file operations. As these could destroy much data if not used carefully, they need to get unlocked through the ☒ *lock* switch.

It is highly recommended to 🟥 *save* / 🟥 *resave* the scene-set to the hard disk before editing, so that it is always possible to undo an unwanted action through a 🟩 *reload*.

First, select the scene on which you want to perform an editing action via `26` *selected scene*. Then perform one of the actions, which are explained in the following.

12.4.1. Erase scene data

The data of a scene can be erased in two ways: *clear* wipes out all data in the selected scene, but leaves the empty scene at its place. *delete* also deletes the scene itself. The following scenes will then succeed by one slot to fill the gap.

To wipe out the whole scene-set completely, which means erase all scenes, click on *delete all*.

12.4.2. Copy/Cut

The selected scene can be copied to the clipboard via *copy*. In the *advanced* dialog of `ds_gui`, there is the possibility to print the current clipboard contents to the *Pd*-window through the *clipboard* button.

To additionally delete the selected scene, there is also a *cut* command. The following scenes will then succeed by one slot to fill the gap.

12.4.3. Paste

The clipboard can be pasted in two different ways: Either by replacing the selected scene (*paste overwrite*), or by creating a new scene and shifting the selected and following scenes upwards by one slot (*paste insert*).

It can be specified, which layer of the scene in the clipboard should be pasted, leaving the rest of the selected scene untouched. This could be useful, if only the transition settings are to be replaced, but not the actual data. De-select the layers which should be skipped on a paste: ☒ *settings*, ☒ *data*, ☒ *tables*, ☒ *properties*.

12.4.4. Insert blank scene

To insert a new blank scene, click on *insert blank*. The eventually already existent scene at the selected slot and the following ones will succeed by one slot.

12.5. Working with multiple domains

It is possible to have multiple storage domains at the same time, by creating additional `ds_logic` modules with different identifiers. The different domains can either be used side-by-side, with no influence to each other, or also connected for special applications.

12.5.1. Multiple domains side-by-side

Any number of domains can be created side-by-side without influencing each other.

Create first storage domain: `ds_logic domain1`

Create second storage domain: `ds_logic domain2`

Register *variable1* to *domain1*: `ds_reg domain1 variable1`

Register *variable2* to *domain2*: `ds_reg domain2 variable2`

12.5.2. Overlapping domains

It is also possible to register a variable to more than one domain at the same time. Values and transitions will be saved independently. Anyway, most of the time you probably don't want a variable to be part of multiple domains, as you might get in big trouble.

Create first storage domain: `ds_logic domain1`

Create second storage domain: `ds_logic domain2`

Register *variable1* to *domain1*: `ds_reg domain1 variable1`

Register the same *variable1* to *domain2*: `ds_reg domain2 variable1`

12.5.3. Nested domains

Sometimes it might be useful to nest one storage domain inside another. This way, different presets can be created in a slave domain and arranged to a play-list by a master domain. This is done by registering the current scene of the slave domain to the master domain. The send/receive name of the current scene can be looked up in the file reference of `ds_gui` in 11.2.1 Main controls.

Create a *master* domain: `ds_logic masterdomain`

Create a *slave* domain: `ds_logic slavedomain`

Register current scene of *slavedomain* to *masterdomain*:
`ds_reg masterdomain /ds/slavedomain/scene/current`

Note, that only scene numbers of the *slave* domain will get stored in the *master* domain. The actual data stays in the *slave* domain.

12.6. Scene transitions

12.6.1. General work-flow

Scene transitions are only activated, if a `ds_transition` module has been created for the particular domain: `ds_transition <domain>`.

The scene transition settings are always stored with the destination scene. To add a transition, first recall the scene, in which the transition should end, and wait until it is completely recalled (the ○ *stop* button in `ds_transition_gui` turns green: 🟢).

HINT: To save time while editing, skip already programmed transitions: open the ○ *advanced* transition settings and turn off the global 🟥 *state* before recalling. This way, many transitions can be programmed efficiently without waiting for them to be done.

It is very important to store the scene after editing a transition. For example through 🔴 *store* in `ds_gui`.

12.6.2. Scene transition dialog

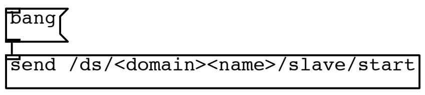
To edit the scene transition in a single variable, click on 🟣 *register list* in `ds_transition_gui`. This opens a list of all variables that are currently registered to this domain (See Figure 11.5). Choose the desired variable by clicking on the corresponding ○ `<name>` to open its scene transition dialog (See Figure 11.6).

If the scene transition should be the same for all registered variables, click on 🟡 *buffer* to open a general scene transition dialog. It has the same settings as the individual registers and can be copied to all of them through 🔴 *send to all registers*. The settings in the buffer are the same as in the single variables.

Slave

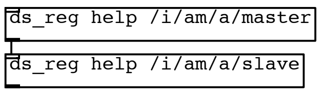
The 🟦 *slave* option makes it possible to chain the recalls of single variables, to be sure that dependent variables are recalled in the right order. If *slave* is set, the variable will wait for an external trigger, that either arrives as a *bang* through the first inlet of `ds_reg` (see Figure 12.3), or is sent to `"/ds/<domain><name>/slave/start"`. `ds_reg` also outputs a trigger signal after each finished scene transition, so that multiple registers can be chained (See Figures 12.4 and 12.5).

Each variable also sends a bang to `"/ds/<domain><name>/dispatched"`, when its transition has finished.



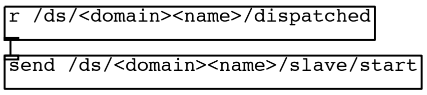
```
bang
send /ds/<domain><name>/slave/start
```

Figure 12.3.: Trigger a *slave* variable through a bang.



```
ds_reg help /i/am/a/master
ds_reg help /i/am/a/slave
```

Figure 12.4.: Chain registers by wire.



```
r /ds/<domain><name>/dispatched
send /ds/<domain><name>/slave/start
```

Figure 12.5.: Chain registers by send/receive.

Delay

To delay the recall of a variable, set a constant ▷0 *delay* time in seconds.

Morph

The 🟥 *morph* toggle enables a fade from another scene to the current scene for this variable.

The morph time can be set in seconds: ▷0 *time/s*.

To apply a **linear fade**, set the *weight* parameter to zero:

weight ▷0 (0...1)

With *weight* = 0, the parameters shape ▷0 *shape*, ▷128 *resolution* and ■□□ *quality* have no effect.

For a **curved fade**, click on ● *show* to open a graphic representation of the curve (see Figure 12.6).

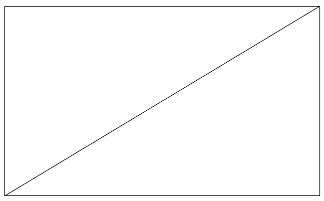


Figure 12.6.: Fade curve for a linear fade (*weight*=0).

Set *weight* to "1" and edit the *shape* parameter:

shape ▷0 (-Inf...+Inf)

A value of "0" produces a half sine wave (see Figure 12.7), "1" a quarter sine wave (see Figure 12.8), and "2" leads to a x^2 function (see Figure 12.9). Float values in between produce a linear blend between these three shapes (see Figure 12.11). For values greater than "2", an exponential x^{weight} function is applied (see Figure 12.10). A negative sign inverts the shape (see Figure 12.12).

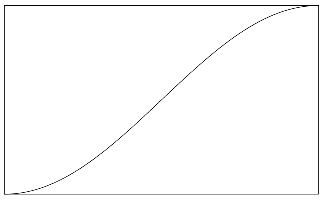


Figure 12.7.: Fade curve for a half sine fade (*weight*=1, *shape*=0).

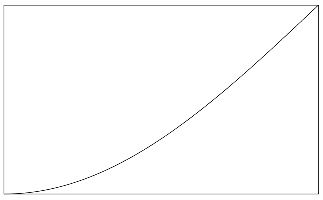


Figure 12.8.: Fade curve for a quarter sine fade (*weight*=1, *shape*=1).

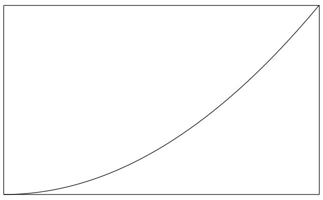


Figure 12.9.: Fade curve for a quadratic x^2 fade (*weight*=1, *shape*=2).

▷128 *resolution* sets the resolution of the fade curve.

■☐☐ *quality* sets how the curve is read:

(0) ■☐☐ No interpolation. Jumps to nearest value.
(1) ■☐☐ Linear (2-point) interpolation between two values.

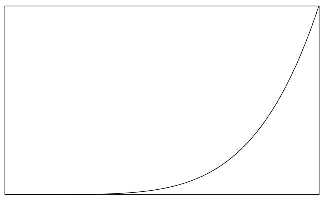


Figure 12.10.: Fade curve for an exponential x^{weight} fade (here: *weight*=1, *shape*=5).

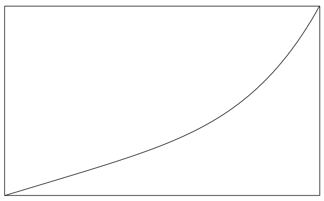


Figure 12.11.: Fade curve for a weighted exponential x^{weight} fade (here: *weight*=0.5, *shape*=5).

(2) ■□□ 4-point polynomial interpolation.

Hint: for MIDI values, a *resolution* of 128 (7 bit) with no interpolation (*quality*=0) is sufficient.

The sample-rate of the fade can be set through the *timegrain* parameter. The default value is 20 milliseconds.

▷20 *timegrain/ms*

Occupy

If 🟧 *occupy* is set, the variable will not respond to another scene recall until its transition is completed. By default, this option is deactivated.

Copy transitions

After editing the scene transition settings of one variable, you can 🟩 *copy* them to the buffer. You can also open this buffer through 🟨 *show buffer*, and edit the

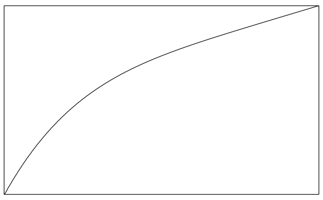


Figure 12.12.: Fade curve for an inverted and weighted exponential x^{weight} fade (here: *weight*=0.5, *shape*=-5).

transition settings there. The buffer can then either be pasted into single variables via 🔴 *paste* inside the individual scene transition dialogs, or into all variables through the buffer window: 🔴 *paste to all registers*.

12.6.3. Stop an active transition

If a scene transition is currently going on, the ○ *resume* button in `ds_transition_gui` turns red: 🔴. When the transition has finished, the ○ *stop* button turns green: 🟢. An active transition can be stopped by clicking on ○ *stop* and continued any time by clicking on ○ *resume*.

12.7. Time-line editing / Play-lists

To activate time-line-editing, a `ds_scheduler` module and the according GUI `ds_scheduler_gui` must be created for the specific storage domain:

```
ds_scheduler <domain>
```

```
ds_scheduler_gui <domain>
```

The play-list view in `ds_transition_gui` allows browsing through the scenes. Each scene can be given an individual ▭ *name* and ▷3 *duration*. It is important to note, that the durations include eventual scene-transitions of the variables.

The ▶ *play* button allows playback of the scene-set starting at the current scene. After the given duration, the next scene gets recalled. If the duration of a scene is set to zero, the playback will pause until you start it again. This way, manual cues can be set inside a pre-programmed show.

The ⏸ *pause* button will pause playback at the given time. If a transition is currently going on, it will be stopped too, and resumed again, if playback is resumed.

During playback, scenes can be skipped or started again through » *forward* and « *backward*. When the playback is paused, these controls have the same effect as ⧁ *next* and ⧀ *prev* in `ds_gui`.

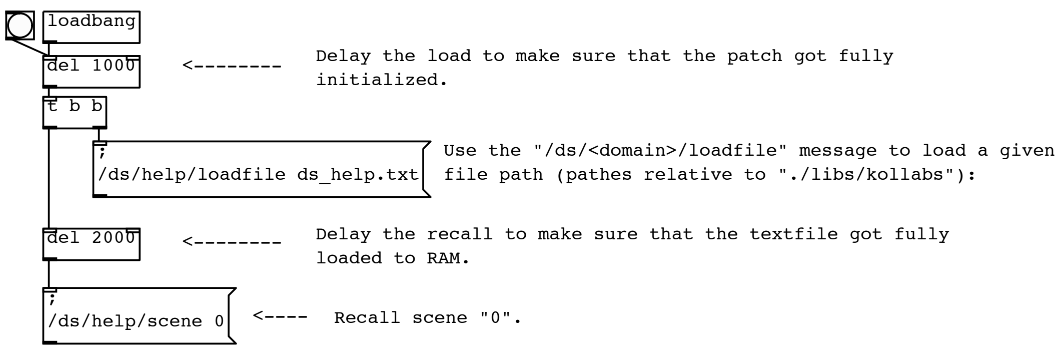
It is possible to set the time resolution of the playback engine in milliseconds through ▷1 *resolution/ms*, as well as the ■□ *timebase*. There you can choose between two time sources for the playback engine:

■□ (*CPU*) The time measured by the CPU of the computer.
□■ (*OS*) The time calculated by the operating system.

For most cases, *OS* is the right choice.

12.8. Load file on startup

In some situations it is required, that the patch automatically loads a scene-set and recalls a specific scene on startup. Send a message consisting of the path and filename to `"/ds/<domain>/loadfile"`, to load a scene-set from hard disk to memory (See Figure 12.13). The path must be given relative to *Kollabs*. This means, if *Kollabs* lies in `"/path/to/patch/libs/kollabs"`, and the scene-sets are saved in `"/path/to/patch/data"`, then the path must be set as `"../../data/<filename>"`. Afterwards recall the required scene by setting `"/ds/<domain>/scene/current"` through a message. A delay should be introduced to make sure, the scene-set is fully loaded and the patch is initialized before recalling.



```
loadbang
del 1000        <---------    Delay the load to make sure that the patch got fully
                              initialized.
t b b
  ;
  /ds/help/loadfile ds_help.txt   Use the "/ds/<domain>/loadfile" message to load a given
                                  file path (pathes relative to "./libs/kollabs"):
del 2000        <---------    Delay the recall to make sure that the textfile got fully
                              loaded to RAM.
;
/ds/help/scene 0   <----   Recall scene "0".
```

Figure 12.13.: Example: Load file on startup and recall scene.

12.9. Additional tools

There are many additional tools built in `ds_reg`, that are used for some of the internal functions. Some of them are also worth being used independently for special tasks.

12.9.1. Route current value of variable

The current value of a registered variable can be sent to any destination by a route command:

Sending a message `"<target>"` to `"/ds/<domain>/<name>/route"`, sends the current value of `"<name>"` to `"<target>"`.

This may be useful for implementing copy & paste functionality outside of *Kollabs/DS*.

It is also possible to route the current values of all variables in a domain to one destination through a global route command:

Sending a message `"<target>"` to `"/ds/<domain>/route"` sends the current values of all registered variables of `"<domain>"` to `"<target>"`.

This functionality is already built into `ds_reg`, but can also be created for variables, that are not registered to a storage domain, through the `ds_route <name>` object.

12.9.2. Update / re-send current value

If the target for a route command should be the variable itself, there is a shortcut available. It re-sends the current value of a registered variable to itself, which means update its status:

Sending any message to `"/ds/<domain>/<name>/resend"` will send the current value of `"<name>"` to itself.

This may be useful for updating GUI elements.

A global re-send for the whole domain can be ordered by a global resend command:

Sending any message to `"/ds/<domain>/resend"` sends the current values of all registered variables of `<domain>` to themselves.

This functionality is already built into `ds_reg`, but can also be created for variables, that are not registered to a storage domain, through the `ds_route <name>` object.

12.9.3. Print current value to the Pd window

`ds_reg` also provides the ability to print the current value of single variables to the *Pd* window:

Sending a *bang* to `"/ds/<domain>/<name>/print"` will print the current value of `"<name>"` to the *Pd* window.

This is sometimes useful for debugging. Single parameters can also be printed via the advanced settings of `ds_gui` (see 11.2.1 Advanced Controls).

Part III.

Discussion

13. Using Kollabs/DS in real-world scenarios

Since its first implementation for *Extended View Toolkit* in 2010 (see 13.1), *Kollabs/DS* has been used in various shows, installations and workshops. With each application, the system has been refined to meet new requirements and eliminate failures.

13.1. Extended View Toolkit

An early state-saving system had been developed in 2010 for the immersive media installation *Extended View Streamed* at *kunsthaus muerz*[1]. The custom projection software evolved further to the open-source project *Extended View Toolkit* (See Fig. 13.1 and Venus and Weger, 2010). Since the first days, it has been closely connected to *Kollabs/DS*.

The usage of *Extended View Toolkit* and *Kollabs/DS* has been taught in in several workshops at universities and conferences, such as *Joanneum University of Applied Sciences Graz*[2] (2010, 2011, 2013[3]), *Pure Data Convention Weimar-Berlin*[4] (2011), *Zurich University of the Arts*[5] (2011), *WORM Rotterdam*[6] (2012), *University*

[1] *kunsthaus muerz*: http://www.kunsthausmuerz.at/

[2] *FH Joanneum Graz*: http://www.fh-joanneum.at/?lan=en

[3] http://ip2013.laras.be/planning-2013/

[4] *PD Convention Weimar-Berlin 2011*: http://www.uni-weimar.de/medien/wiki/PDCON:Conference/Extended_View_Toolkit

[5] *Zurich University of the Arts*: https://www.zhdk.ch/

[6] *WORM Rotterdam*: http://www.worm.org/home/view/event/1873

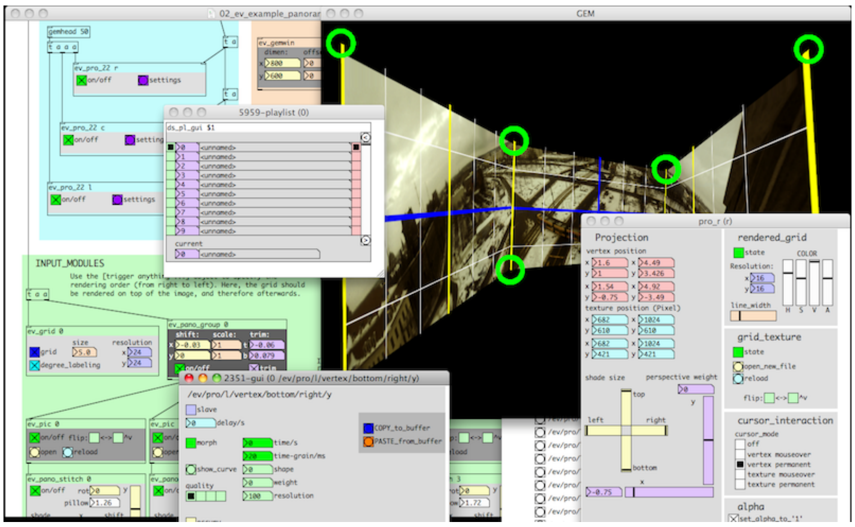


Figure 13.1.: *Extended View Toolkit*. Video: http://vimeo.com/51567993

of Florida - Digital Worlds Institute[7] (2012) and *Linux Audio Conference Graz*[8] (2013).

13.2. Monster

In 2012, the *Kollabs/DS* state-saving system was used for the mixed-media performance *Monster* (See Fig. 13.2). The project was a collaboration with composer Wen Liu and was performed several times in Vienna (*Semperdepot*[9], *Porgy & Bess*[10], *Odeon Theater*[11]) and was featured in the opening concert of the *International Computer Music Conference 2012*[12] in Ljubljana.

As a starting point for quick calibration of the projection mapping, different presets

[7] *University of Florida - Digital Worlds Institute*: http://www.digitalworlds.ufl.edu/
[8] *Linux Audio Conference Graz 2013*: http://lac.linuxaudio.org/2013/program
[9] *Semperdepot / Academy of Fine Arts Vienna*: http://www.akbild.ac.at/
[10] *Porgy & Bess*, Vienna: http://www.porgy.at/
[11] *Odeon Theater*, Vienna: http://www.odeon-theater.at/
[12] *ICMC 2012, Ljubljana*: http://www.icmc2012.si/

needed to be saved and loaded. Detailed information about the project and its artistic and technical realization can be obtained in my bachelor thesis (Weger, 2012).

Figure 13.2.: Monster. Video: `vimeo.com/weger/monster-short-trailer`

13.3. In Caelum

In Caelum (see Venus and Weger, 2012 and Fig. 13.3) is an immersive media installation, which features a panoramic projection environment and ambisonic sound. It is based around a 360 degree panoramic video camera system and data from satellites, transmitted in realtime to control the installation. The installation is inspired by the fact, that a lot of our daily communication is handled via satellites, which are invisible and mostly unnoticed in the sky above us. *In Caelum* observes the sky above us, visualizes and brings them to life acoustically.

Kollabs/DS has been used to store all settings of the projection mapping, sound synthesis, etc. to automatically start with the computer every day of the exhibition for one month.

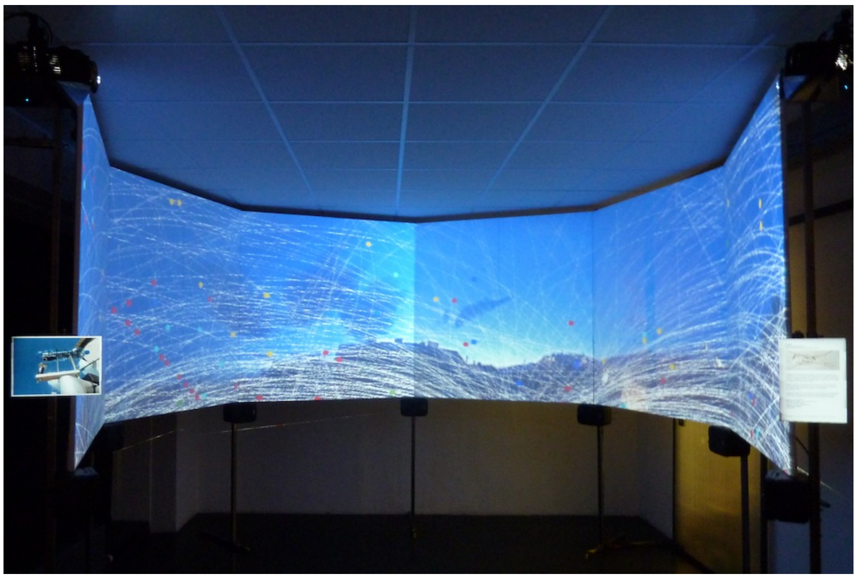


Figure 13.3.: In Caelum. Video: `http://vimeo.com/weger/incaelum`

13.4. Orpheus & Eurydice

In the dance theater project *Orpheus & Eurydice* (see Horvath, 2013 and Fig. 13.4), the versatile scene-morphing functionality of *Kollabs/DS* has been used the first time in a bigger show. All parameters of the interactive sound design, realized by two computers (one running *Mac OS X* and one running *Linux*) were managed by *Kollabs/DS*(See left display in Fig. 13.4).

The video projections were produced by Peter Venus, who also made extensive use of *Kollabs/DS*. He worked with a computer running *Linux*, *Pd* and *Extended View Toolkit* (See right display in Fig. 13.4. Scene transitions with very long fade times (several minutes) were used to blend in video projections smoothly and precise. The ability to trigger these transitions through one button proved to be practical and error-safe.

As this was the first time, that very long transition times were used, it was never before thought of a possibility for pausing and resuming transitions. This project lead to the implementation of these advanced transition controls, which proved to be essential for theater productions.

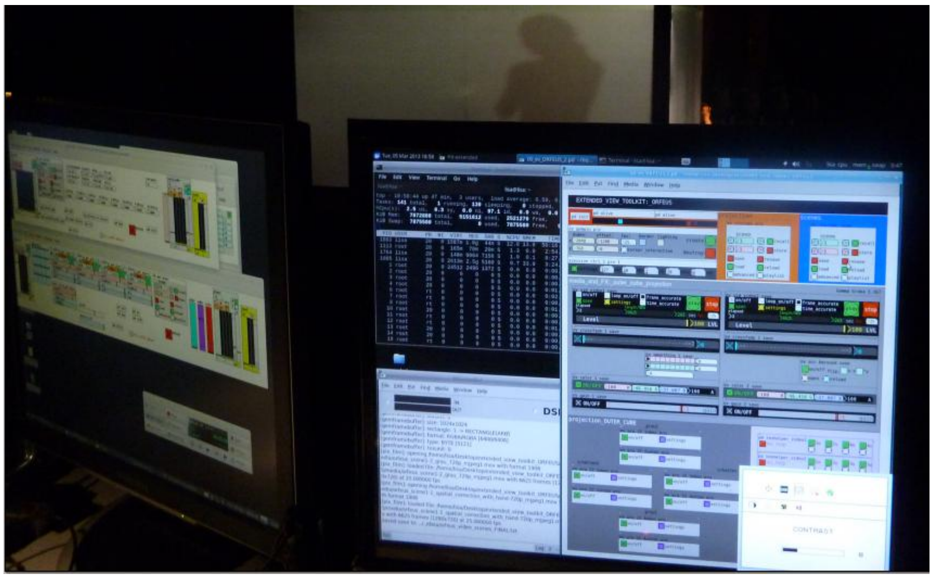


Figure 13.4.: *Orpheus & Eurydice*. Video: http://vimeo.com/78349033

13.5. 3x cosi fan tutte

The research project *3x cosi fan tutte* in 2013 (see *3x Cosi fan tutte* 2012) was planned to be another testing environment for *Kollabs/DS*. The project involved three individual performances of the opera "Cosi fan tutte" by Wolfgang Amadeus Mozart, each with a completely different team. The productions were performed in *MUMUTH Graz*[13] and *Deutsche Oper Berlin*[14]

One version was produced by Michael von zur Mühlen and Christoph Ernst. Peter Venus contributed the video projection and sound design, all realized with a computer system running *Linux*, *Pure Data* and *Extended View Toolkit*. Peter Venus made extensive use of the *Kollabs/DS* state-saving system to trigger scenes both for video and sound. Unfortunately, the *Pd*-powered video projection setup was omitted last-minute due to artistic decisions of the producer, despite of the software's stable functionality.

[13] *MUMUTH Graz*: http://www.mumuth.at
[14] *Deutsche Oper Berlin*: www.deutscheoperberlin.de/

I contributed the sound design to the second team, lead by Margo Zalite and Martin Miotk. Unfortunately, I had to drop my prepared *Pd*-patch for realtime sound processing due to artistic decisions and needed to switch to *Ableton Live*[15], which provided faster results for this special purpose. This way, only the very basic functionality of *Kollabs/DS* was used for spatialization of *Ableton Live's* output through *Pd*.

13.6. Der Druckauftrag

In the theater play *Der Druckauftrag*[16] by the Graz-based group *Zweite Liga für Kunst und Kultur*, *Kollabs/DS* had been used as part of *Extended View Toolkit*.

The display of a laptop computer on stage was mirrored to the external VGA output and then sent to a video capture card in the projection server. The server was running Linux and *Extended View Toolkit* to project the computer screen contents onto a big projection screen above the stage. *Extended View Toolkit* was used to cut off the task-bar and also calibrate the projection mapping and color. Scenes were used to switch between different screen modes (i.e. full-screen video playback, text mode) during the performance.

[15] *Ableton Live*: https://www.ableton.com/en/live/
[16] *Der Druckauftrag*: http://zweiteliga.weblog.mur.at/?page_id=506

14. Kollabs in the future

In the past, *Kollabs* has proved to be a stable and useful tool, but it is still far away from being a finished product. At the moment, the state-saving system *Kollabs/DS* is well documented, but there still many features planned, which need to be implemented. The other parts of *Kollabs*, such as *OSC* and *MIDI* communication, still lack a proper documentation or even need to be implemented.

14.1. Planned features for *Kollabs/DS*

There are still many features on the wish list, which need to be implemented in the state-saving solution *Kollabs/DS*. Some of them are discussed in this section.

14.1.1. Recall mix of multiple scenes

For some use-cases, it would be nice to have the possibility to not only morph between two presets, but load a mix of both. Right now, there is the workaround to do a morph between the scenes, stop it at the desired point and then store this mixture to a new scene. An aim would be to recall a mixture of an arbitrary number of scenes by giving exact percentages of them.

EXAMPLE:

Sending a message `"0.3 5 0.6 6 0.1 2"` to `"/ds/<domain>/scene/current"` will recall a mixture of the three scenes 2, 5 and 6, with specific factors: 0.3 times scene 5, 0.6 times scene 6 and 0.1 times scene 2.

14.1.2. Load and draw user fade curves

It is a shame, that tables can be stored with the state-saving system, but there is no way to save individual fade curves for scene morphing. This is because the storage of tables is a feature, which was implemented after the whole system of scene morphing was done. To load and draw user fade curves, the whole system of scene morphing must be rethought.

14.1.3. Morph between lists and tables

Morphing between lists is no primary goal, because they can easily be split up into float values, if scene-morphing is required. However, as tables in *Kollabs* are treated just like lists, this would additionally lead to the ability of morphing between tables.

The implementation of this would demand splitting of lists with arbitrary length and dynamic creation of an individual morph object for every list item. A problem arises, when the list changes its size between two scenes, or contains symbols, which can not be morphed in a meaningful way.

14.1.4. Manual cross-fade between scenes

During the presentation of this work, it came out, that I omitted one of the most cool features of lighting consoles: To have a big cross-fader for manually fading between two scenes. A possible work-flow could be as follows:

1. Select the next scene without recalling it
2. Use a fader to control the manual transition to that scene.

14.2. Extensions of the *Kollabs* library

By now, *Kollabs/DS* has made much progress, which means the other parts of the *Kollabs* library need to catch up.

14.2.1. *OSC, MIDI, DMX*

OSC and *MIDI* are already implemented very well, but need to be reworked and documented. *DMX* is not even implemented yet, but as it works similar to *MIDI*, the *MIDI* implementation could be adapted very easily. It is planned to integrate a learn-mode into *Kollabs*, which makes it possible to map between *OSC*, *MIDI* and *DMX* in a very fast and intuitive way.

14.2.2. Bank management for control surfaces

The conjunction of *Kollabs/DS*and *Kollabs/MIDI* is planned to be used for advanced bank management for controllers with motorized faders or knobs with led-ring, such as the *Behringer BCF2000*[1] controllers (See Fig. 14.1). These controllers could integrate better with the software, if the whole bank-management would be computed and stored together with the individual program. This would mean, always the same bank in the controller is used, but different pages can be switched in the computer.

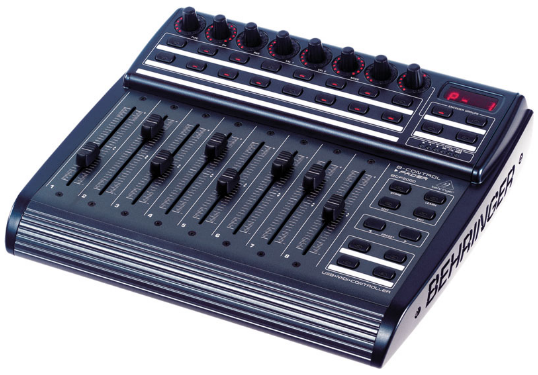


Figure 14.1.: The *Behringer BCF2000* DAW controller.

[1] *Behringer BCF2000*: `http://www.behringer.com/EN/Products/BCF2000.aspx`

15. Conclusion

With *Kollabs/DS*, a versatile tool for scene-based state-saving has been developed. While there are many features still waiting for implementation, *Kollabs/DS* is already more powerful than any other available state-saving solution for *Pure Data*. It has been proved in many cases, that *Kollabs/DS* is a reliable tool for shows and installations.

Anyway, like many open-source software projects, it will probably never be finished completely. As this project aims on a universal solution, which suits many different tasks, I am dependent on feedback of users to cover all aspects and eliminate critical bugs.

This document shows only the current state at the time of writing. To stay up to date, the following information channels can be used:

GitHub repository (main project host): `https://github.com/m---w/kollabs`

Project page on the *Pure Data* homepage: `http://puredata.info/downloads/kollabs`

For further information, do not hesitate to contact me under mail@marianweger.com

Appendix

Bibliography

3x Cosi fan tutte (2012). URL: http://www.kug.ac.at/news-veranstaltungen/news/kug-aktuell/details/article/3-x-cosi-fan-tutte.html (cit. on p. 77).

Barknecht, Frank (2008). *sssad - Stupidsupersimplistic State Saving ADVANCED*. URL: http://puredata.info/downloads/sssad (cit. on p. 8).

Bukvic, Ivica Ico. *L2Ork*. URL: http://l2ork.music.vt.edu/main/ (cit. on p. 9).

Horvath, Lisa (2013). *Orpheus and Eurydike*. URL: http://www.lisahorvath.at/orpheus-eurydike/ (cit. on p. 76).

Moser-Booth, Mike (2011). *mmb library*. URL: https://github.com/dotmmb/mmb (cit. on p. 9).

Salzberg, Jeffrey E. and Judy Kupferman (2013). *Stage Lighting Primer*. URL: http://www.stagelightingprimer.com (cit. on p. 7).

Venus, Peter and Marian Weger (2010). *Extended View Toolkit*. URL: http://extendedview.mur.at (cit. on p. 73).

Venus, Peter and Marian Weger (2012). *In Caelum*. URL: http://marianweger.com/projects/in_caelum.shtml (cit. on p. 75).

Weger, Marian (2012). "Monster - Ein interaktives Projektions-System für bewegte Objekte bei Tanzperformances." URL: http://iem.kug.ac.at/projects/workspace/2012/monster-ein-interaktives-projektions-system-fuer-bewegte-objekte-bei-tanzperformances.html (cit. on p. 75).