A Flexible Tool for the Visualization and Manipulation of Musical Mapping Networks

Aaron Henry Krajeski

Department of Music Technology
Schulich School of Music, McGill University
Montreal, Canada

August 2013

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Arts.

© 2013 Aaron Henry Krajeski

2013/08/02

Abstract

This thesis project presents MapperGUI, a cross-platform graphical tool for the manipulation of musical mapping networks. Most digital musical instruments (DMIs) gather gestural input from musicians by way of electronic sensors and transform these data into sound through separate synthesis engines. The mapping of control inputs to synthesis parameters in DMIs is arbitrary, multi-faceted and extremely important as to their effectiveness. Software mapping tools exist to aid in this process, attempting to make the task of musical mapping more transparent, swift and configurable.

The libmapper software library, developed at the Input Devices and Music Interaction Laboratory, creates a standard framework for DMIs to communicate data on a distributed network and to map their signals collaboratively and in real-time. MapperGUI presents a graphical user interface for libmapper networks, allowing non-expert users to manipulate the text-based system. The interface aims to be flexible, such that it can accommodate the vast array of musical networks and tasks that must be performed when mapping. To this end it provides multiple, independent visualizations and interaction modes within a single framework.

This document explores some of the issues challenging the field of musical mapping, and describes the motivations behind the MapperGUI project in this context. Relevant research in the fields of data visualization and interface design is summarized and applied to the task of creating a graphical user interface for libmapper networks. Prior graphical interfaces for libmapper are examined for successful features that can be incorporated into MapperGUI. Specific challenges of implementation and features of the final program are described. Insights gained from interviews with users of MapperGUI are presented, along with future work and possible extensions for the interface.

MapperGUI is available for free download as a standalone application at `www.libmapper.org/downloads`. All code is open-source and can be accessed at `https://github.com/mysteryDate/webmapper`.

Résumé

Acknowledgments

name	role
Marcelo Wanderley	Advisor, IDMIL
Joseph Malloch	libmapper, Maxmapper
Stephen Sinclair	libmapper
Vijay Ruraraju	Vizmapper
Jon Wilansky	GridView, HiveView
Håkon Knutzen, Mailis Rodrigues, Clayton Mamedes, Julie René	Feedback

l l

Contents

1 Introduction & Motivation **1**
1.1 Context and Motivation 2
1.2 Project Overview 4
1.3 Thesis Overview 5
1.4 Contributions 5

2 Background **6**
2.1 Mapping 6
2.1.1 Mapping theory 7
2.1.2 Mapping for digital musical instruments 11
2.2 Data Visualization and User Interface Design 14
2.2.1 Graphical dimensions 14
2.2.2 Relevant visualization techniques and systems 16
2.2.3 The model-view-controller architecture 19

3 libmapper **21**
3.1 Open Sound Control and libmapper Syntax 22
3.2 Structure of libmapper Networks 24
3.3 Connection Properties 25
3.4 libmapper Bindings 27
3.5 Prior Interfaces for libmapper 28
3.5.1 Maxmapper 28
3.5.2 Vizmapper 29
3.5.3 Webmapper 31
3.6 Evaluation of libmapper Variables as Visual Data 32

4 Design & Implementation **34**
4.1 Development of a Flexible System . . . 35
4.1.1 MVC architecture . . . 35
4.1.2 Top toolbar . . . 38
4.2 Integration of Interface Features . . . 40
4.2.1 Structure of ListView . . . 40
4.2.2 Display libmapper metadata . . . 41
4.2.3 Locating devices and signals . . . 42
4.2.4 Visual feedback . . . 44
4.2.5 Improvements to user interaction . . . 45
4.3 Extension of Control and Visual Elements . . . 47
4.3.1 Multiple selection . . . 47
4.3.2 Accommodating varying window sizes . . . 48
4.3.3 Visual redesign . . . 50
4.3.4 Alternate views . . . 51
4.4 Other GUI features . . . 54
4.4.1 Saving & Loading . . . 54
4.4.2 Creation of a standalone & distribution . . . 55

5 Applications & Discussion **56**
5.1 User Feedback . . . 56
5.1.1 General feedback . . . 57
5.1.2 Saving & loading . . . 58
5.1.3 Reliability & responsiveness . . . 58
5.1.4 Effectiveness of alternate views . . . 59
5.2 Testing program responsiveness . . . 59
5.2.1 Rate limiting functions . . . 60
5.3 Comparison to Similar Interfaces . . . 62
5.4 Evaluation of Goals . . . 64

6 Conclusions & Future Work **65**
6.1 Summary and Conclusions . . . 65
6.2 Future Work . . . 66

6.2.1 Unimplemented features 66
6.2.2 Possible extensions 66

References **68**

List of Figures

1.1 A sample of libmapper code 3

2.1 The function described in equation 2.1, graphed in two dimensions. 7
2.2 Equation 2.2 projected on the Cartesian plane. 8
2.3 Equation 2.4, a many-to-many mapping. 9
2.4 The four mapping classes 10
2.5 Cleveland and McGill's rankings for quantitative perceptual tasks. 15
2.6 An example of Tufte's $1 + 1 = 3$ noise 18
2.7 A dense interconnected network displayed with and without hierarchical edge bundling techniques 19
2.8 An illustration of Krasner and Pope's MVC structure 20

3.1 A simple libmapper network 25
3.2 The Maxmapper interface 29
3.3 The Vizmapper interface 30
3.4 The Webmapper interface 31

4.1 Structure of MapperGUI. Blue arrows show propagation of network changes, while dashed arrows denote messages requesting a network change. 36
4.2 The upper toolbar 39
4.3 ListView with all devices selected 41
4.4 ListView with device **testsend.1** selected 42
4.5 Functionality of hiding unconnected elements on a network with many devices and few links. 44
4.6 Multiple selection and row striping in ListView. 45

4.7 Draggable links and connections. 46
4.8 Simultaneous connection of multiple signals 48
4.9 Resizing the ListView window. Rows condense, scroll bars appear and the top menu collapses in the smaller version. 49
4.10 ListView before visual redesign 50
4.11 GridView 52
4.12 HiveView 53

5.1 Illustration of a delayed function. 61
5.2 STEIM's JunXion software 62
5.3 The OSCulator interface 63

List of Tables

2.1 An example of key/value pairs (countries and currencies) 11
2.2 Bertin's graphical relationships 15
2.3 Mackinlay's Graphical Rankings 16

3.1 libmapper metadata types 33

4.1 Metadata available in Webmapper versus ListView 43
4.2 Shortcut keys in ListView 47

List of Acronyms

DMI	Digital Musical Instrument
GUI	Graphical User Interface
IDMIL	Input Devices and Music Interaction Laboratory
API	Application Programming Interface
SWIG	Simplified Wrapper and Interface Generator
OSC	Open Sound Control
MVC	Model View Controller
HTTP	HyperText Transfer Protocol
HTML	HyperText Markup Language
CSS	Cascading Style Sheet

Chapter 1

Introduction & Motivation

> "In order that our tools, and their uses, develop effectively: (A) we shall have to give still more attention to doing the approximately right, rather than the exactly wrong..." (Tuckey 1965)

Throughout the vast majority of human history the term "musical instrument" has signified both the physical object with which the musician interacted *and* the direct source of the sound created: a violin with vibrating strings, a reeded saxophone, a timpani with its membrane, etc. With the advent of electronic sound in the late 19th century, it became possible for interactive objects to be separated from the sound producing devices they control (Chadabe 2000). As technological development progressed, so did the capacity to divide musical instruments into independent parts. With digitization it is now not only possible to arbitrarily connect a control element to any sound synthesis dimension, but also to modify this association according to the whims of the user. Since mechanical linkages are no longer necessary in the design of musical instruments, control surfaces can, and often do, take on a variety of wild and arbitrary shapes and modes of interaction.[1] All that is necessary for this process is for control devices to output some kind of electronic signal that other, sound-producing instruments can accept. With no obvious means of implementation, the success or failure of these new digital musical instruments (DMIs) often depends on how artfully their output signals are "mapped" to synthesis parameters.

More and more frequently, the mapping itself becomes a part of the expressive element

[1] International Conference on New Interfaces for Musical Expression. [Online]. Available: `http://www.nime.org/`. Accessed June 23, 2013.

of a musical work (Hunt and Kirk 2000), as it associates itself with both composition and performance with certain DMIs. Thus is becomes necessary for mapping to be dynamic and interactive: sometimes poured over in composition studios, or sometimes edited mid-piece. Musicians are not necessarily computer programmers, thus ideally the act of mapping should not require computer expertise. This means that on top of the low-level layer of interactive mapping (simply instructing a machine to connect signals to others in specific ways), there needs to exist an interface to make such an activity easy, logical, intuitive and in line with the artistic process.

As the actual act of mapping is as expansive and nebulous as the instruments it hopes to assist, thus the design of such a mapping interface presents many interesting challenges. Due to the tremendously wide variety of possible use cases, several seemingly contradictory goals emerge: What is the best way to visually represent complex musical networks while simultaneously allowing for the user to easily manipulate them? How can systems with many devices and signals be well represented while still allowing in-depth control of small networks? How can an interface be transparent to non-technical users while still accommodating all possible functionality that advanced users may wish to use?

1.1 Context and Motivation

The world of digital musical instruments is still dominated by keyboard type input devices. Though many novel DMIs currently exist (and many more are being created) these devices are usually unique and often difficult to use without their creator being present (Cook 2009). Since mapping is such an important feature of DMIs, a means of transparently editing them could inspire more people to use novel musical controllers. In response to this challenge, libmapper, a tool for collaborative mapping, was created at the Input Devices and Music Interaction Laboratory (IDMIL).

In its most basic state, libmapper takes the form of an application programming interface (API). APIs are primarily a means for different pieces of computer software to communicate with one another. The only possible way to communicate directly with the libmapper API is through coded text. For example, Figure 1.1 causes a synthesizer to announce itself and begin communicating with other devices on a libmapper-enabled network (Malloch et al. 2008).

This is obviously inaccessible to users who do not have the time or desire to read

```
#include <mapper.h>
mapper_admin_init();
my_admin = mapper_admin_new("tester", MAPPER_DEVICE_SYNTH, 8000);
mapper_admin_input_add(my_admin, "/test/input","i"))
mapper_admin_input_add(my_admin, "/test/another_input","f"))

// Loop until port and identifier ordinal are allocated.
while ( !my_admin->port.locked || !my_admin->ordinal.locked )
{
   usleep(10000); // wait 10 ms
   mapper_admin_poll(my_admin);
}

for (;;)
{
   usleep(10000);
   mapper_admin_poll(my_admin);
}
```

Fig. 1.1 A sample of libmapper code

through documentation files, or those who have no knowledge of programming semantics. A steep learning curve is especially a problem for a network tool like libmapper: because it is primarily a means of communication between instruments, it can only be successful if it is widely adopted. A libmapper-enabled controller will only be useful if many high quality libmapper synthesizers exist. In turn, synthesizer makers will only have incentive to incorporate libmapper into their designs if there are already controllers that use the system.

An API can be contrasted with a graphical user interface (GUI), an interface that contains abstractions on top of the raw code. These abstractions can be features like buttons, menus, visual representations of data, etc. In general, GUIs are designed to be familiar to those who have used digital devices in the past, and thus easy to learn and use. Two GUIs have been created for libmapper (see Section 3.5): a basic interface built in Max/MSP[2] and vizmapper (Rudraraju 2011), a more abstract representation of a libmapper network. Both of these GUIs have their strengths, yet neither adequately meets

[2]MAX: You make the machine that makes your music. [Online]. Available: `http://cycling74.com/`. Accessed June 17, 2013

the full range of possible use cases for libmapper. A more flexible approach is required if the GUI is to be usable in situations with hundreds of signals, transparent for systems with multi-leveled hierarchical devices, intuitive during performances where devices output light and haptic feedback as well as sound, and responsive for tasks where speed of manipulation is an absolutely necessity.

With such an interface in place, libmapper can greatly expand its user base. As a result, more controller and synthesizer designers may choose to incorporate libmapper into their devices, and in turn these devices will be easier to learn and use. Hopefully the end result will be greater adoption of non keyboard-based DMIs in the electronic music community.

1.2 Project Overview

The focus of this project is to create a graphical user interface for libmapper, hererafter referred to as MapperGUI. This interface aims to be flexible and intuitive, simultaneously allowing for useful control of the full range of possible libmapper networks while also not intimidating non-technical users with complexity. The presupposed solution to this problem is to provide users with multiple independent modes of viewing and interacting with the network. Certain view modes can excel in providing precise control, while others can help users understand the structure of complex networks. The idea is to provide multiple imperfect solutions to an unsolvable problem, so that each can be "...approximately right, rather than exactly wrong" (Tuckey).

This project was structured in four major, non-sequential parts: a review of prior visualized mapping interfaces, the integration of presently available GUIs for libmapper, the extension of interface features and the collection of user feedback. Results of the research phase informed implementation and are presented here. Development began by updating a cross-platform implementation of the current Max/MSP-based GUI, while integrating functionality from vizmapper. New view modes were integrated into design while refining functionality of the previous ones. Throughout the design process, MapperGUI was provided to potential users who gave feedback on the strengths, weaknesses and potential avenues for improvement.

1.3 Thesis Overview

The remainder of this document is organized as follows. Chapter 2 outlines concepts necessary for providing context for this thesis project. A wide variety of domains inform the creation of a musical mapping interface. Special attention is paid to mapping theory, data visualization, relevant existing user interfaces, user centric design techniques and specifics of libmapper itself. Chapter 3 describes the design and implementation of MapperGUI. This chapter presents design decisions made, technical details of implementation and how the user-centric approach informed the process. Chapter 4 evaluates results, both on the empirical level of software performance as well as qualitative user feedback. Finally, Chapter 5 presents conclusions of the work and suggests further developments for the software.

1.4 Contributions

The contributions of this thesis are: the exploration of issues related to user interface design for musical mapping networks, the design and implementation of an interface for libmapper that aims to improve on usability and flexibility of the system, and this thesis document, which describes the research and development therein.

Chapter 2

Background

Dynamic mapping is becoming an increasingly important requirement for digital musical instruments. This chapter surveys currently available tools that allow for manipulation of musical and non-musical networks in real time. The first section presents a review of mapping itself, both from a theoretical and a musical standpoint. This portion also introduces the libmapper application programming interface. The second section reviews relevant work in the visual representation of information. The following portion describes applicable techniques in user interface design. Finally, a review of user interfaces for mapping is presented.

2.1 Mapping

At the most fundamental level, *mapping* is the act of associating two or more sets of information. Mappings can be mathematical, computational, linguistic (like translation), geographic, or even poetic[1]. Within the context of DMI design mapping is the relationship between sensor outputs and synthesis inputs. The entire character of a new instrument can be drastically altered though mapping, even while control surface and sound source are held constant (Hunt et al. 2003). As a result, the theoretical formalism of mapping becomes yet another necessary tool in the modern instrument designer's arsenal.

[1] What is metaphor if not the association of unlike things?

2.1.1 Mapping theory

Mapping as function and mapping cardinality

From the perspective of mathematics, the term *mapping* is very nearly synonymous with *function* (Halmos 1970), as both describe how one set of numbers corresponds with another. The first group is commonly referred to as the *domain* and the second as the *codomain* or *range*. An in-depth review of functions in mathematics is beyond the scope of this thesis, however a few fundamental examples will be useful for reference in Section 2.1.2. The following are instances of two basic types of mathematical functions:

$$y = 2x - 1 \tag{2.1}$$

$$y = x^2 \tag{2.2}$$

Each function takes a single input value (x) and *maps* that number onto its range (y). The fact that each of these equations take in only a single number as input, and output a single number in turn, means they can be graphed in a two dimensional space. This is not necessarily the case, as functions can input and output lists of numbers (vectors). Mathematically they are not very interesting, but they represent two fundamentally different *kinds* of functions.

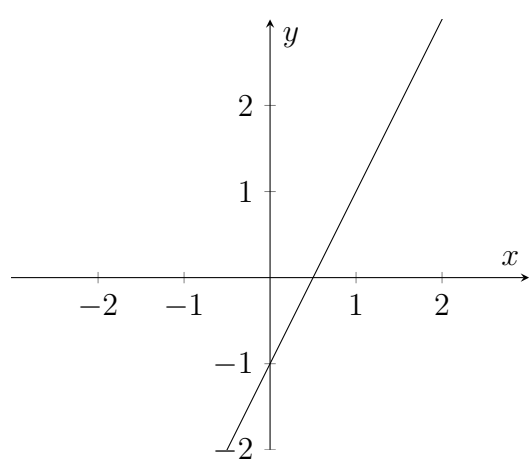


Fig. 2.1 The function described in equation 2.1, graphed in two dimensions.

For equation 2.1 each input value has *one and only one* corresponding output value. The same is true if the function is to be inverted, as each output value corresponds to

only one input value. The range is simply a scaled and shifted version of the domain. The mapping's *one-to-one* nature can clearly be seen in Figure 2.1. To mathematicians this is known as the mapping's *cardinality*.

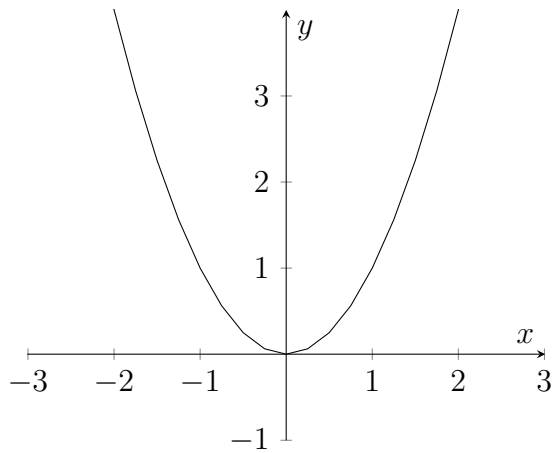


Fig. 2.2 Equation 2.2 projected on the Cartesian plane.

This is not the case for equation 2.2, for although each input has only one output, single positions in the codomain can have multiple corresponding inputs (e.g. both 3^2 *and* -3^2 are equal to 9). Thus we can consider equation 2.2 to be a classical example of a mapping with a cardinality *many-to-one*. In Figure 2.2 the range of the function is wrapped back onto itself such that a horizontal line could intersect the curve twice.

Two more mapping cardinalities are relevant to instrument design, an example of each:

$$y = \pm\sqrt{x} \tag{2.3}$$

$$y = \pm\sqrt{1 - x^2} \tag{2.4}$$

They are not considered to be functions by mathematicians[2], but are nonetheless important for our purposes. In equation 2.3 a single input can result in multiple outputs (an input of 4 results in the output of *both* 2 and -2), yet each output has only a single input. This is simply the inverse function of equation 2.2, and is an example of a *one-to-many* mapping. On a graph of such a mapping, a *vertical* line may cross at multiple points. The final equation is that of a circle centered at the origin with a radius of one. This is a

[2]In mathematics, a true function can have no more than one output value for every input value.

many-to-many mapping, as both it an its inverse result in multiple outputs from a single input.

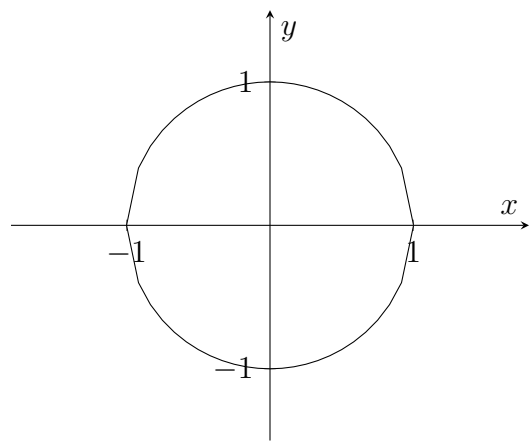


Fig. 2.3 Equation 2.4, a many-to-many mapping.

Though a graphical plane is the most common way for mathematicians to visualize two-dimensional functions, drawing the direct association between input and output will be more useful going forward. Figure 2.4 provides an illustration of such an approach. The astute reader will notice a striking similarity to the GUI view mode described in Section 4.2.1 and these diagrams.

Mapping as association

In computer science, a mapping is less commonly referred to as a function and more usually called an *associative array* or a *dictionary*, though the word *map* is also used (Mehlhorn and Sanders 2008). This type of data structure is generally the most flexible way for computers store information. An associative array consists of key/value pairs, where the *value* is the data to be stored and the *key* is the reference to that data.

In table 2.1 the data is non-numeric and associations between keys and values are arbitrary (from a mathematical point of view). There obviously exists no distinct function that can transform a countries name into the name of its currency, thus the computer must explicitly remember the associations between the words in the form of a *hash table*. At the lowest level, computers store information on a vast array of zeros and ones, and the value "Kwanza" only arises through a non-trivial process of encoding and decoding. In order to retrieve it the computer *must* know where it can be found. The hash table takes the input

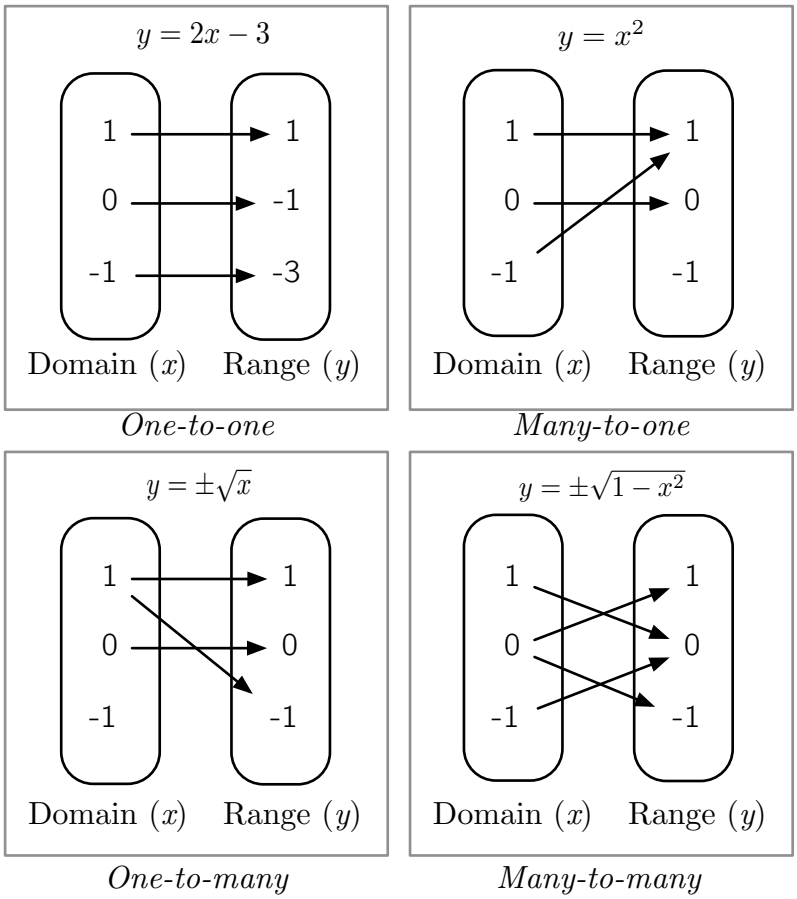


Fig. 2.4 The four mapping classes

of a key, finds the address for the value and returns it. In this way the hash table is literally the association between two sets of data and thus the mapping between them.

The four mapping classes outlined in the above section are not limited to the functional domain. The associative array in table 2.1 is another example of a many-to-one mapping, as many countries have the same currency name. In this vain a one-to-many mapping could be the same keys with values switched to "Former Monarchs" ("France" would map to both "Louis XVI" and "Napoleon III", etc), while a value of "Official Languages" would be a many-to-many mapping ("Canada" maps to both "English" and "French" while both "Canada" and "France" map to "French").

Though most applicably represented in computer science, data structures like associative arrays appear in many other fields. Library card catalogs (one-to-one), multilingual

Table 2.1 An example of key/value pairs (countries and currencies)

key	value
Canada	Dollar
France	Euro
Bahrain	Dinar
Germany	Euro
Angola	Kwanza
USA	Dollar

dictionaries (many-to-many) and address books (many-to-one) are all very straightforward instances of key/value pairs. In a library card catalog the call number even acts as a sort of hash table. In a large library, a book that is placed in the incorrect position on the shelves will likely be lost for a very long time. Thus the system must not only remember the keys (titles) and associated values (the books themselves) but also their positions in memory, their call numbers.

2.1.2 Mapping for digital musical instruments

With an acoustical musical instrument a musician must interact directly with the physical object that produces the sound. In this context, the concepts of "control surface," and "synthesis devices" are not very relevant, as they are intrinsically linked. In the case of an acoustic guitar the pick *could* be considered to be a sort of control device (as it is primarily used for instrumental interaction), with the strings and body a acting as the sound producing section. The problem with this type of approach is that changing the material of the pick, perhaps to give a different feel for the player, will also necessarily modify the sound produced. The same can be said for modifying nearly any aspect of an acoustic instrument: it will change both the control interface and the created sound. This coupling of parameters causes any concept of a *mapping layer* to be irrelevant.

As stated in the introduction, this is not the case for electronic instruments (Hunt et al. 2000). Electronic sensors transduce musical gestures into signals, which are in turn converted into auditory phenomena by amplifiers and speakers. Any arbitrary transformation can happen to the signals[3] in between these two phases. This flexibility is most obvious with outlandish novel instruments like (TODO: maybe pictures?), but is fundamentally true for

[3]Especially digital signals, which are remarkable for their robustness and mutability.

any electronic instrument. An electric guitar senses gesture with a magnetic pickup that transforms the signal of a vibrating string into an electronic signal, which is made audible by an amplifier. Though this can happen directly, more or less reproducing the sound of an acoustic guitar, it is also possible to greatly modify this signal before it is amplified, creating tones that may be unrecognizable as the original acoustic instrument.

The mapping layer

In response to the importance of this uncoupling of parameters, electronic instruments are often conceptualized as having three independent layers (Wanderley and Depalle 2004): (TODO diagram)

- The "gestural controller": The device with which the musician interacts directly. It generally has sensors that collect gestural data and can provide haptic feedback. The generated signals are output into the mapping layer.
- The "sound generation unit": This device receives input signals from the mapping layer and uses them to generate sound. This layer can contain melody generating algorithms, sound modifying effects, physical models of acoustical instruments or any other construct that is directly used to produce sound.
- The "mapping layer": The abstract space that receives input signals from the gestural controller and outputs to the sound generation unit. These signals can be connected and modified independently of actions in the other two layers.

As can be seen above, the words "output" and "input" become ambiguous, whether one is speaking from the perspective of devices (control devices *output* signals that are *input* into the synthesis devices) or the perspective of the mapping layer (the mapping receives *input* from the controller which is *output* to the synthesizer). For the detailed analysis of mappings and mapping devices, this can obviously create confusion. To avoid this, signals arriving at the mapping layer from the control surfaces will be referred to as *source signals* and signals sent from the mapping layer to the sound generation units will be called *destination signals* for the remainder of this thesis. This follows the nomenclature described in Malloch et al. (2013) and the libmapper API in general.

Functional versus systems perspective on mapping

Both the more mathematical perspective of mapping as functions and the computer science standpoint of mapping as association are relevant to DMI design. These two concepts are often referred to as the *functional* and the *systems* points of view for mapping, respectively (Nort 2010).

Once two signals are connected, say the position of a knob and the cutoff frequency of a low-pass filter,[4] it is very possible that the raw numbers sent from the knob are not appropriate as input for the filter. It may be that the knob transmits numbers ranging from 0 - 127 (2^7) and the filter accepts numbers from 0 - 1023 (2^{10}). As a result the filter will always be more or less closed no matter how the user turns the knob. To account for this, the mapping needs to *scale* the source signal (by a factor of 8) to fit the destination range. This is a functional kind of mapping, analogous to Section 2.1.1. The source signal may need to be transformed in many ways.

The other, higher-level perspective on mapping deals with the actual connection of source to destination signals. On any mapping network there can exist several devices, each with numerous signals. The act of associating devices with devices, signals with signals can drastically change the character of a DMI or group of DMIs. This is known as the systems perspective on mapping. It is necessary for libmapper and the GUI to be able to assist with both kinds of mappings.

Mapping strategies

For expressive musical networks, simple one-to-one mappings are often insufficient. Kvifte (2008) argues that it is extremely rare to find such associations in acoustic instruments, as the control parameters are usually tightly coupled with several acoustic dimensions. Interfaces with hundreds of knobs and sliders, each one connected to a single sound parameter have thus been found to "...hinder rather than help expressive musical behavior." (Kvifte) In practical experiments where mappings of varying complexity are compared, the most complex were generaly found to be the most expressive and useful (Hunt and Wanderley 2002). However, Goudeseune (2002) states that mappings need to be simple enough for the performer to comprehend them. Goudeseune argues for "...static mappings over dynamic,

[4] A standard synthesis parameter that controls the brightness of a sound, think of the difference between the vowel 'o' in 'food' (low cutoff) and the vowel 'a' in 'sad' (high cutoff).

and simple over complex" and proposes an algorithmic solution to compute them. These "interpolated mappings" are generated by associating single points in the source and destination spaces (i.e. certain performer gestures with certain sounds) and mathematically filling in the spaces between.

One proposed solution to the cognitive complexity of associating many source and destination signals is to create a second mapping layer (Hunt et al. 2000). Instead of dealing with raw sensor output, like acceleration and inclination, musicians can interact with more interesting gestural information such as "jab" or "left-arm swing." These "cooked" parameters are argued to be more meaningful and useful musical information than the raw signals. This approach is explored in Momeni and Henry (2006) for mapping both to audio and visual synthesis. The conventional wisdom that mappings need to be complex, yet transparent and meaningful all point to the necessity of a tool for the intuitive and expressive configuration of mappings.

2.2 Data Visualization and User Interface Design

The graphical user interface described in this thesis is a purely visual interface. No means of auditory or haptic response was implemented, or even seriously considered. Creating an auditory tool for controlling musical instruments is obviously problematic and most personal computers provide no means of producing haptic feedback. This obviously limits the usable dimensions, but also greatly simplifies the problem of how to best represent the tremendous variety of libmapper networks.

Fortunately graphic designers and statisticians have already deeply probed the problem of how to best display data visually. It is necessary here to briefly review some of this work, especially the techniques relevant to the creation of a libmapper GUI and visual systems from which inspiration was drawn.

2.2.1 Graphical dimensions

The visual dimension can be broken down into many sub-dimensions. These dimensions are not fully separable, but doing so creates a useful paradigm for understanding and creating solutions for our visual problem. Bertin (1983) presents a simple vocabulary for categorizing graphical objects and relationships.

Table 2.2 Bertin's graphical relationships

Marks	Points, lines and areas
Positional	1-D, 2-D and 3-D
Temporal	Animation
Retinal	Color, shape, size, saturation, texture and orientation

Visual presentations use marks to encode information by way of their positional, temporal and retinal qualities. In table 2.2 *retinal* properties are so called because the eye is sensitive to them independently of their position. Though the third positional dimension is relevant and would be useful, it is currently beyond the scope of this research, not to mention the hardware on which the GUI is to run.

Cleveland and McGill (1984) expand on this vocabulary, enumerating further sub-dimensions of marks and retinal properties. An experiment is described in which subjects are asked the relative values of various visual objects (e.g. the first box is 50% larger than the box on the left), for various visual dimensions. From the data, they were able to create a ranking of visual dimensions for quantitative information. In Figure 2.5, differences between objects are more accurately perceived when the difference is encoded using a variable higher up the chart. Note that variables like shape, texture and opacity are not included.

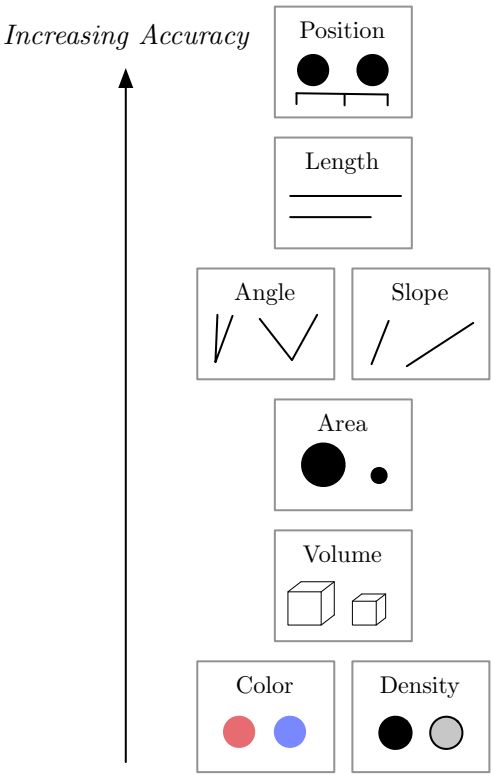


Fig. 2.5 Cleveland and McGill's rankings for quantitative perceptual tasks.

Mackinlay (1986) uses this ranking to expand into non-quantitative data sets. Nominal information is that in which elements can be understood to be similar or dissimilar to one another, yet have no definite order or value. libmapper uses nominal information in the form of device, signal, link and connection names, as well as connection modes and boundary conditions. Ordinal data fits between quantitative and nominal, ordinal items are understood to be greater than or less than one another, while having no definite numerical

ratios. If multiple devices of the same class are present on the same libmapper network, libmapper will append ordinal numbers to the end of their device names (e.g. `tstick.1`, `tstick.2` and `tstick.3`).

Table 2.3 Mackinlay's Graphical Rankings

quantitative	**ordinal**	**nominal**
position	position	position
length	density	color hue
angle	color saturation	texture
slope	color hue	connection
area	texture	containment
volume	connection	density
density	containment	color saturation
color saturation	length	shape
color hue	angle	length
texture	slope	angle
connection	area	slope
containment	volume	area
shape	*shape*	volume

In table 2.3, items in italics are considered unsuitable by Mackinlay. Though position is the most accurate dimension for all types of data, dimensions like *length* differ widely. For the visualization of libmapper networks, it is often necessary to encode many dimensions of data onto a single mark. Devices, signals, connections and links all have a set of meta-data with quantitative, ordinal and nominative information. In the design of an effective GUI it will be necessary to properly associate high-accuracy visual dimensions to network properties that require them and reserve low-accuracy dimensions for those that do not. In this way the problem of this thesis conveniently becomes one of mapping: how can we best correlate visual dimensions with properties of libmapper networks?

2.2.2 Relevant visualization techniques and systems

Encoding Color

"Color" itself is a multi-dimensional phenomenon that does much to communicate information in modern user interfaces. Since color was an uncommon feature of computer displays at the time neither Bertin (1983) nor Cleveland and McGill (1984) explore its use in depth.

Cleveland and McGill simply state that it is not good for encoding quantitative information. Mackinlay (1986) elaborates on this, breaking color into "hue" and "saturation," upgrading its use for ordinal and nominal data.

Tufte (1990) provides a definite procedure for incorporating color into evidence displays[5]. Techniques are gleaned from centuries-old map making and applied to computer interfaces. Principal rules, summarized and expanded from Imhof (1982), are found to be (paraphrased):

- *First rule*: Bright colors are painful when used uninterrupted over large areas or when placed adjacent to each other, but can be extremely powerful when used sparingly while accompanied by dull tones.
- *Second rule*: Light, bright colors produce unpleasant results when accompanied with the color white.
- *Third rule*: Background and base colors should be muted or neutral. For this reason, *gray* is regarded to be the most versatile of colors.
- *Fourth rule*: Two or more large, enclosed areas within a single display cause the image to fall apart. Unity can be maintained if the colors of one section are interspersed throughout the other. "All colors of the main theme should be scattered like islands in the background color."

Links and causal arrows

For the visualization of networks, the idea of a visual "connection" becomes very important. This linking action is usually accomplished by an arrow-like object in evidence displays. Tufte (2006) enumerates numerous guidelines for incorporating line-like objects into presentations. Again drawing inspiration from map making (an obvious inspiration for "mapping"), the use of differentiation among linking arrows is greatly emphasized: "Nouns name a specific something; arrows and links are too often non-specific, generic, identical, undifferentiated, and ambiguous." The use of many line properties, such as dashing, arrow-heads and color can better illustrate a variety of influences in a linked chart.

[5] Tufte's favorite term for data-driven graphics

Tufte also cautions against using heavy line weights when unnecessary, as it effectively decreases display resolution. Thick lines are also more likely to create $1 + 1 = 3$ noise, or the effect of negative space acting as a display feature.

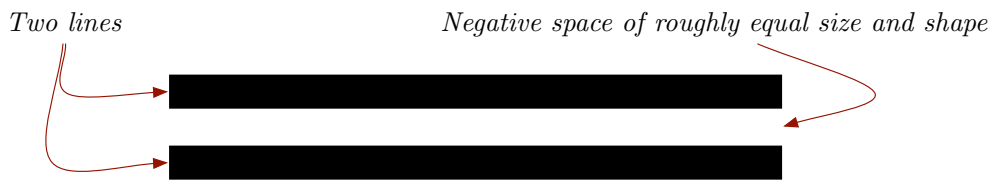


Fig. 2.6 An example of Tufte's $1 + 1 = 3$ noise

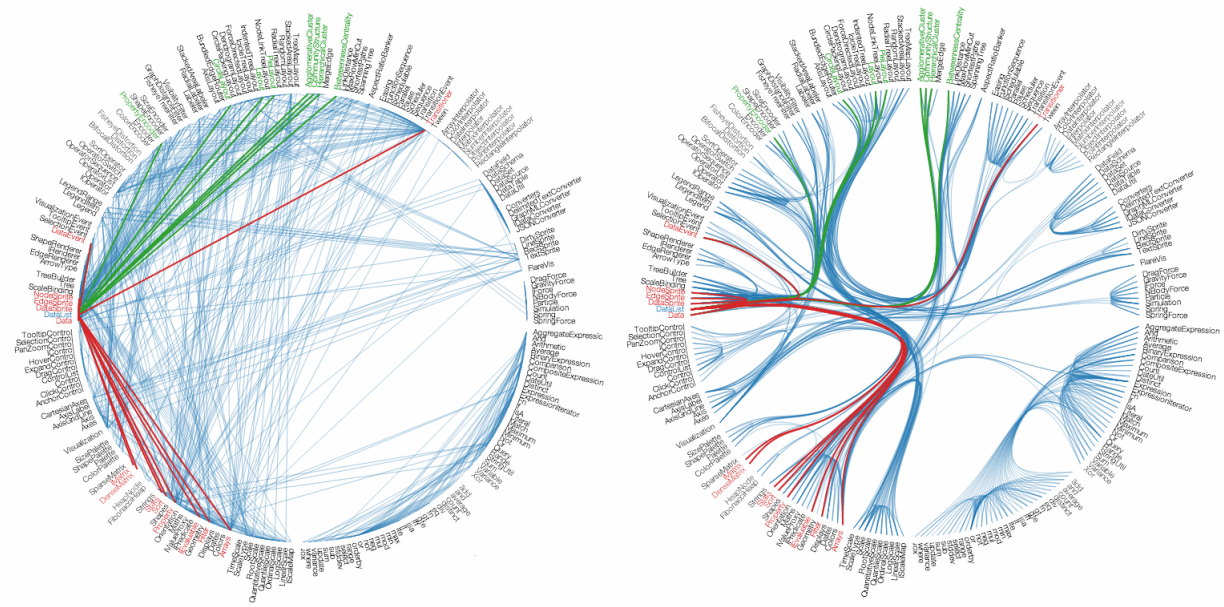
In Figure 2.6 the negative space between the two lines appears as its own white line, as opposed to simply empty space. In displays with numerous or thick lines this can cause negative space to compete with informative features, attenuating the overall effectiveness of the display. $1+1 = 3$ noise plagues dense computer user interfaces, borders and other non-essential display features should be lightened, thinned and removed when at all possible.

Hierarchical edge bundling

Of course, in diagrams with tremendous amounts of connections no amount of thinning and coloration can create an informative display. The technique of hierarchical edge bundling (Holten 2006) groups lines based on adjacency relationships. Displays take advantage of hierarchical information encoded within the dataset. Linking arrows are curved towards other arrows that are connected to related elements. Figure 2.7 demonstrates this effect for arbitrary data.[6]

In a libmapper system, this would mean that connections between signals on the same device will be pulled towards one another. If a hierarchical structure exists in the naming convention, connections between related signals will experience an even stronger force between them. For example, the connections from signal `tstick.1/raw/accelerometer/1/x` will be bundled tightly with connections from signal `tstick.1/raw/accelerometer/1/y`, but less tightly to `tstick.1/raw/accelerometer/2/x`. Any of these connections will not be pulled at all towards connections from signals on other devices.

[6]Images courtesy of: mbostock - The d3 visualization library. [Online]. Available: `https://github.com/mbostock/d3/wiki/Gallery`. Accessed July 24, 2013



No hierarchical edge bundling

With hierarchal edge bundling

Fig. 2.7 A dense interconnected network displayed with and without hierarchical edge bundling techniques

Braun

Braun is an application for visualizing OSC data flows on a scrolling graph (Bullock 2008). Users are presented with options to adjust what dimension is displayed on the y-axis, the x-axis being reserved for time. Multiple data flows can be viewed on the same set of axes, and time scales can be set arbitrarily, giving the users an overall impression of trends in OSC messages over their networks. An extremely simple visualization, it creates a sort of oscilloscope, but for network OSC data.

2.2.3 The model-view-controller architecture

As computer user interfaces regularly contain many interdependent parts, problems can occur if the code is not rigorously structured. The model-view-controller (MVC) architecture (Krasner and Pope 1988) is system by which interface features can be made modular. This is especially relevant to the work of this thesis, as we are attempting to create multiple modular views for the same interface (see Section 4.1.1).

The MVC architecture consists of three main parts, the model, the view and the con-

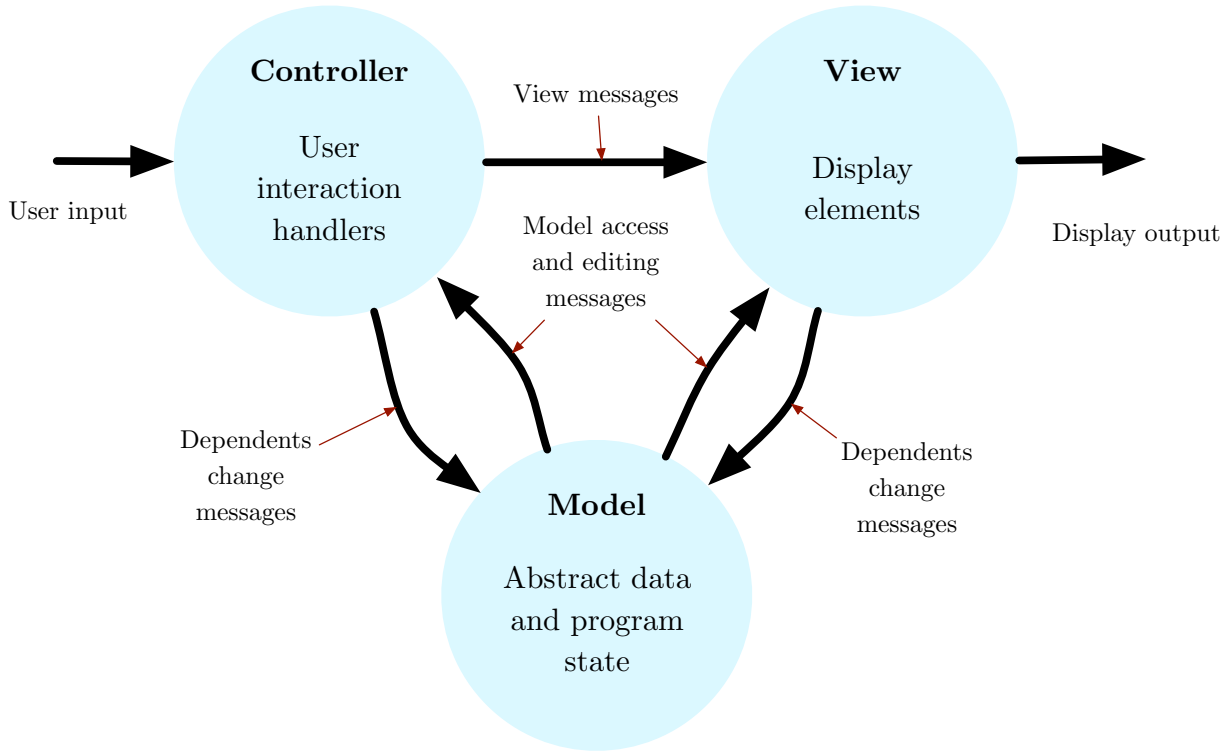


Fig. 2.8 An illustration of Krasner and Pope's MVC structure

troller. The model consists of an abstract representation of all that is present in the interface. It contains data independently of how it is being viewed. The view contains software elements that actually show on the screen. Typically these use data from the model to affect their display. The controller is the portion of the software that interfaces with the user, relaying messages to the model to change the state of the system. The three sections communicate with one another by a messaging standard defined by the designer.

The division between the view and the controller is not always clear, and it is sometimes beneficial to program in view-controller pairs (Krasner and Pope 1988). These pairs both display data and accept user interaction, though the data of the model is still treated as a separate class. What is important is that view-controller pairs are written modularly, such that many pairs can interact with the same model. This improves program extensibility, for if a new kind of interaction or display is desired, it simply needs to conform to the established communication standard and it will function properly.

Chapter 3

libmapper

The McGill Digital Orchestra project[1] began in 2006 with the aim of helping researchers and performers in music technology to work collaboratively in creating hardware and software solutions for live performance with digital technology. The libmapper project began in response to the difficulty of creating dynamic musical mappings in a collaborative setting (Malloch et al. 2008). In its most basic state, libmapper is a library for connecting things. As described by its website:

> "libmapper is an open-source, cross-platform software library for declaring data signals on a shared network and enabling arbitrary connections to be made between them. libmapper creates a distributed mapping system/network, with no central points of failure, the potential for tight collaboration and easy parallelization of media synthesis. The main focus of libmapper development is to provide tools for creating and using systems for interactive control of media synthesis."[2]

Without libmapper, DMI designers are usually required to "hard-code" mappings into their designs. This has the disadvantage of being slow to modify, as it might be necessary to re-compile[3] code any time a change is made. If the DMI is built in a development environment like Max/MSP modifications can be more quickly implemented. Max/MSP

[1] The McGill Digital Orchestra. [Online]. Available: `http://www.music.mcgill.ca/musictech/DigitalOrchestra/`. Accessed July 9, 2013

[2] libmapper: a library for connecting things. [Online]. Available: `libmapper.org`. Accessed June, 2013

[3] A process in which human-readable code is translated into something the computer can understand. This can take anywhere from a few seconds to days.

is a "high-level" abstraction on top of machine readable code, so Max/MSP programs are prone to slowness and cross-compatibility issues, inhibiting collaboration (Place and Lossius 2006). In either implementation it is difficult for someone other than the original designer to modify mappings.

As a C[4] library, libmapper does not introduce many abstractions on top of the data and can work quickly. Any device that embeds libmapper in its code can communicate with other devices that have done the same. In a libmapper network devices communicate with one another directly, as opposed to through some centralized network device. This means that less data overall needs to be sent over the network, and failure of a single device (like the router) will not crash the entire system (Malloch et al. 2013), an especially dire situation during live performance.

Another advantage of libmapper, which is especially relevant to this project, is the ability to create an administrative device. These "monitors" can query libmapper devices for data, and thus collect data on the network overall. Monitors also are able to create, destroy and modify connections on the network. This allows for external visualization and control of a libmapper network.

3.1 Open Sound Control and libmapper Syntax

Like any communication, communication between digital devices functions well only when the devices speak the same language. In the Internet age this becomes particularly relevant: the vast array of continuously connected devices, sending and requesting information would instantly collapse if every developer coded to his or her own idiosyncrasies. To prevent this, computer scientists make use of various communication "protocols" when creating software. Hypertext Transfer Protocol (HTTP) is the most famous example of such a system.

At its core, libmapper builds its on language on top of the Open Sound Control (OSC) protocol, as described by Wright and Freed (1997). OSC defines the format for messages that are sent between sound producing devices (as implied by the name), but can also be used for related multimedia devices such as stage lights or vibrating motors. It provides means for flexible, high-resolution communication and was intended to replace MIDI[5], the

[4] An extremely popular, multi-purpose programming language.

[5] MIDI Manufacturers Association - The official source of information about MIDI. [Online]. Available: `www.midi.org`. Accessed July 11, 2013

30-year-old standard for musical instrument communication.

OSC formats messages much like Internet URLs, arbitrary strings of characters separated by '/' characters. libmapper messages also take on this format, using the structure to expose hierarchy of signals:

- `tstick.1/raw/accelerometer/1/x`: The data for the 'x' dimension of the first accelerometer of the first instrument of class "tstick" on the network (see TODO for a description of the gestural controller T-Stick). Here the word "raw" denotes that no pre-processing has been applied to this signal.
- `tstick.1/raw/accelerometer/2/y`: A signal transmitting the data for the same instrument as above, but the 'y' dimension of the second accelerometer.
- `tstick.1/cooked/accelerometer/2/amplitude`: A "cooked" signal. All three dimensions of accelerometer 2 are combined to compute the overall acceleration of the point. These signals can also be cooked to expose angle and elevation as signals.
- `granul8.2/filter/evelope/frequency/low`: The data for the low-end cutoff for the shape of the filter for the instrument named "granul8.2" (a granular synthesizer, thus a destination device).

This structure of signal names aims to be semantically relevant, and allows a GUI to display hierarchical structure of networks. Any one of the above signals transmits not only the signal's value, but also metadata. Signal metadata usually includes data type, length (single number vs. vector), units like volts or meters per second, maximum value and minimum value. Designers can "tag" signals with any extra metadata they may wish to add, such as physical position, color or owner's name. In the GUI it is necessary to allow users to view and manipulate any arbitrary kind of signal metadata.

To make signal names as coherent and consistent as possible, libmapper makes use of the *Gesture Description Interchange Format* (GDIF) (Jensenius et al. 2006), which provides a standard for motion capture data. Structures are given short, semantically relevant names. GDIF also provides a standard vocabulary for describing motion with dimensions such as "weight," "space," "time" and "flow." Though these standards are not enforced, as libmapper signals can be given any sort of names by their creators, most extant libmapper-enabled devices use them.

3.2 Structure of libmapper Networks

In order to maintain internal consistency, libmapper introduces a naming convention of its own. At the heart of any libmapper network are *signals*, defined in Malloch, Sinclair, and Wanderley (2013) as:

> "Data organized into a time series. Conceptually a signal is continuous, however our use of the term signal will refer to discretized signals, without assumptions regarding sampling intervals."

Here Malloch et al. are referring to digital as opposed to analog signals (hence the use of the term "discretized"). Notice how signals are not necessarily numeric by this definition, though they will almost certainly will be going forward. Signals are the only information actually passed from control surfaces to synthesizers, all other data structures exist to organize and label them. *Source signals* is data entering libmapper from control surfaces while *destination signals* belong to synthesizers and receive data. A *connection* is a bridge between two signals. Once a connection is created within libmapper, a source signal begins sending its data to a destination signal. A single source signal can be connected to many destination signals (a one-to-many mapping), but at the time of the writing of this document single destination signals cannot receive input from many source signals (a many-to-one mapping). Justification for this lack of functionality is discussed in Malloch et al. (2013).

Devices are essentially groups of signals. A device often has some kind of physical entity that makes the grouping logical, e.g. the "T-Stick," which has many "child" signals. Within software the grouping is usually a discreet computer program. With development environments like Max/MSP users are free to group signals into devices however they may wish. As mentioned previously, libmapper devices do not send all signal data to some centralized router, and instead work directly with one another. In order to accomplish this devices must be explicitly *linked*. Figure 3.1 demonstrates instances of libmapper devices, signals, links and connections.

Devices and signals can carry a variety of *metadata*. Devices usually list the number of child signals they possess and their location on the network (IP address and port). As previously stated users can tag devices and signals with arbitrary metadata. Connections have a much more specific set of metadata.

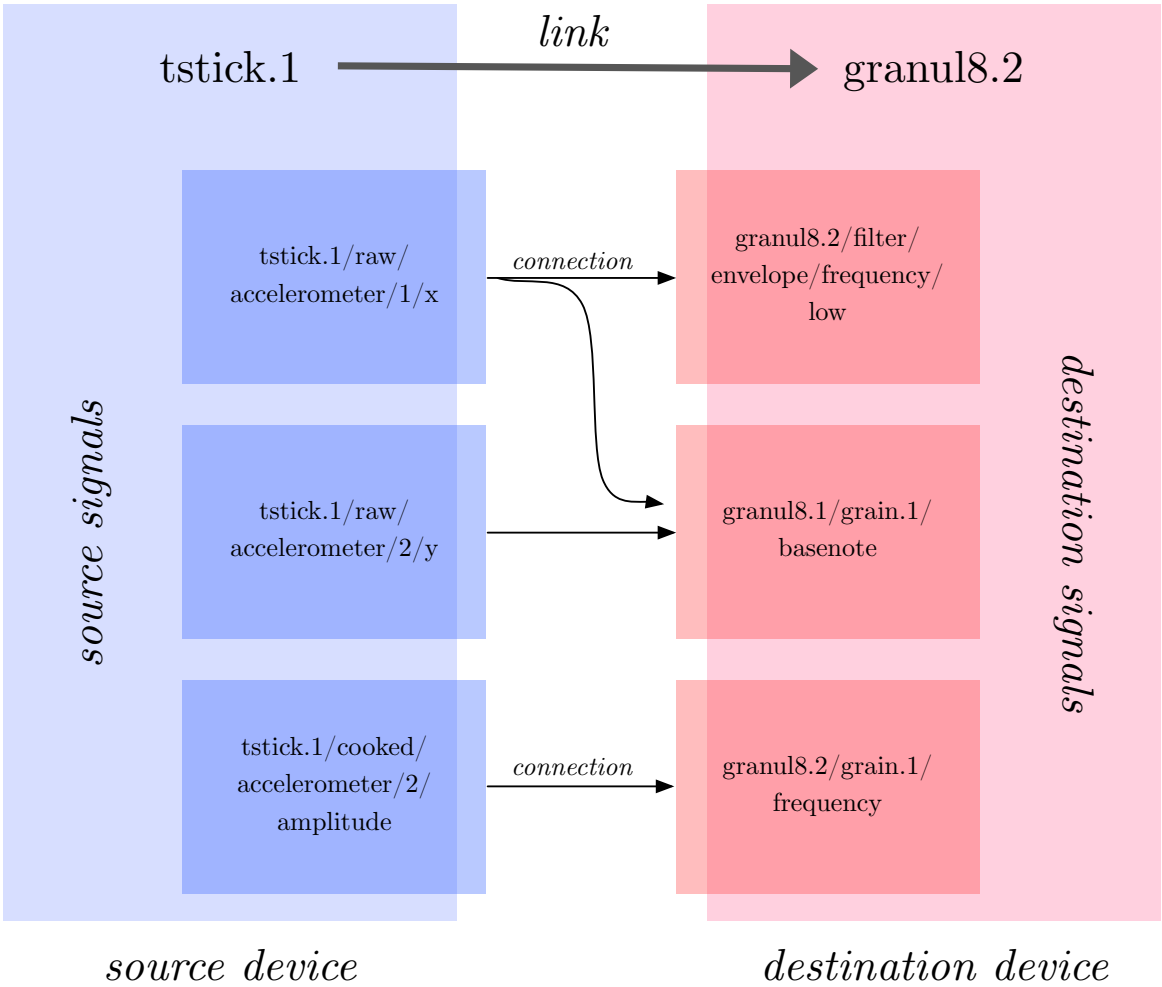


Fig. 3.1 A simple libmapper network

3.3 Connection Properties

Creation of links and connections is mapping from the systems perspective, but libmapper also allows for functional mapping through the modification of connections. This can be accomplished by altering certain properties possessed by every libmapper connection:

- **Expression**: A mathematical equation relating the source (x) to destination (y) values. An expression of $y = x$ will simply pass through source values, while an expression of $y = 3x + 2$, will apply a linear transformation to the source data (e.g. a value of 1 will be output as 5). libmapper supports a variety of expressions, including

exponential functions, trigonometric relations, comparison operators, derivation and integration.

- **Range**: An array of four numbers containing the user-specified maximum and minimum values for both the source and destination signals.
- **Mode**: The type of connection, this influences the effect of the expression and range properties and can be one of four categories:
  - *Linear*: libmapper automatically scales the output such that it fits the destination range, based on the source range. For example, if a certain connection has a source range of $[0, 1]$, and a destination range of $[5, 10]$, libmapper will automatically apply an expression of $y = 5x + 5$, such that the minimum and maximum source values will correspond to the minimum and maximum destination values respectively. A source value that is outside of this source range will result in a destination value that is also outside of the range. In this mode the user cannot directly modify the expression.
  - *Calibration*: The same as the linear mode except the source range parameter is ignored. libmapper instead polls the source signal to find the source range directly.
  - *Bypass*: Source values are sent directly through to the destination signal, as would happen with an expression $y = x$.
  - *Expression*: The user is able to set the expression to any arbitrary relation.
- **Boundary**: What is to happen to data values when they extend beyond the destination range. There are four options:
  - *None*: Values are passed through unchanged.
  - *Clamp*: Values outside of the boundary are constrained to that value.
  - *Mute*: No values outside of the boundary are passed to the output.
  - *Wrap*: Values exceeding the maximum are "wrapped" back to the minimum bound and vice versa.
  - *Fold*: When the signal passes outside of the boundary, the value is inverted back onto the destination range.

- **Mute**: A true-or-false value muting and un-muting data sent over the connection.

- **Send as instance**: Not all signals on libmapper networks are unique and long lasting, a good example being a keypress on a keyboard. During the keypress, data like aftertouch and release can be sent, making it a bona fide signal. However, musicians constantly create and complete keypress events during performance with keyboard instruments. To maintain every keypress as a unique signal with unique metadata would be tremendously unhelpful for mapping. Also, forcing a user to map every keypress event individually would make live performance impossible.

  To support this, libmapper gives connections the *send as instance* property. Sending data as an instance means that libmapper treats the connected signals as instances of a general class of signals. New instances of a signal class will be treated like previous instances and do not need to be mapped individually.

- **Link scope**: Not a property of connections, but of links. By default links are "scoped" to notify destination devices of the creation and destruction of signal instances on linked source devices. For intermediate devices, ones that function as both source and destination, this may not be the desired behavior. If device `A` is linked to intermediate device `B`, which is in turn linked to device `C`, then `C` will not be notified of instance events on `A` with default link scope settings. The user can modify the scope of link `B` → `C` to include `A` if desired.

3.4 libmapper Bindings

A final crucial libmapper feature is its multi-language *bindings*. The C language is "low-level" in that it is very and does not allow for very abstract data structures. This makes it extremely flexible, but difficult and time consuming to use. To make libmapper more friendly for different kinds of developers, "bindings" have been created for the higher-level Python[6] and Java[7] programming languages. libmapper functions are bound to other languages using the Simplified Wrapper and Interface Generator (SWIG)[8]. SWIG auto-

[6]Python Programming Language - Official Website. [Online]. Available: `http://www.python.org/`. Accessed July 17, 2013

[7]java.com: Java + You. [Online]. Available: `java.com/en`. Accessed July 17, 2013

[8]Simplified Wrapper and Interface Generator. [Online]. Available: `http://www.swig.org/`. Accessed July 17, 2013

matically writes a kind of dictionary that interprets function calls from other languages to the original C. Automatically generated files sit in-between the controlling code and the original library.

Though concept of "mapping" itself is extremely abstract, the libmapper API places it into a concrete context. libmapper is not only means of organizing networks though the creation and destruction of links and connections, it is also a tool for customizing response by its support for modifying connection properties. In this way it can serve both the high-level systems perspective and the low-level functional view of mapping. Though designed for musical devices, the API's loose framework could readily be applied to any type of multimedia system. libmapper is an extremely powerful, flexible tool and requires a user interface that can elegantly deploy its full range of capabilities.

3.5 Prior Interfaces for libmapper

3.5.1 Maxmapper

At the beginning of this project the most commonly used GUI for libmapper was a Max/MSP application designed by Joseph Malloch at IDMIL, referred to here as Maxmapper. A list-style interface (see Section 4.2.1), Maxmapper allows users to connect signals by dragging between elements on two tables. All source devices are listed in an array of tabs above, clicking on these tabs displays child signals for the device, as well as child signals for all linked devices. The GUI features a top toolbar for saving, loading and editing signal behavior. Maxmapper is extremely functional, and has been used with a wide variety of projects, performances and experiments. To many libmapper users, Maxmapper *is* libmapper.

Though Max/MSP works well for creative uses, as well as for prototyping software, it has some well-known limitations that inhibit the functionality of programs like Maxmapper. All Max/MSP standalones must be bundled with a set of necessary objects from Max/MSP itself. This leads to much larger programs, currently Maxmapper occupies nearly 16 times as much computer memory as the MapperGUI standalone.[9] The program is also relatively slow to launch, and requires a larger share of computer resources than other implementations. Due to the dependent nature of the code, it is also difficult to maintain and extend

[9] 22.1 megabytes versus 1.4 megabyes.

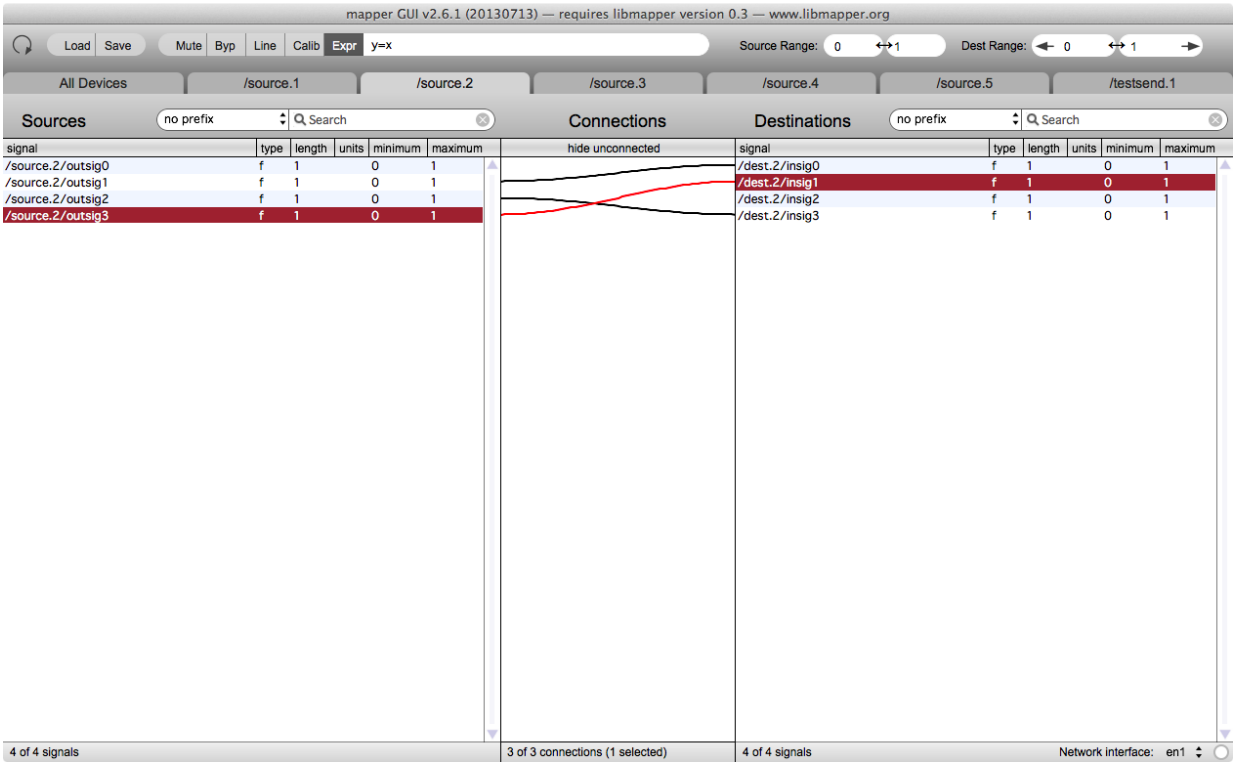


Fig. 3.2 The Maxmapper interface

Maxmapper, as updates to Max/MSP can cause errors for the program.

The greatest limitation of Maxmapper, and the principal motivation for this project, is the cross-incompatibilty of Max/MSP. The program does not run on Linux systems, and cannot be ported to mobile applications. For the creators of libmapper, this is seen as a fatal flaw. For libmapper to be successful it must be widely adopted, and to dis-include all non-Widows or Macintosh users is unacceptable.

3.5.2 Vizmapper

List-style views for libmapper do not scale well for large and complex networks. To address this need, the Vizmapper provides a novel visualization tool for libmapper networks (Rudraraju 2011). In Vizmapper devices and signals are symbolized by circles distributed about the perimeter of a central screen. Unlike other interfaces, Vizmapper allows the user to zoom in on particular groups of signals, if their names imply some kind of heirarchical structure. For example, the signals `tstick.1/raw/accelerometer/1/x` and

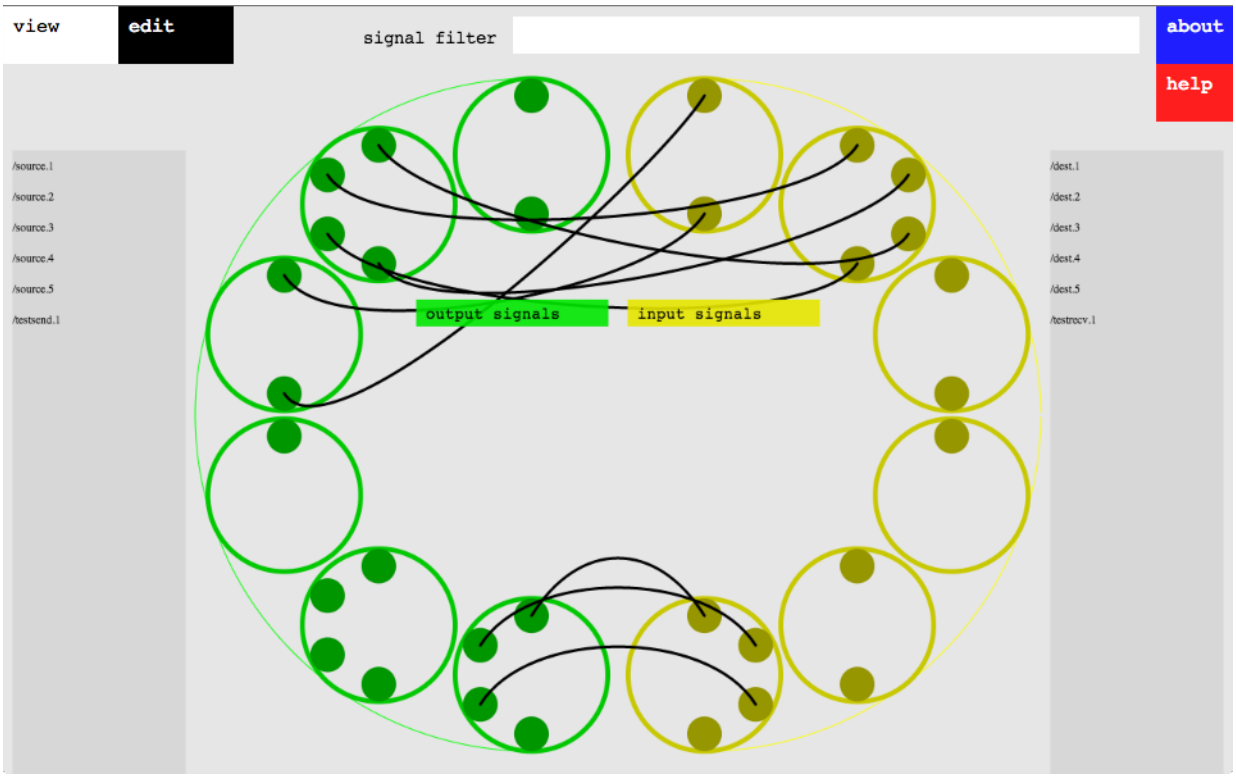


Fig. 3.3 The Vizmapper interface

`tstick.1/raw/accelerometer/1/y` are displayed as two different circles within the circle `tstick.1/raw/accelerometer/1`. By clicking on this circle the view redraws the left portion of the display to only show signals which are sub-signals of the T-Stick's first accelerometer.

In this way Vizmapper is capable of displaying all connections on a network simultanously, giving the user a better impression of overall structure. Unfortuantely many functionalities of Maxmapper are not included in Vizmapper. Notably the user can only form connections and links by navigating menus and editing text (as opposed to dragging between nodes). In order to benefit both from the visualization of Vizmapper and the interactibility of Maxmapper, a user would need to run both simultaneously, hence our motivation to integrate approaches. Vizmapper's "whole network" visualization is mimiced in the HiveView visualization for MapperGUI, described in Section 4.3.4.

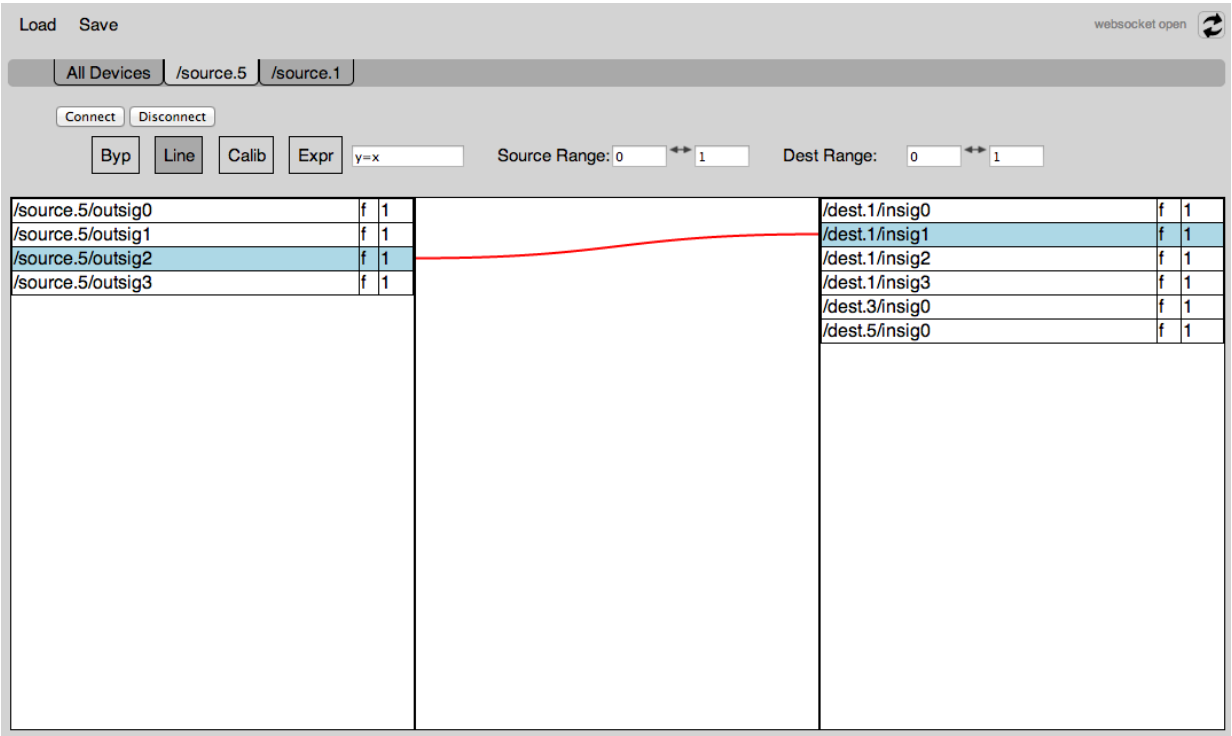


Fig. 3.4 The Webmapper interface

3.5.3 Webmapper

Work on this project began with a moderately featured, little used GUI for libmapper known as "Webmapper." Webmapper was created at IDMIL as an attempt to replace the Max/MSP GUI as the result of limitations described in Section 3.5.1, the principle among which being the cross-platform incompatibility of Max/MSP. It was thought that a browser-based approach would greatly simplify the process of creating cross-compatibility with all major operating systems and even mobile devices.

Webmapper utilizes the Python bindings for libmapper by registering an administrative monitor to communicate with a libmapper network. The monitor can create and modify connections or links, as well as query the network as to what devices, signals, links and connections are present. The Webmapper code creates a simple HTTP server and attempts to open Google Chrome[10] on the host computer. If Google Chrome is not present, the user must navigate directly to the server using the web address `localhost:50000`. The monitor

[10]Chrome Browser. [Online]. Available: `https://www.google.com/intl/en/chrome/browser/`. Accessed July 17, 2013

communicates with the libmapper network and the local server, the browser is able to see messages the monitor "posts" to the server (such as 'new device') and respond to them appropriately. The browser in turn can send messages to the server (like 'connect') that will propagate up to libmapper itself, eventually resulting in a message cascading back down to the browser reflecting the change to the network (such as 'new connection').

The interface itself is written for a web browser using the scripting language JavaScript[11] to control web-standard HyperText Markup Language (HTML) elements and Cascading Style Sheets (CSS). The JavaScript code stores four main data structures: devices, links, connections and signals. The code never directly modifies any of this data, and instead waits for messages relayed from libmapper. For example: if a user creates a new link, Webmapper does not add the link directly to the links array but simply sends a message to the network. If it receives back a 'new link' message, only then does it add the new link to the array. This is done to ensure that the data structures within Webmapper always reflect what is actually present.

Figure 3.4 displays the look of the interface before this project began. Users are able to perform all libmapper functions: connecting, linking and modifying connections, but only the simplest of feature sets is included. In order to form a connection the user must click on a source signal, click on a destination signal and then click on a button labeled "connect." Many useful features of Maxmapper, such as column headers, table sorting, drawing connections and search filtering are not present.

3.6 Evaluation of libmapper Variables as Visual Data

In order to examine different possibilities for visually encoding libmapper data, we have compiled a list variables and their categories as described by Mackinlay (1986). The list in table 3.1 is by no means a complete set, as libmapper may yet expand to include data like device position and users are able to tag devices/signals/links/connections with any extra data they may want.

A fourth data category *boolean* has been added to specify data that only has two values (*true or false*), as it is a common metadata feature. Boolean information is not covered in the Mackinlay paper. Going forward it will be treated more or less like ordinal data.

[11] JavaScript — MDN. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. Accessed July 17, 2013

Table 3.1: libmapper metadata types

Devices			
quantitative	*ordinal*	*nominal*	
number of inputs	device ordinal	device name	
number of outputs			
ip address			
port			
Signals			
quantitative	*ordinal*	*nominal*	
length	direction	parent device name	
minimum value		signal name	
maximum value		data type (float, integer, etc.)	
sampling rate		units	
Links			
quantitative	*ordinal*	*nominal*	
		link name	
		source device name	
		destination name	
		scope	
Connections			
quantitative	*ordinal*	*nominal*	*boolean*
source minimum	instance number	boundary modes	mute
source maximum		connection mode	send as instance
destination minimum		destination data type	
destination maximum		mute	
		expression	

Chapter 4

Design & Implementation

The development of a graphical user interface for libmapper creates a unique challenge. Obviously such an interface is a practical tool, and yet it must also work in concert with DMIs, which are inherently designed for creative use. For the purposes of this project, the assumed solution to this innate paradox is to provide the user with multiple independent control modes. libmapper itself is an extremely flexible API that makes few assumptions about the network of devices and signals or how they are mapped. It is thus fitting that a GUI for libmapper would be equally flexible. In lieu of a single perfect solution for network visualization and interactivity, providing users with various independent views offers a good compromise.

Work on MapperGUI began with the Webmapper interface described in Section 3.5.3. An MVC structure was built around the code to make the program more extensible and to easily integrate of multiple views. Missing features from Maxmapper were incorporated into the main view mode, ListView. ListView was extended in various ways, taking advantage of the new code base. Two new view modes, GridView and HiveView,[1] were then integrated into the main GUI. Finally, the code was compiled together as a standalone application ready for wide distribution.

[1]Both designed by Jonathan Wilansky at the IDMIL

4.1 Development of a Flexible System

Prior GUIs for libmapper have been used successfully for some time, but all have failed to become a standard for the same reason: they cannot accommodate all possible use-cases of libmapper. List based views like Maxmapper and Webmapper do not show hierarchies, while the cluster view implemented in Vizmapper can be overly cumbersome for interaction with simple networks. With so much work already completed on prior GUIs, it was more suitable to integrate different approaches into a single GUI, rather than to begin work on some new, hopefully superior approach that would likely prove to be flawed like all others that came before.

MapperGUI integrates multiple via a drop-down menu on the upper corner of the window. Options on this menu represent available visualization modes. By selecting a new visualization mode the GUI drastically changes its appearance, replacing nearly every visual element in the display.

4.1.1 MVC architecture

Because we require a modular design, the Model-View-Controller architecture[2] was used as a general framework for structuring the application. In fact, the whole-scale swapping of independent visual modes is a very straightforward implementation of MVC. Unfortunately, the `libmapper` → `pythonmonitor` → `browser` implementation complicates matters slightly.[3] A few layers of abstraction are added to take into account the monitor, the network itself and control features independent from the view (see Section 4.1.2), but the general MVC architecture is maintained.

Independent communication

First and foremost, it is essential that data on the screen reflect data on the network. This is not entirely straightforward, as asynchronous messages are constantly relayed between MapperGUI and libmapper. In a truly distributed system, data on the libmapper network changes continuously as other users add devices and modify mappings. Our system insulates the actual libmapper network, the displayed data and user interaction elements from one

[2]Described in Section 2.2.3.
[3]Compare Figure 4.1 with Figure 2.8.

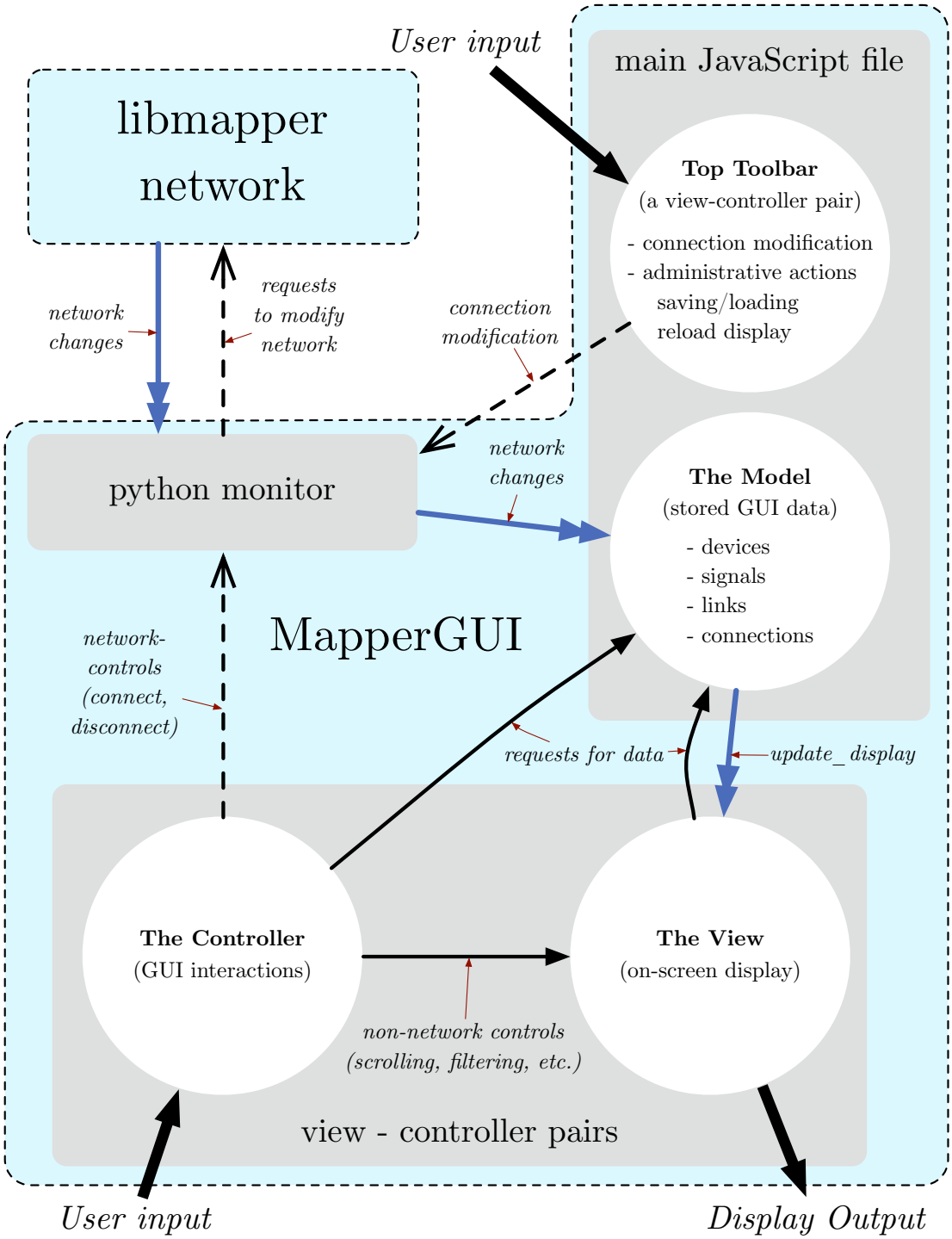


Fig. 4.1 Structure of MapperGUI. Blue arrows show propagation of network changes, while dashed arrows denote messages requesting a network change.

another (Figure 4.1). For example, a user command to link the two devices `source.1` and `dest.1` will cause the controller to send the following message to the python monitors:

```
{"cmd":"link","args":["/source.1","/dest.1"]}
```

Meaning: a linking command is sent to `source.1` and `dest.1`.[4] After this, the display does not change, as it has not yet been notified of a new link. The monitor then relays this message to the libmapper network. If the link is successful, the monitor receives notice, and sends a message to the model:

```
("new_link",{"src_name":"/source.1","dest_name":"/dest.1"})
```

This states that a new link has been formed between the source device `source.1` and the destination device `dest.1`. Only then does the GUI respond to the change on the network. Signal data itself is not available to MapperGUI in any way, as libmapper networks are designed to prevent this kind of bandwidth clutter (Malloch et al. 2013).

The model

The model consists of an abstract copy of the libmapper network. Independent views can consult these data, but cannot directly modify it. Messages from the python monitor announce new links, modifications to connections, or any other changes on the network to the model which records these changes into four data structures:

- `model.devices`: Storage of all present devices and device metadata.
- `model.links`: A record of all links on the network.
- `model.signals`: Monitors signals on the network, but only those that are currently visible in the GUI. This is done to save bandwidth and processing power.
- `model.connections`: All connections and connection metadata between signals currently in the model.

It is possible that previously viewed signals will persist in the model, but their connections will not be updated.

[4] The message itself is a python dictionary.

View-controller pairs

All interaction handlers (responses to mouse clicks and key presses) and visualizations are stored in modular, view-controller pairs, as recommended by Krasner and Pope (1988). Each view-controller pair corresponds with a single view mode. Pairs can have any combination of UI handlers and visual features, but must implement the following four functions:

- `view.initialize()`: Calls upon the view to create its visual elements and add its individual interaction handlers.
- `view.get_focused_devices()`: Returns whichever devices are currently visible in the view. This is used for populating the `model.signal` and `model.connection` data structures, as well as for saving and loading.
- `view.cleanup()`: Causes the controller to remove all interaction handlers.
- `view.update_display()`: Called whenever the model changes. The view is not made explicitly aware of *what* has changed, but only that a change has occurred. In each view mode, this call causes visual features to be cleared and re-drawn. Though this creates more processor overhead (see Section 5.2), it allows for much greater flexibility in designing new views. The model does not need to be aware of any specific informational requirements for each view.

4.1.2 Top toolbar

It is sensible to include certain tasks and information providing structures across visualization modes. In light of this, a single view-controller pair runs continuously in MapperGUI in the form of a toolbar at the top of the window. As a part of the code structure, it communicates independently with the monitors and other view-controller pairs. This toolbar contains all administrative controls and connection modification fields.

- **Administrative controls**
  - *Load/Save Buttons*: These elements respond to clicks to save and load mappings, as discussed in Section 4.4.1.
  - *Visual Mode Selection*: A drop-down menu containing all view modes for user selection.

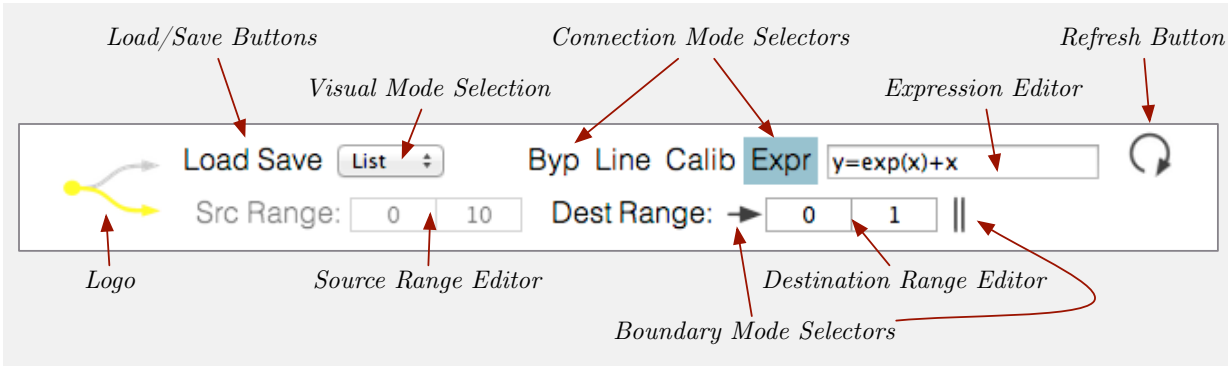


Fig. 4.2 The upper toolbar

- *Refresh Button*: When clicked, all data residing on the model is erased and re-gathered. This is useful if the monitor somehow desynchronizes with the network.

- **Connection modification**: The following controls are only available when the user selects a single connection.
  - *Connection Mode Selectors*: An array of buttons allowing the user to choose between available connection modes.
  - *Expression Editor*: Here the user can input a custom expression if the selected signal is in the *Expression* mode. In other connection modes this field displays the connection's expression but is not editable.
  - *Source Range Editor*: These two numbers display the maximum and minimum values of the input signal. These fields is only editable in the *Line* connection mode.
  - *Destination Range Editor*: Same as above but for destination signals. Due to boundary conditions these fields are useful in all modes.
  - *Boundary Mode Selectors*: Two buttons that cycle through the five boundary modes for the maximum and minimum destination values. A small graphic represents each mode.

All interface features not present in the top toolbar are part of the current visualization mode and reside in a "container" element below, occupying the remainder of the window.

The file and communication structure described in this Section allows for quick modification and extension of the interface. All components are modular, so developers can program new visual modes relatively easily. Hopefully this will eventually lead to a GUI with many useful view modes that can accommodate nearly every use-case for libmapper.

4.2 Integration of Interface Features

Development began by unifying features of the Maxmapper onto the Webmapper code. Webmapper was selected as a starting point because of the cross-platform nature of a web-based implementation. The general two-table structure of Maxmapper and Webmapper created the first view mode of the interface: ListView.

4.2.1 Structure of ListView

Of all currently available views, ListView provides the most straightforward way to visualize and interact with libmapper. Two tables dominate the visible area listing source elements on the left and destination elements on the right. Bézier curves sit on a central canvas and form lines between associated list elements on each side. Because these curves do not always represent the same data structure, the lines themselves are referred to as *arrows* by the GUI code and by this document.

The view itself is divided into two major modes: "All Devices" and linked devices. Switching between these modes is accomplished through tabs that appear at the top of the container. In the All Devices tab, ListView lists network devices, as in Figure 4.3. Source devices inhabit the left table, while the right table lists destination devices. Intermediate devices[5] will be listed in both tables. The view displays device metadata as columns of each table. Here arrows represent links between devices. Since no connections or signals are shown, most of the top bar (see Section 4.1.2) is disabled in the All Devices tab. Saving and loading are also disabled.

MapperGUI draws a tab for every source device with at least one link to a destination device. Clicking on any of these tabs will redraw both tables. The left table now shows all child signals for the selected source device, while the right table displays child signals for every destination device linked to that source. In this mode, arrows represent connections

[5]Devices with both inputs and outputs, such as implicit mappers described in Goudeseune (2002).

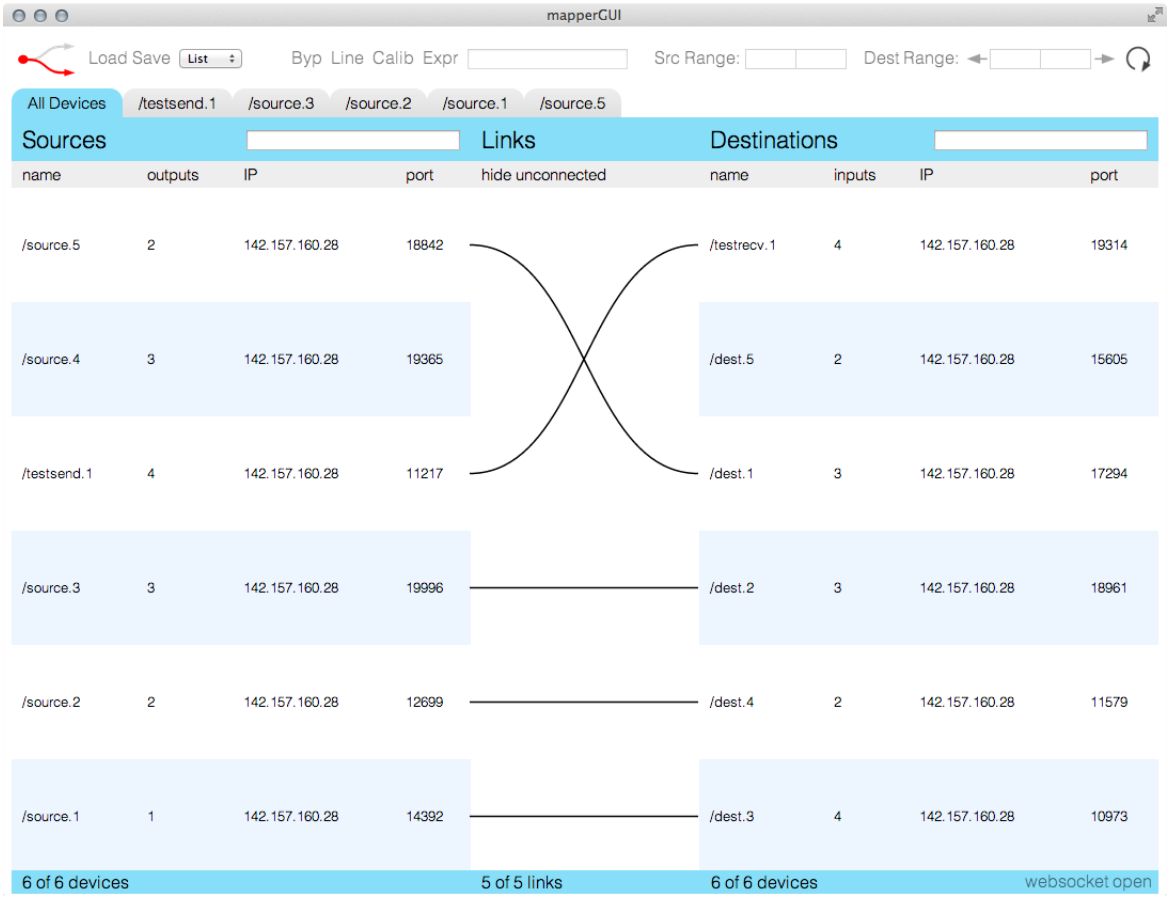


Fig. 4.3 ListView with all devices selected

that can be modified using the top toolbar.

4.2.2 Display libmapper metadata

Tables in the original Webmapper interface have no headers. Without these queues, only a small amount of metadata is provided (see Figure 3.4).

By incorporating a useful feature of Maxmapper, column headers have been added to the ListView. New pieces of device and signal metadata are also included, as listed in table 4.1. Tables support additional metadata, as they are filled by a generic function that will include any data found in the model.

In general, MapperGUI tries to keep possible extensions to libmapper like this in mind. Very little is assumed about the network itself. In turn, the only device metadata that *must*

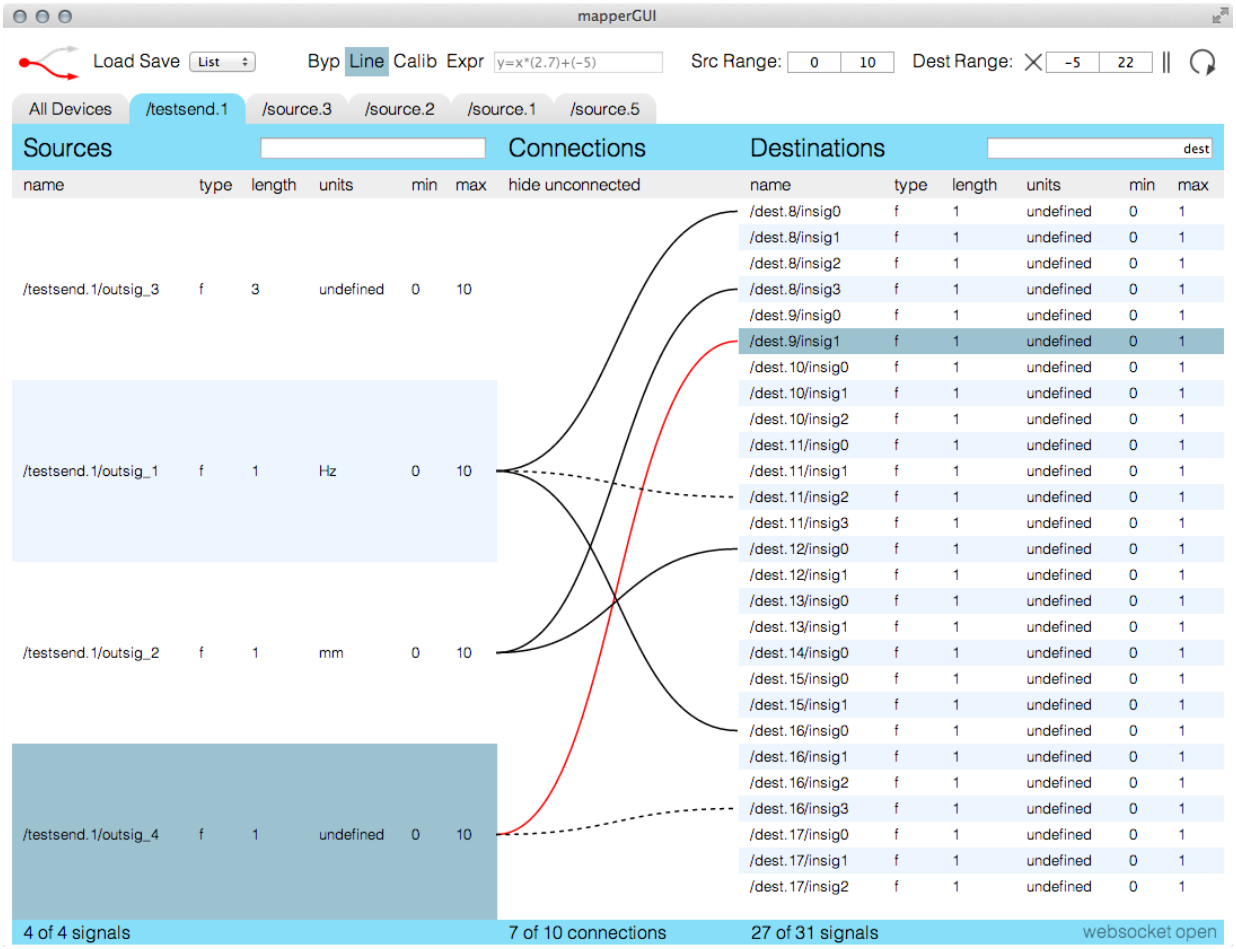


Fig. 4.4 ListView with device **testsend.1** selected

exist is the device name and number of inputs/outputs. MapperGUI uses the number of inputs/outputs to place a device into either the source or destination tables. For signals, MapperGUI takes vector length into account when deciding whether two signals are compatible and can be connected. However, not including length in the signal metadata will not result in an error.

4.2.3 Locating devices and signals

In networks with lengthy arrays of devices or signals, it can be difficult to find particular objects. Three features from Maxmapper were adapted for use with ListView to aid in such tasks.

Table 4.1 Metadata available in Webmapper versus ListView

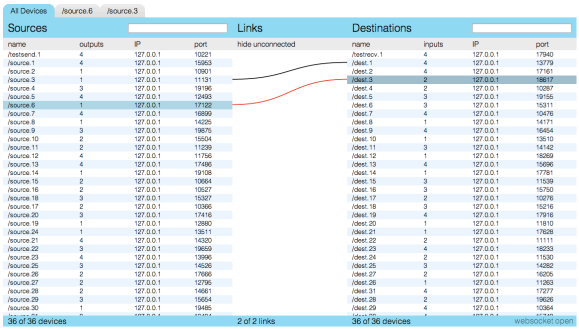
webmapper		**ListView**	
Devices	Signals	Devices	Signals
name	name	name	name
IP address	data type	IP address	data type
port	vector length	port	vector length
		number of inputs	units
		number of outputs	maximum value
			minimum value

First, regular expression[6] supporting search-bars are now present at the top of each table. If the user types an expression of any kind into either of these fields, ListView will filter elements displayed in the table beneath. Table rows can be filtered not just by the names of the signals/devices, but also by length, units, IP address or any other piece of information in the table row. To make the filter more responsive the code runs with every key press such that the table is dynamically modified as the user inputs characters to the search field.

A common use-case for Maxmapper was a large list of signals with only a handful of connections that needed constant modification. Obviously scrolling through a list with hundreds of rows, only to repeatedly select between the same handful of connections can be very tedious. To assist these users Maxmapper features a "hide-unconnected" button that was incorporated into ListView as well. The button sits in a previously unused piece of screen above the central canvas (see Figures 4.3 and 4.4). When clicked, the GUI hides all signals not currently connected to any others, the text on this button then changes to "show-unconnected."

Finally, tables can sort themselves by individual columns. Signals and devices are initially placed into the table in whichever order they appear in the model. Upon a click to any column header, the table sorts the information "descending" (lexicographically) by that column. A second click on the same header will re-sort the information in an "ascending" fashion. Users can sort table rows by any column appearing in the table.

[6] TODO



(a) list view with 72 devices and 2 links

(b) same network, filtered using "hide unconnected" button

Fig. 4.5 Functionality of hiding unconnected elements on a network with many devices and few links.

4.2.4 Visual feedback

User feedback was a very important part of the design process (see Section 5.1). One observation re-iterated by nearly all users of MapperGUI and prior GUIs was that it became extremely frustrating when the display became out of sync with the network or when it seemed like the GUI *might* be out of sync. To ameliorate these difficulties, MapperGUI incorporates a few Maxmapper features entirely for visual feedback.

At the very bottom of the window is a bar displaying the number of elements on the network versus the number currently visible. For example, if there are 36 source devices on the network, but the user has filtered out all but two, then the field below the source table will read “2 of 36 devices” (as in Figure 4.5). These data are also shown for destination devices, links, connections and signals. This is done in order to help the user diagnose technical problems. If a desired signal does not appear, perhaps the device has become unresponsive or the user has encountered an error in MapperGUI. If the user has simply filtered out the signal somehow, it is much more straightforward to see this immediately than to begin searching for possible technical problems.[7]

The top toolbar automatically reflects metadata for selected connections. Expressions, connection modes and ranges can be observed simply by clicking on the arrow representing a connection. The toolbar displays non-editable fields (depending on connection mode) as slightly more transparent. Arrows are re-drawn with dashed lines for muted connections, as in Figure 4.4.

[7]libmapper and MapperGUI are both at a stage of development where bugs are an inevitable part of the user experience. We like to call them “features.”

Large tables are more easily navigable when rows have "zebra" striping. The display re-calculates this alternate row striping any time a user filters the view. Rows highlight themselves when selected. Any number of rows on either table can be selected simultaneously. Row highlighting works in combination with row striping (see Figure 4.6).

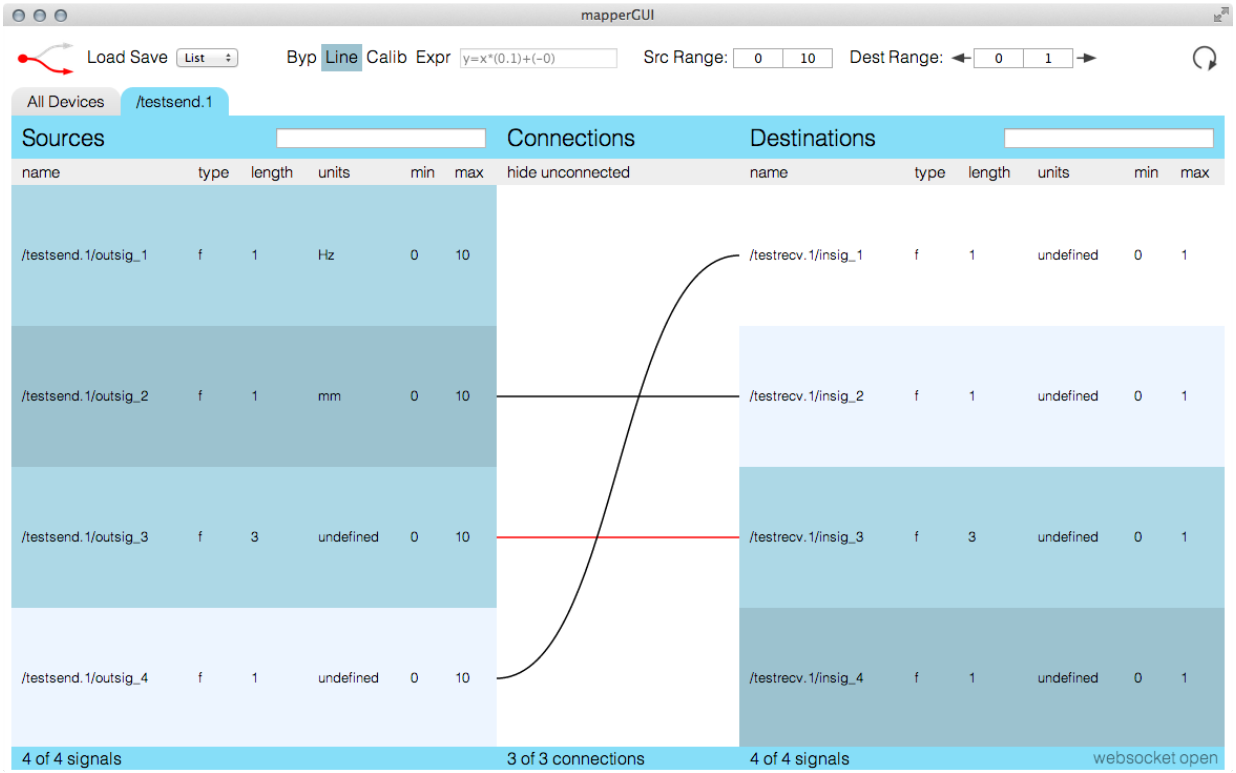


Fig. 4.6 Multiple selection and row striping in ListView.

By incorporating popular visual feedback elements from Maxmapper, we were able to make the display more robust and useful. Though difficulties with interaction can still occur (missing devices, unresponsive connections, etc.), good visual feedback should allow users to more quickly diagnose and solve these problems.

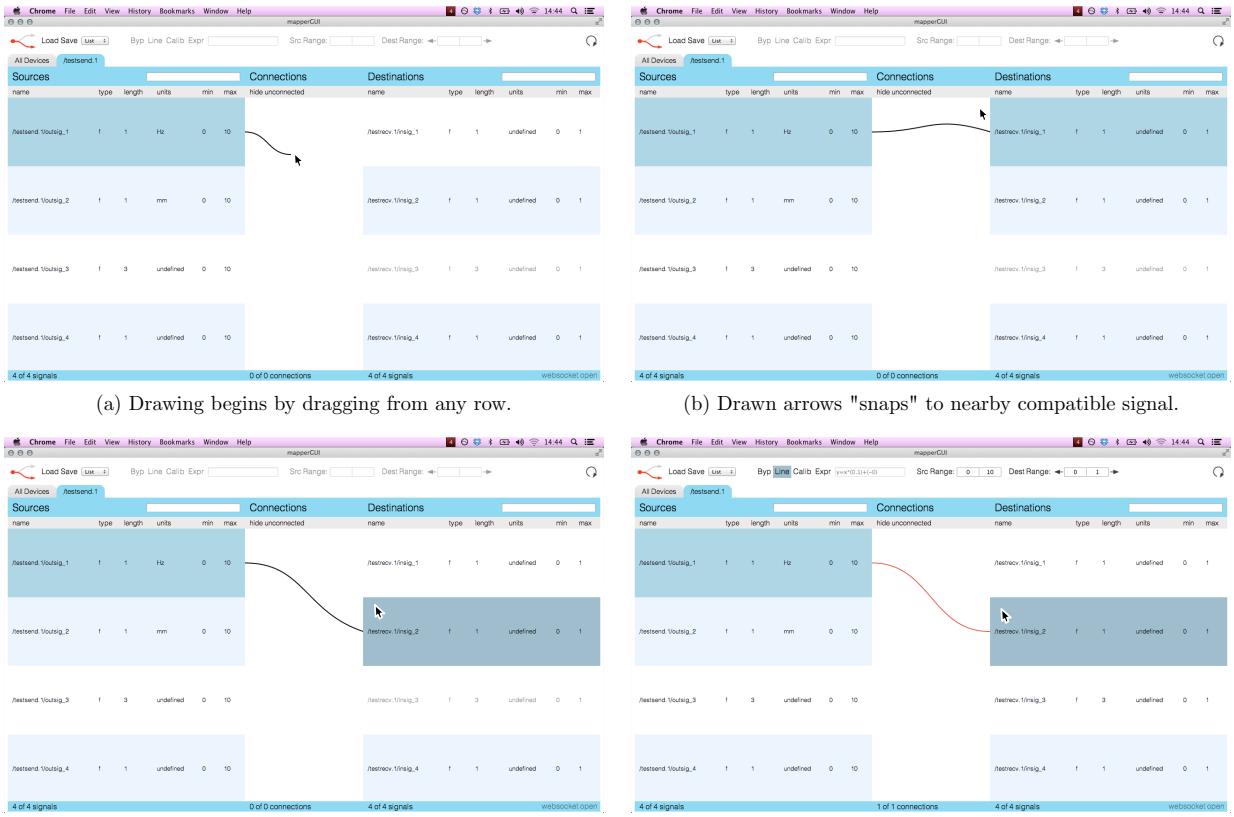
4.2.5 Improvements to user interaction

The most common user complaint about Webmapper was the nature of its interaction. In order to form a connection, a user must click on a source device, then a destination device and then finally click a "connect" button. Even for simple mappings, this was seen as

overly cumbersome. In order to make MapperGUI useful, we clearly need to improve on interaction speed. Fortunately UI features in Maxmapper help solve this problem.

Draggable links and connections

The drag-to-connect-gesture is common among similar interfaces (Robillard 2011, Bullock et al. 2011) and it is featured in Maxmapper as well. Though more advanced to program than the improvements listed above, it was seen as necessary to get libmapper users to switch from Maxmapper to our GUI.



(a) Drawing begins by dragging from any row.

(b) Drawn arrows "snaps" to nearby compatible signal.

(c) Drawn arrow snaps to desired signal.

(d) Connection is formed on mouse up.

Fig. 4.7 Draggable links and connections.

The user can click on any table row and drag onto the central canvas. Upon doing so, a slightly thicker Bézier curve begins to follow the mouse pointer about the canvas. Incompatible signals become transparent. Once the mouse pointer comes within 50 pixels of the other table, the drawn arrow snaps to the nearest row if it is compatible, highlighting

that device or signal. The user can then scroll the mouse up and down the rows of the target table and the drawn arrow will continue snapping to the nearest available row. Once the user releases the mouse button, MapperGUI sends a message to libmapper asking either to connect the appropriate signals or link the appropriate devices.

ListView does not draw the final linking/connecting arrow until a confirmation message is received from the monitor by way of libmapper itself. Figure 4.7 demonstrates a dragged connection starting from a source signal and ending on a destination signal, though the same gesture is possible beginning with destination elements.

Keyboard shortcuts

To further accelerate GUI operations, some keyboard shortcuts were added:

Table 4.2 Shortcut keys in ListView

key combination	action	from Maxmapper?
c	Connect/link selected rows	no
delete	Disconnect/unlink all selected	yes
command + a	Select all visible connections/links	yes
alt + tab	Change tab to the right	no
alt + shift + tab	Change tab to the left	no
m	Mute all selected connections/links	yes

For PC users, the "select all" key command is "control + a." Tab changing is meant to further mimic functionality of web browsers.

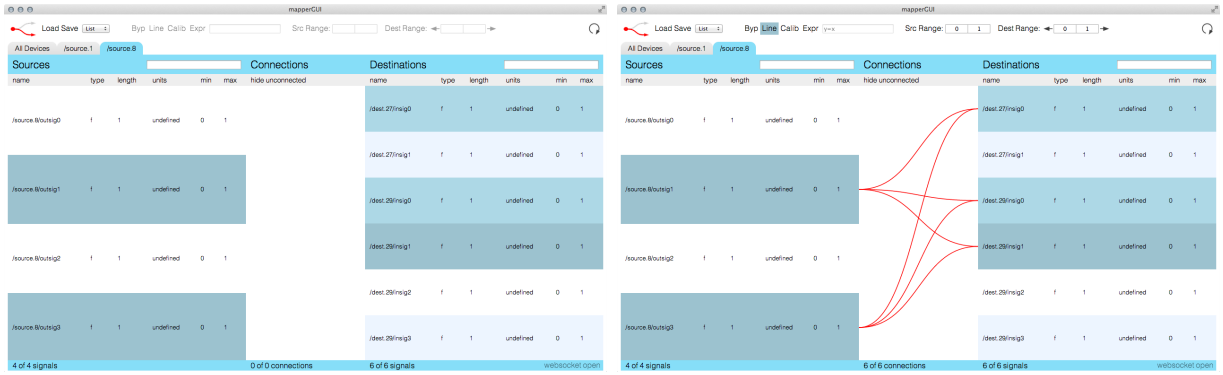
4.3 Extension of Control and Visual Elements

With the new web-based framework up and running, it is fairly easy to extend interface features beyond that of Maxmapper. Requested features that would have been very difficult to implement in Max/MSP were added to MapperGUI. Also, two new view modes were created, taking advantage of the modular, MVC-style code base.

4.3.1 Multiple selection

Unlike in Maxmapper, it is possible in our GUI to select table rows with a mouse click. Any combination of rows can be selected on either table. This allows for multiple signals

or devices to be connected/linked simultaneously by pressing the 'c' key. This particular command connects all selected source to all selected destination elements.



(a) Multiple rows selected (b) After pressing the 'c' key

Fig. 4.8 Simultaneous connection of multiple signals

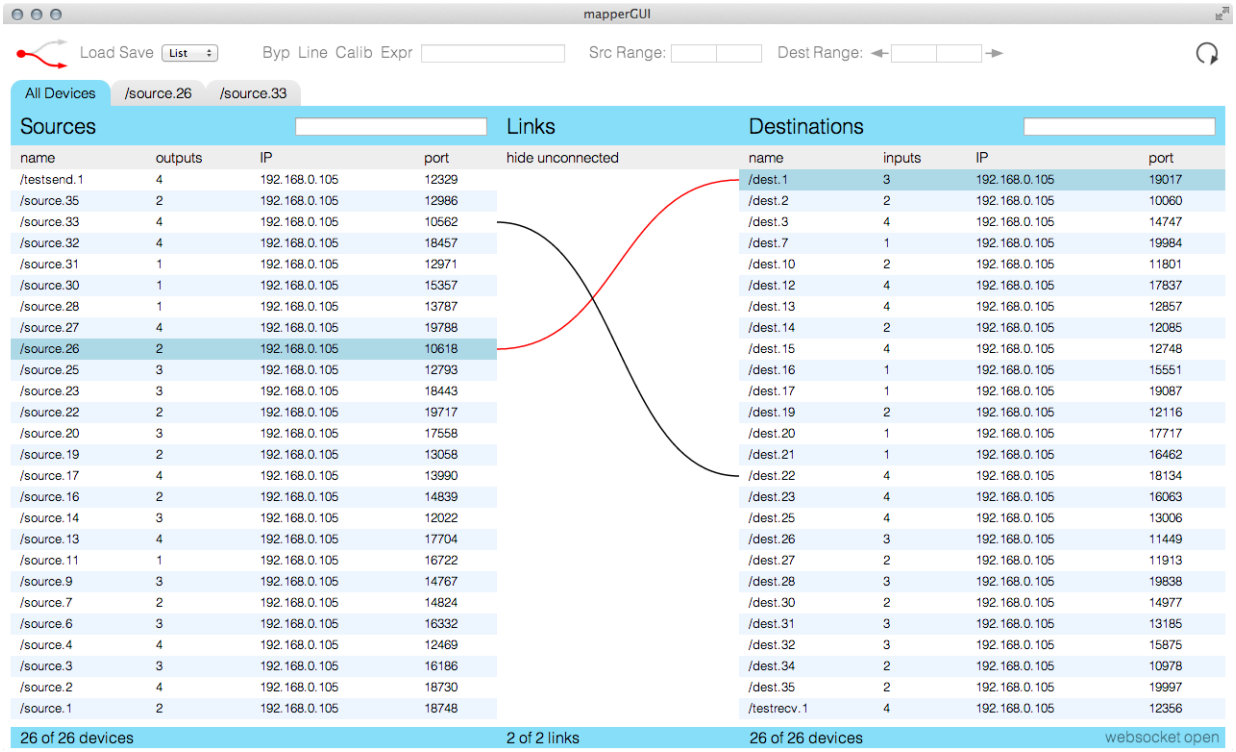
Users can also depress the "shift" key to select multiple rows simultaneously, a functionality common in other list interfaces, like the Windows and Macintosh file browsers. Clicking anywhere in the container, except for the tables, deselects all currently selected rows and arrows.

4.3.2 Accommodating varying window sizes

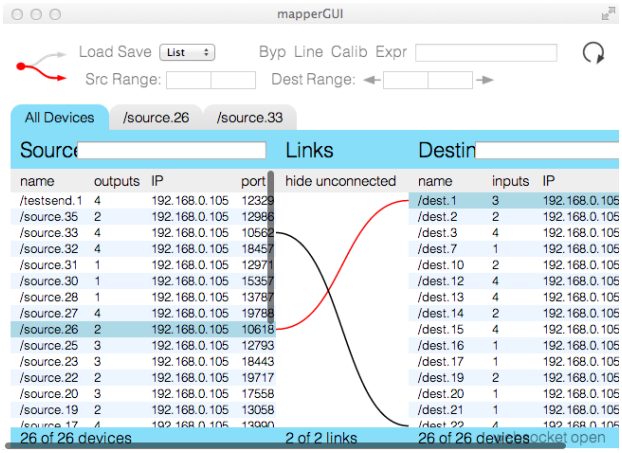
One notable shortcoming of the Maxmapper GUI is its inability to resize the application window. This creates problems for users with small screens, or those who would like to run Maxmapper side-by-side with other applications. MapperGUI can be resized in the same fashion as any other application, supporting windows as small as 100 x 124 pixels (about 3cm x 3cm).[8]

Upon resizing, the size and shape of various on-screen elements change dynamically to fit the new window size (see Figure 4.9). The two device/signal tables always occupy two-fifths of the container area each, with the central canvas filling the remaining fifth. The container itself fills the entire window not occupied by the top bar. It will expand to fill any size, but has a programmed minimum height of 150 pixels (about 4cm) and minimum width of 700 pixels (about 18cm). Upon hitting these minimum dimensions, the GUI adds

[8] All physical screen sizes quoted in this Section are for a 72 pixel-per-inch hi-res display. Lower resolution displays will result in larger windows.



(a) List view at 1280 x 760 pixels



(b) Same view resized to 650 x 450 pixels

Fig. 4.9 Resizing the ListView window. Rows condense, scroll bars appear and the top menu collapses in the smaller version.

scroll bars to allow the user to view the entire display. Elements within the top menu fold onto multiple lines to accommodate narrower windows.

Maxmapper table rows have fixed height. Unless many devices or signals are present large parts of the display are often empty. ListView instead calculates table elements to fill the available space, as can be seen in most of the figures of this chapter. Minimum row heights are set to 17 pixels. Once there is not enough space to accommodate all necessary rows, MapperGUI adds a scroll bar to the appropriate table.

4.3.3 Visual redesign

Keeping in line with visual guidelines summarized in Section 2.2, we overhauled the look of the Webmapper interface to reduce visual noise, make better use of color and generally improve its aesthetic appeal.

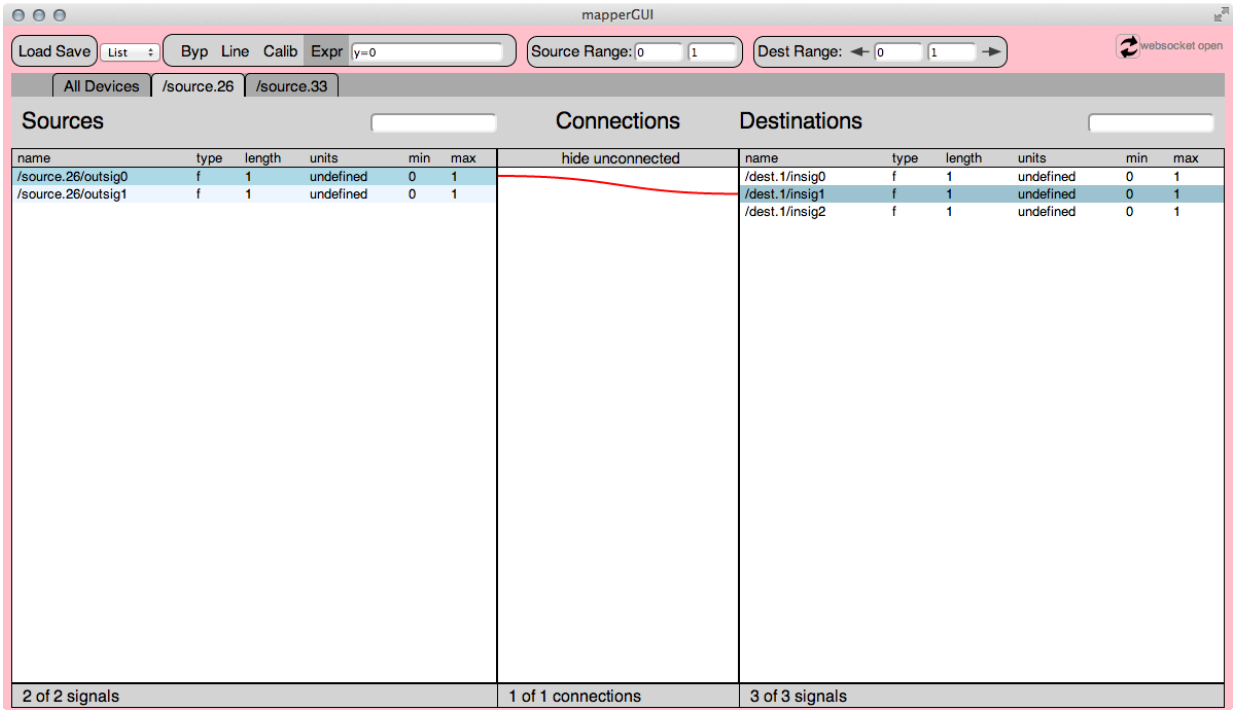


Fig. 4.10 ListView before visual redesign

First, much of the display area was wasted for simple networks, leading to the dynamic row sizing described above. The display was plagued by $1 + 1 = 3$ noise, causing negative space like the central canvas to attract the eye and making the display seem much more

complex than necessary. All black borders were removed and font weight was lightened for all text, drastically reducing visual noise. Arrows now display with one-half of the stroke weight. This too reduces visual clutter and also differentiates arrows that are in the process of being drawn.

The pink background, though whimsical and popular at the IDMIL, was deemed too bright to be used effectively over such a large area. It also distracted from the red color used to highlight selected connections and links. A neutral white was selected for the background, both to blend with input areas and to cause more contrast with row striping and highlights. Aside from the red for selected arrows, all colors are now a variation of an unobtrusive gray-blue. This contributes to the visual uniformity of the display, but also allows us to make visual distinctions between odd rows, even rows, selected rows, table headers and table footers without using borders.

Finally, a logo was added to the upper left-hand corner of the display. The logo is a simplified version of the overall libmapper logo[9] with a white background. The red color for highlighting is maintained to match highlighted links and connections.

4.3.4 Alternate views

Jon Wilansky, a fellow master's student at the IDMIL, created two new views for MapperGUI. These took advantage of the new MVC architecture for the program. With these views in place it finally became possible to test our foundational hypothesis that a variety of displays would aid in mapping tasks.

GridView

First programmed and implemented was the "grid" view. The network is represented with two m-by-n grids. The leftmost grid lists source devices on the horizontal axis and destination devices on the vertical axis. Links are formed by clicking on the square at the intersection of the desired source and destination devices. The second grid represents signals and connections. Signals must be explicitly added to this grid from the device grid by selecting the intersection or labels and clicking "add", or by pressing the "a" key.

Both grids are fully filterable using text input fields below, which borrow code from ListView. Grids are also zoom-able using the endpoints of the scroll bars. Either grid can

[9] Can be seen at `www.libmapper.org`

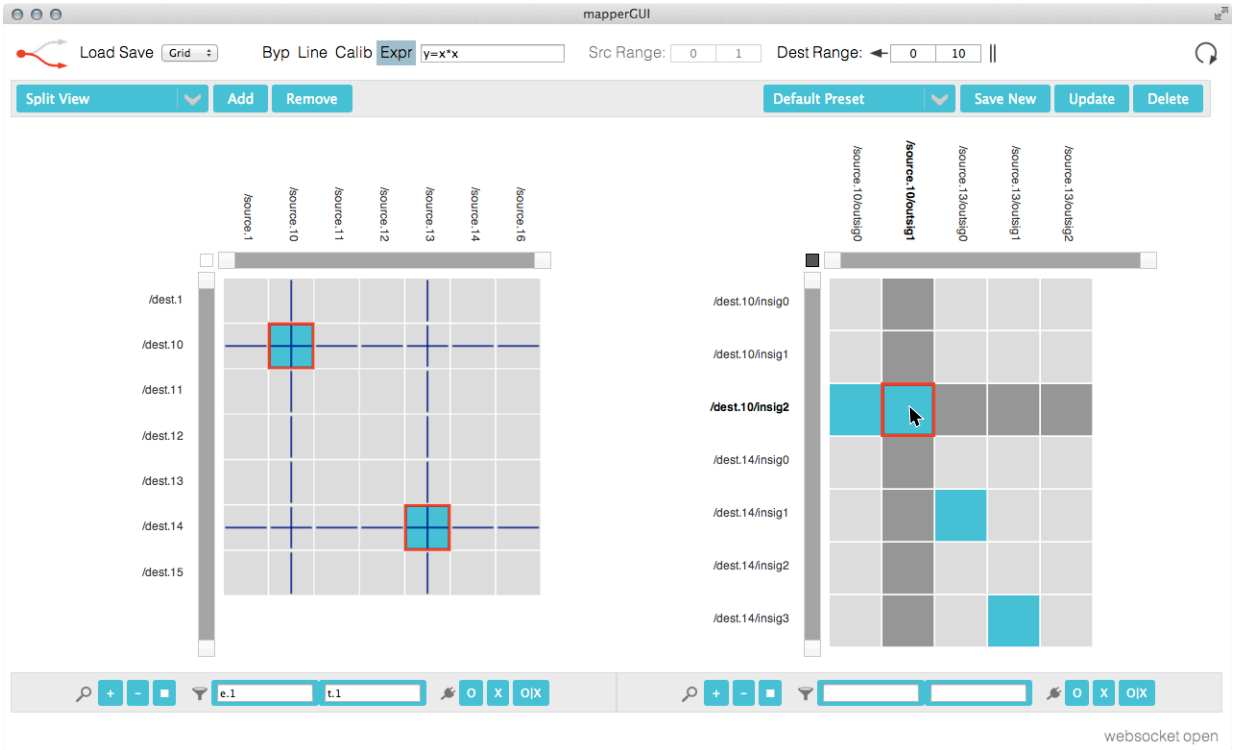


Fig. 4.11 GridView

be hidden, allowing the user to focus the entire display on a single grid. Because this view is more customizable than ListView, the designer added an option to save view settings, causing GridView to remember which devices have been added to the signal grid.

GridView provides visual feedback by highlighting the associated row and column when the user places the cursor over a grid intersection. Text of the relevant devices/signals is also highlighted in this situation. Colors are designed to match the gray-blue style in ListView, hopefully creating the feel of a unified interface for MapperGUI. GridView highlights selected grid squares with the same red color as the one used in ListView and the logo.

The top bar looks and functions in the exact same fashion for GridView as in ListView. MVC architecture allows us to create modular view-control elements like this to be used with a variety of other view-controller pairs.

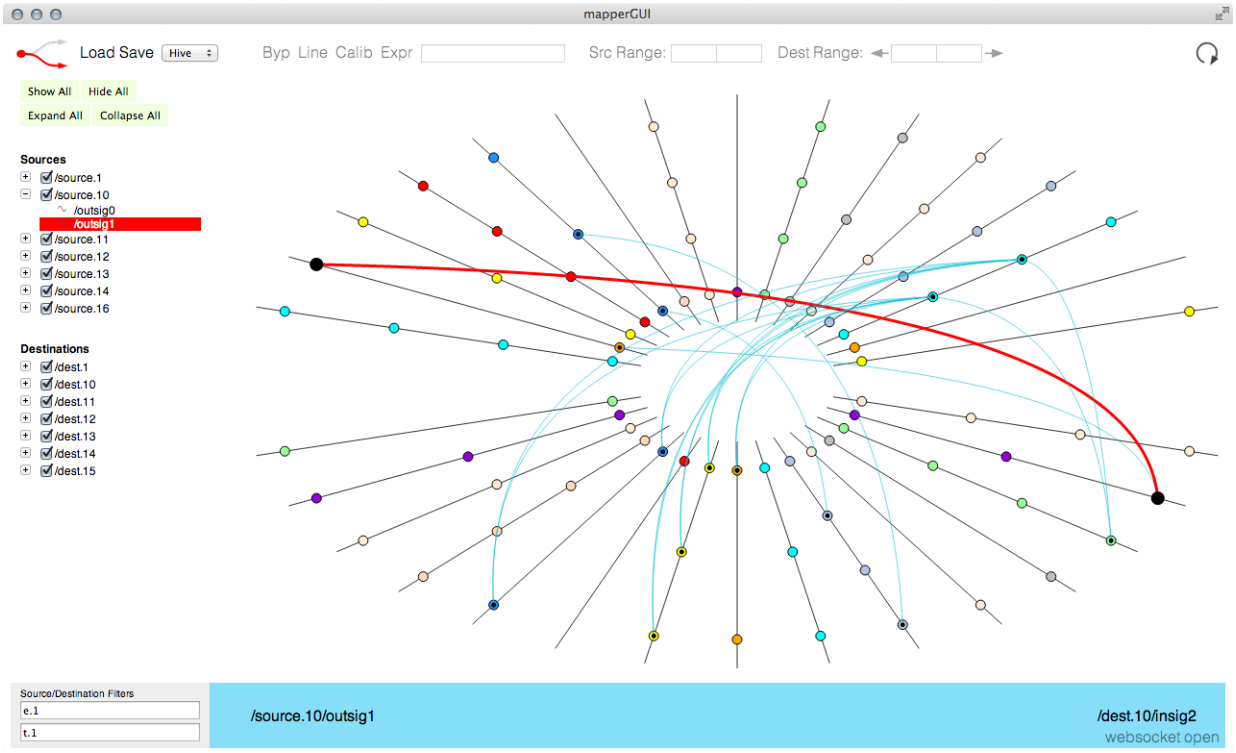


Fig. 4.12 HiveView

HiveView

The "hive" view attempts to address the problem of visualizing entire networks simultaneously. This visualization borrows many techniques from Vizmapper (see Section 3.5.2). Solid black lines emanating from the center of the view signify network devices. Source devices are placed in the top half of the display, destination devices on the bottom. Small circles representing child signals are distributed throughout each line. Thin blue curves flow between these circles, signifying connections.

On the left side of HiveView, a menu displays an expandable and collapsible list of all network devices. Expanded entries in this list display child signals. Connection lines highlight to red when clicked, and the bottom bar (colored our standard blue) presents the names of the connected signals. Placing the mouse cursor over device lines highlights all connections to that device. The same is true for mousing over individual signals or any item in the list on the left.

A text filter in the bottom right will filter the namespace on the left side of the screen.

Connection lines can also be hidden by toggling check-boxes in the device list on the left.

HiveView is not yet as interactive as the other two views. It is not possible form connections or links by any means, though a dragging-type interaction like the one in ListView would be most desirable. It is possible to modify selected connections using the top toolbar, as the MVC structure preserves this functionality across views.

4.4 Other GUI features

4.4.1 Saving & Loading

Saving and loading presents an interesting problem for libmapper networks. Upon clicking the "save" button, connection information is serialized into a JSON[10] file that the user is asked to name. Only visible connections, ones that are present in the selected tab in ListView, are recorded to the save file. Saving is not yet fully supported for GridView or HiveView.

A "naïve" interpretation of loading mappings is implemented here. Though device information is encoded into the save file, it is not considered in the loading process. Mappings are loaded for all applicable signals. For example, in the following situation:

- Devices `tstick.1`, `tstick.2` and `granul8.1` all exist on the network
- Both `tstick.1` and `tstick.2` have child signals named `raw/accelerometer/1/x` (they both should, as they are both t-sticks)
- A mapping is loaded that contains a connection `tstick.1/raw/accelerometer/1/x` → `granul8.1/filter/envelope/frequency/low`

For the above case, the single connection will be loaded for *both* t-sticks, creating two total connections to `granul8.1`'s low filter envelope. If two instances of the granul8 synthesizer exist, then the connection will be loaded four times, one for each iteration of t-stick → granul8.

This naïve implementation is used to maintain a modularity for mappings such that similar devices and equivalent devices with different names can share mappings. This

[10] JavaScript Object Notation. A human-readable data-interchange format. [Online]. Available: `www.json.org`. Accessed July 30, 2013

makes sense for libmapper networks, as they are ideally collaborative, and ordinal numbers appearing after device names are arbitrary.

4.4.2 Creation of a standalone & distribution

Though it is already in use, we would like for MapperGUI to be quickly adopted by a greater number of users. This would assist us in debugging and improving the interface. It would also hopefully bring a new set of users to libmapper itself, encouraging use of the API and its implementation into new DMIs. Unfortunately, to get libmapper and MapperGUI up and running from the source code requires installing package management software and compiling multiple dependencies. This is rather time-consuming and well beyond what should be expected of a non-programmer.

To aid in ease-of-use and adoption, we compiled a "standalone" version of MapperGUI. It is presently available for free download at `www.libmapper.org/downloads` under the name Webmapper[11]. The standalone includes libmapper code and can therefore be run on a machine that does not have libmapper explicitly installed. To the user, it looks like any other Macintosh application: an icon on which to double-click.

The current version is still very much in test phase and includes a readme file describing likely bugs and the non-ideal startup method.[12] It presently supports Macintosh OSX only, though Linux and Microsoft Windows releases are planned.

[11] The name has not yet been officially changed.

[12] Running the program and navigating to `localhost:50000` in the Google Chrome browser.

Chapter 5

Applications & Discussion

This chapter presents a discussion of MapperGUI's software design and its consequences for musical mapping, as well as revisions made to the code since the initial release. The interface's features are explored in an attempt to evaluate the successes and failures of the design. Feedback from users was gathered throughout the project and through informal interviews after the software's release. This feedback is summarized and presented here. A modification to the code, motivated by feedback from users, is also described. MapperGUI is then compared to similar interfaces, analyzing especially for new features that could be incorporated into our flexible framework. Finally, the system is evaluated overall with respect to the project's initial goals.

5.1 User Feedback

The entire MapperGUI project began with user feedback from prior GUIs for libmapper. Throughout the design process functional versions of MapperGUI were provided to libmapper users. Their feedback was crucially important to the evolution of the software. After the first official release of MapperGUI, long-term users were informally interviewed. These users were questioned specifically as to the projects with which utilized MapperGUI in an attempt to learn more concretely about the variety of use-cases for the software.

Even at this early stage of release, users have already incorporated MapperGUI into a wide variety of projects. This reflects our initial assumptions (based on experiences with prior GUIs) that a successful GUI must be flexible. Throughout development MapperGUI was used as an experimental tool and aid in designing DMIs. MapperGUI was used in con-

cert with motion capture systems, vibrotactile feedback and even loaded onto a Raspberry PI[1]. During this whole process users encountered problems, had ideas for extensions and used the GUI in ways we could have never imagined.

5.1.1 General feedback

Most of our users had used libmapper previously, and had attempted to compile and use the library from scratch. Many commented on how well MapperGUI lowered the barriers to entry for non-technical users. Users who had never used libmapper before pointed out how much time had been saved in their work flow, versus when they used to hard-code mappings.

The best reviewed feature of MapperGUI was the automatic linear scaling control found in the top bar. Some users previously needed to detect signal minima and maxima by hand, then directly calculate and apply a linear scaling functions. With MapperGUI the task is trivially easy: one must simply enter the desired destination range and set the connection to the *Calibrate* mode. Most of the "magic" in this feature is the result of the libmapper API, but providing users access through an easy-to-use GUI is also important. One user expressed frustration because she was not aware this feature existed, and instead continued to painstakingly condition her signals in Max/MSP. She was very impressed over how much time was saved by switching this workload to libmapper and MapperGUI.

Use of the other connection modes was rare. Users found the expression input box hard to use. Directly calculating the appropriate mathematical expression was seen as too abstract. This is a sensible problem to have, as difficult text-based input is precisely the thing that MapperGUI is designed to avoid. One user suggested a two-dimensional graphical tool, showing the transposition from input to output, to help with this task.

Some users requested for signal values themselves to be available in MapperGUI. This would create a lot of bandwidth clutter, as all devices would need to constantly send signal data to the GUI. It was suggested the user should be able to query signal data by clicking or placing the mouse cursor over signal names.

[1] Raspberry Pi — An ARM GNU/Linux box for $25. Take a byte! [Online] Available: `http://www.raspberrypi.org`. Accessed August 1, 2013

5.1.2 Saving & loading

Nearly all users made use of the saving and loading features in some way. For both experimental and design-based setups, returning to prior mappings is very useful, as it avoids the tedium of performing the same tasks repeatedly.

We received criticism for the naïve loading system. One user found it tedious that mappings would accumulate when loading multiple. He required rapid switching between the same few mappings for his experiment. Once these mappings were created there was little that needed to be done to modify them. For his experiment it became tedious to erase a previous mapping before loading a new one. Obviously in a live-performance context the amount of delay that would be created with this kind of task would be unacceptable.

Another user wished to switch between mappings in his work, but required some kind of intermediate space between the states. Ideally loading would have the option of blending between two mappings, such that the transition is not too harsh for the audience. To maintain this functionality, the actual saving and loading of patches was transferred to Max/MSP for this project.

In a situation with many devices of the same class, loading a single mapping can be somewhat absurd. Because each connection will be loaded $m * n$ times, where m is the number of similarly named input devices, and n is the number of relevant output devices, certain simple mappings can result in hundreds of unwanted connections when a mapping is loaded. Perhaps some kind of staging area wherein the user must explicitly for which devices he or she wishes to load a mapping could solve this problem.

Another user asked for some kind of mapping preset that could be created and loaded whenever the program is opened. This way if the same experiment or performance is conducted repeatedly the user would simply need to launch MapperGUI and get started.

5.1.3 Reliability & responsiveness

Multiple users commented on the frustrating nature of interacting with MapperGUI when it became out of sync with the libmapper network. As one user stated, "The program is not useful if you do not *trust* the display." In this way small errors, devices not appearing, signals not accepting connections, delays in operations, etc. become a very big problem for user satisfaction with MapperGUI. Users reviewed the refresh button very favorably. If something seemed amiss with the GUI or the network, and refreshing the display solved

the problem, then trust in the display was restored.

Some problems were due to errors in the libmapper code and were out of the control of MapperGUI. Others were created when MapperGUI code started to make assumptions about the libmapper network. For example, with the drag-to-connect gesture, originally the drawn arrow persisted upon release of the mouse button. MapperGUI assumed that a connection would be made and kept the line to avoid delays. Of course, occasionally the signals were *not* connected, due to dropped messages or incompatibility. In this circumstance the faulty arrow, representing nothing, became very confusing. Due to negative feedback the code was changed such that the drawn arrow disappears immediately after the drawing gesture. If the connection is successful, it is redrawn. This results in a slight flicker (as the arrow is erased and re-drawn), but this was much more popular than the potential erroneous arrows persisting in the display.

Some heavy operations, like scrolling and forming multiple connections, could create significant delays (on the order of a few seconds) in MapperGUI. Users responded very negatively to such delays, as they were accustomed to computer programs responding much more quickly. Generally multi-second delays were thought to be errors, thus reducing the user's trust in the application. In Section 5.2 we explore solutions to this problem.

5.1.4 Effectiveness of alternate views

GridView and HiveView have only recently been recently included into the program. As a result most of our users are much more familiar with ListView. Users reported that while the alternate views were interesting, ListView was the most straightforward for creating mappings. It was reported that GridView was useful once most of the mapping was completed, as one could notice patterns that were not apparent in ListView, and alter mappings to reflect the findings. The limited functionality of HiveView meant that to most users it was simply a visualization tool. Also it was extremely common among our test users for use-cases to include very few devices with many connections, so the "whole-network" view in HiveView was not advantageous.

5.2 Testing program responsiveness

Extension of interface features discussed in Section 4.3 leads to some control possibilities that could be difficult for the GUI to handle. Addition of shortcut keys for connecting,

linking, disconnecting and unlinking, as well as the ability to select multiple devices and signals at once allows users to create and delete hundreds of connections with a single key press. Naïve saving and loading produces to situations where dense mappings will accidentally be applied to several instruments at once.

Though the "update display" model works extremely well for code modularity, it generates awkward situations when dealing with massive network operations. Since the system updates the entire display with each change to the network, deleting 100 links (if the user is clearing a large network) results in 100 independent `delete_link` messages arriving at the monitor. For each one of these messages, the display will fully update. In the case of ListView, all arrows will be cleared, and redrawn with one fewer present (as if the links are being deleted one-by-one). In total 4950 arrow drawing operations[2] will occur, resulting in significant delay.

As reported by users in Section 5.1.3, any GUI operation that takes more than a few seconds, without some kind of visual feedback (like a "loading" bar), leads to frustration and mistrust of the program. Obviously, if the GUI is going to support these kinds of massive network manipulations, there needs to exist some way to keep them under control.

5.2.1 Rate limiting functions

In order to prevent thousands of unnecessary, display re-draws, a "waiting" period was added to certain critical functions. This operating systems process is described in Silberschatz et al. (2003). Essentially certain functions no longer execute in the code immediately once called, instead, a delay timer starts. If the function is called again during this delay, the delay timer simply restarts. The function is only executed once the delay timer finishes. This way, if a function is called 100 times simultaneously, it will only execute once after a short delay. Figure 5.1 shows the effect of the waiting period if the function is called a single time, and if it is called once during the delay.

Two functions are limited in the GUI: `view.update_display` for all views and `update_arrows` for ListView. `update_display` is common to all views, and the massive network changes described above can result in many processor intesive functions to run needlessly. With the `update_arrows` function, operations that do not result in changes to the network

[2] $99 + 98 + 97 + ... + 2 + 1 = \frac{99*100}{2}$. Note that $\frac{n*(n+1)}{2}$ arrows will be drawn for any n number of connections or links.

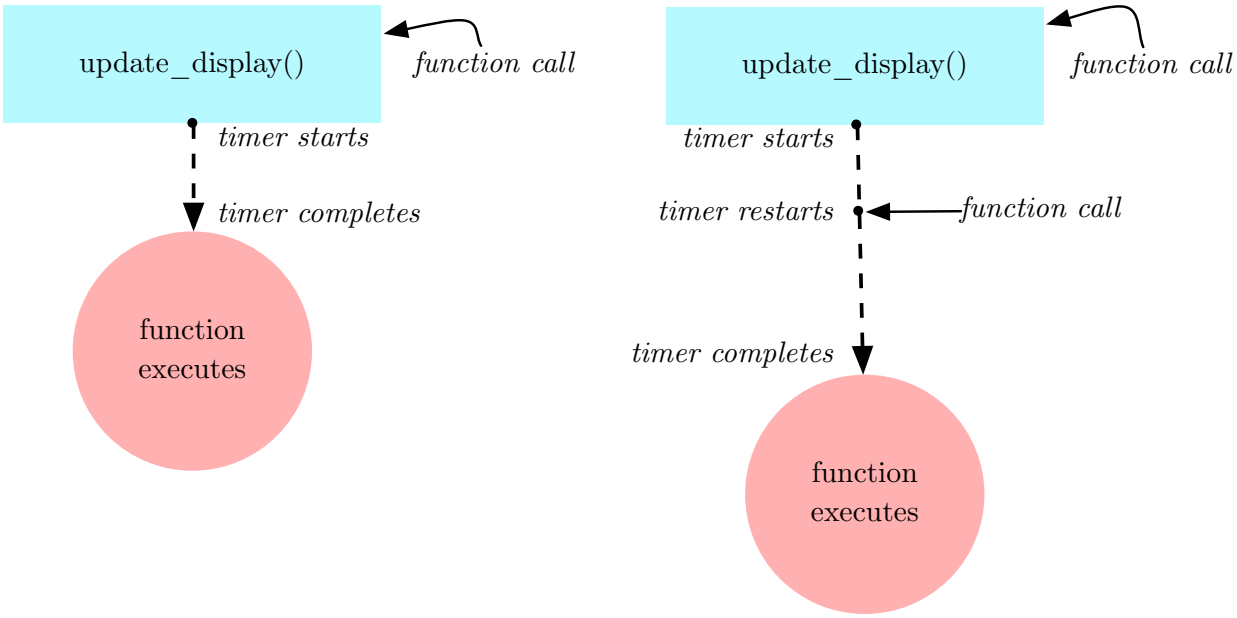


Fig. 5.1 Illustration of a delayed function.

(scrolling, changing tabs) require constant re-drawing.

Exactly how much time this delay should be set to is not obvious. If the delay is too short it is possible for massive network operations to still call the function many times if they arrive asynchronously. A too-long delay means that users may notice for simple actions, like scrolling with a single arrow (causing the scrolling to appear jerky). Another consequence of a long delay is that a process which calls the delayed function at a regular interval could continuously restart the delay. In this case the function will *never* execute, a situation known as "starvation" (Silberschatz et al. 2003).

After some informal tests of delays between 17 and 1000 milliseconds, a delay of 33 milliseconds was selected for both functions. Substantial improvement in execution speed was observed for even very short delays, as often hundreds of function calls would reach the `view.update_display` virtually simultaneously. With delays closer to one second there was little improvement in response to massive network operations, and the delay itself became noticeable (especially with scrolling operations). 33 milliseconds is in the range where nearly every operation results in just a single function being executed, but is also an imperceptible delay to a human user. The number 33 itself was selected because it is the length of two screen refreshes on a 60 Hz display (a measure recommended in Silberschatz

et al. 2003).

5.3 Comparison to Similar Interfaces

Other systems exist to help non-programmers map control inputs to sound synthesis parameters. This section compares this research to these related works.

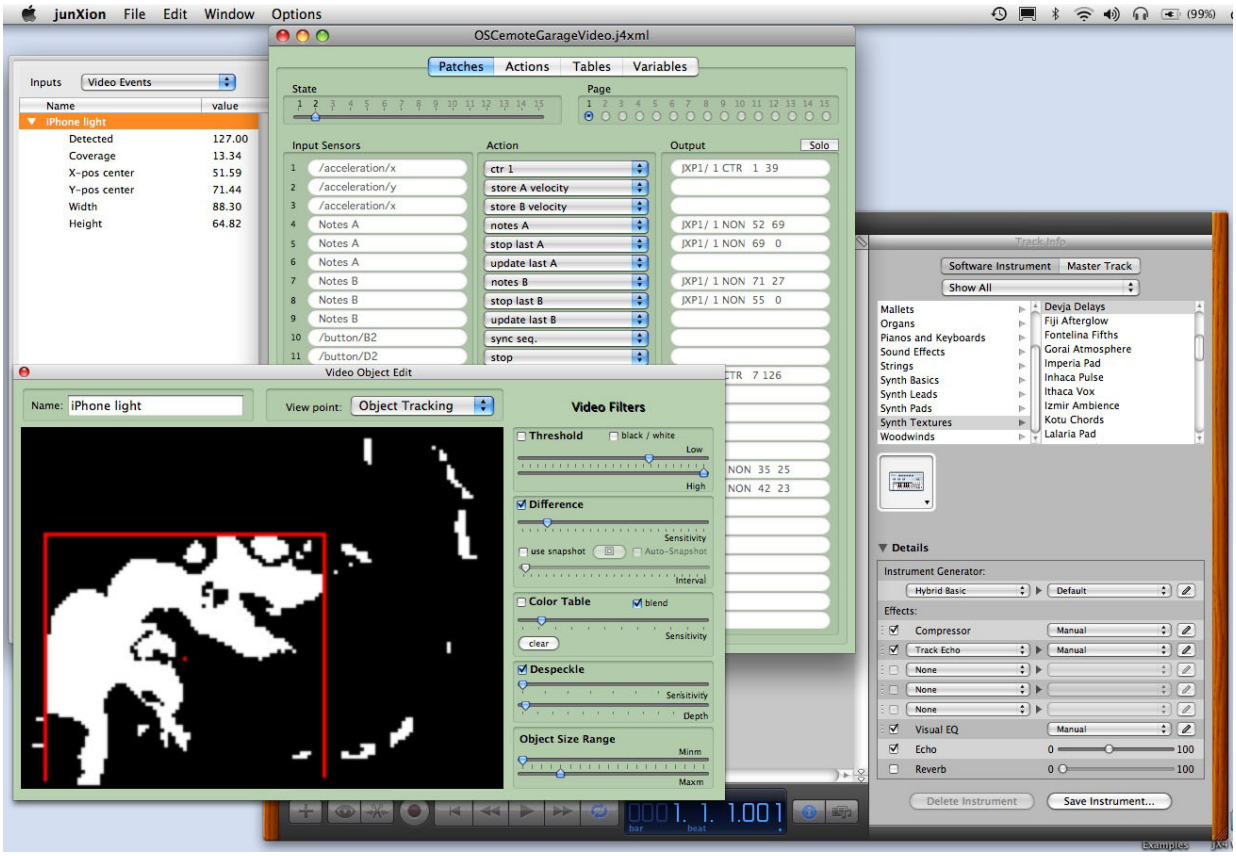


Fig. 5.2 STEIM's JunXion software

The Studio for Electro-Instrumental Music (STEIM) distributes JunXion (STEIM 2004), a software application for controlling MIDI and OSC-based systems. JunXion automatically detects input devices like computer mice and USB video-game controllers. The user is able to drag any child signals from these controllers onto one of 25 possible inputs. From there the user can switch to the "actions" tab, where destinations and connection properties can be customized. Connection properties are stored in groups that populate drop-down

menus in the central column. JunXion features a very interesting "state" system similar to MapperGUI's saving and loading. Once a successful mapping is created, users can change the state, starting a new mapping. With multiple mappings, users can quickly switch between states. JunXion also has a very interesting graphical signal conditioning editor. The program presents a two dimensional field and the user can draw, generate curves and set bounds. Incorporating such a feature into MapperGUI would assist users who were unimpressed by textual expression input.

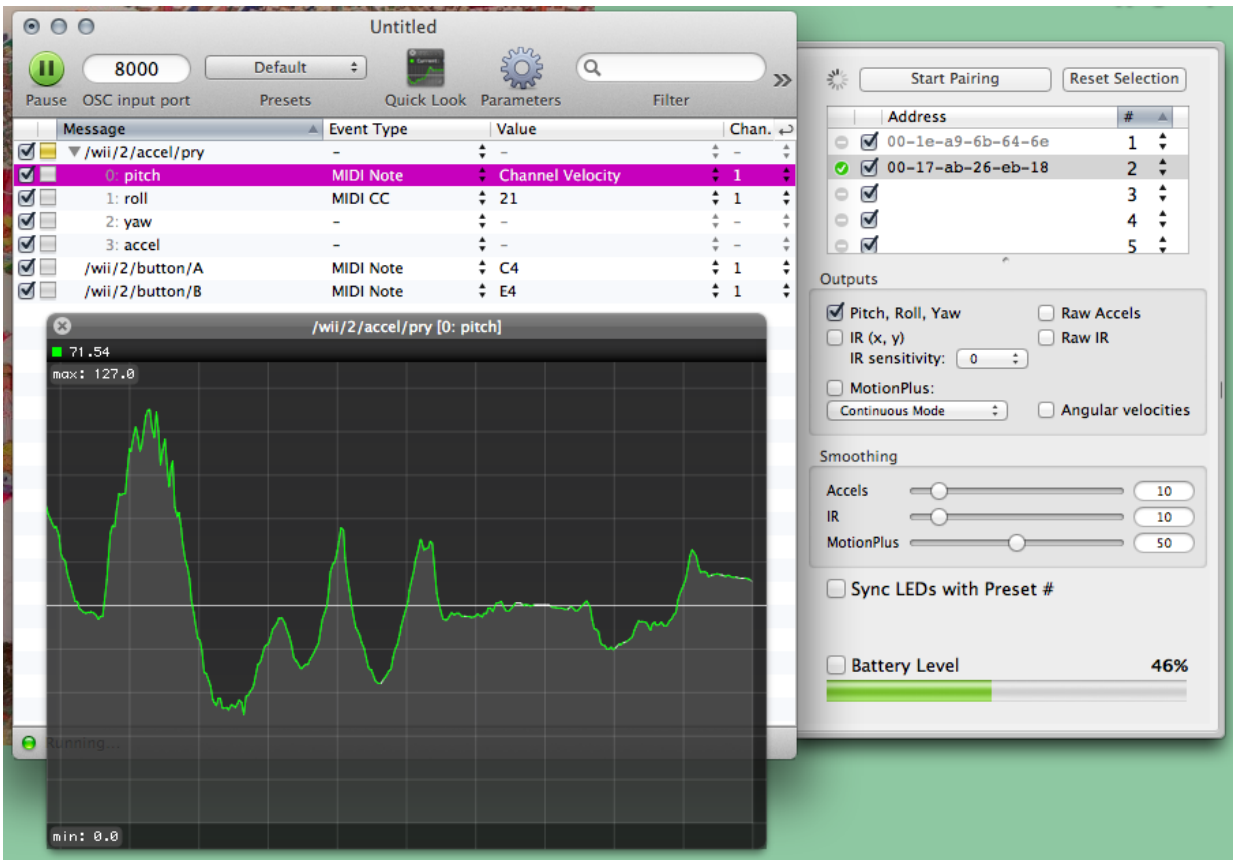


Fig. 5.3 The OSCulator interface

The OSCulator system (Wildora 2012) is very similar to JunXion. Compatible controllers appear automatically and can be mapped to MIDI or OSC signals. OSCulator also relies on a drop-down menu based interface for selecting where and how the output will be routed. As in JunXion the idea of a "connection" is not emphasized, instead a MIDI or OSC message is simply sent on a specific channel (the receiving end must be specifically

notified on which channel to receive messages). As can be seen in Figure 5.3, OSCulator displays a real-time oscilloscope like visualization for selected signals. A similar feature would help MapperGUI with visual feedback, which was a desire of some users.

The Eaganmatrix (HakenAudio 2013) partly inspired GridView in MapperGUI. The signals of a single control and synthesis device are displayed on the x and y axes of a grid display. Connections between the two are made by clicking on the intersections. The Patchage interface (Robillard 2011) contains uses an interaction very similar to ListView, where objects containing lists of signals can be connected by dragging gestures. Max/MSP and an Integra Live (Bullock, Beattie, and Turner 2011) also feature this interaction, but neither are necessarily for creating mappings.

5.4 Evaluation of Goals

A set of goals for the software was established at the beginning of this document. These were to create an interface for libmapper that was easy to use for non-programmers. Of primary concern was to create an interface that was flexible and intuitive. We also intended to unite features of the three prior GUIs, both to capitalize on work already completed and to create a single, standard GUI for libmapper.

The GUI currently exists in a distributable form, allowing Macintosh users to download and easily use the software. Unfortunately standalone applications for non-Macintosh platforms are not yet available. Most features from prior interfaces were integrated into a cross-compatible web-based form. A modular code-base was created for the application, greatly improving the processes of maintaining and extending this GUI verses prior interfaces. Two new view modes were integrated into the display, though it is too early to conclude as to whether they significantly contribute to the flexibility of the system.

The software was provided to users, and thus tested in a variety of contexts. MapperGUI was able to handle most use-cases in its present state, according to user feedback. All shortcomings were recorded, those that have not yet been dealt with are listed, along with possible solutions, in Section 6.2 of this document. Though it has not yet been the case, in the future MapperGUI will likely be used to handle mappings in a live performance context. This will give us a new perspective on how the software performs in a situation where instant reactivity is necessary and errors can be disastrous.

Chapter 6

Conclusions & Future Work

This chapter summarizes the work presented this thesis, presents conclusions, and summarizes possible avenues for further research.

6.1 Summary and Conclusions

This thesis began by exploring issues relevant to musical mapping interfaces. DMI designers typically hard-code mappings into their designs, making collaboration, cross-compatibility and modification difficult. Certain tools exist to aid these designers and their users, but often they are difficult to use for non-experts in computer programming. Our work was motivated by this situation in musical mapping software. MapperGUI aimed to lower the barriers to entry for those who wished to use libmapper, a software library for collaborative and configurable musical mapping. The GUI was designed to allow for quick and straightforward manipulation of musical networks.

Techniques from data visualization and user interface design were presented to illustrate general principles used in MapperGUI's design. The ideas and structure of libmapper were summarized to describe the requirements of a GUI for the library. Prior GUIs for libmapper were described, as ideas and code were borrowed from them in the creation of MapperGUI.

The final GUI took the form of a modular interface. Various independent view modes could be swapped in and out by the user, making MapperGUI useful for a wider variety of libmapper networks. The code itself was also structured in a modular fashion such that extensions could be created more easily. The program was made accessible to libmapper users throughout this project, and their feedback became a crucial factor in design decisions.

MapperGUI has met many of the goals set at the beginning of this thesis work. The interface is available, functional and very accessible. The majority of libmapper variables can be accessed and manipulated from within the GUI. Within ListView connection and linking are easy and intuitive. Two alternate views are present for networks and tasks where list-type views are cumbersome. Users can save and load mappings.

The current release of MapperGUI, however, is still in a test phase. A number of issues need to be resolved before the software can be adopted as standard GUI for libmapper.

6.2 Future Work

6.2.1 Unimplemented features

A few features present in Maxmapper have not yet been implemented in MapperGUI. Most importantly, MapperGUI currently does not support sending a signal as an instance. Instances are one of the true strengths of libmapper. To design a way, even an inelegant way, to allow the user to take advantage of this libmapper feature is a high priority for the next release. Users are also unable to edit link scopes in the current version. Support for this will be fairly easy to implement, via a drop-down menu on the top bar and some extensions to the python monitor.

Our search functions, though usable, are not yet quite as powerful as those found in Maxmapper. Maxmapper allows users to filter signals for common prefixes, by way of a drop-down menu. MapperGUI also forces users to remain on whichever network on which the program was launched. In the case where multiple networks are available, it would be a good extension to allow users to select and switch between them. Finally, MapperGUI's expression editing was poorly reviewed by users. Upon a double click of the *Expression* button, Maxmapper displays a palate with all possible expression syntax (to create exponential functions, averages, etc.). Incorporating this feature would be a good start.

6.2.2 Possible extensions

With the MVC architecture and some alternate views in place, our group has planned extensions to MapperGUI that may be very interesting. Firstly, HiveView should be made fully interactive, allowing the user to create links and connections with a dragging gesture.

The hierarchical edge bundling technique described in Section 2.2.2 would be very useful for this view, as currently connection lines are drawn somewhat arbitrarily. Attempts were made to integrate Vizmapper into MapperGUI as a single view, as it was an initial goal of the process. Unfortunately idiosyncrasies in the Vizmapper code-base made this more difficult than anticipated. In the future the we hope to restructure the Vizmapper code so that it might be included, as it is a very interesting and effective network visualization.

Another way that the research in Section 2.2.2 could be applied is through a new "area" view mode. In this mode network elements would be displayed as shapes on a Cartesian plane, with each axis user-mappable to any kind of quantitative metadata. Other metadata could be visually mapped to these shapes' color, orientation, shape, opacity, etc. This would be a very interesting way to analyze the findings of Mackinlay (1986), and may be the topic of a future project.

The saving and loading features of MapperGUI obviously need improvement. Work has already begun on a staging process for loaded mappings, such that saved mappings are only loaded for the desired devices. Finally, standalone support for Windows and Linux systems is definitely required for future releases of MapperGUI. It would also be interesting to begin work on mobile versions of MapperGUI, though that would be a much more time-consuming process.

References

Baalman, M., V. de Belleval, C. L. Salter, J. Malloch, J. Thibodeau, and M. M. Wanderley. 2010. Sense/stage - low cost, open source wireless sensor infrastructure for live performance and interactive, real-time environments. In *Proc. of Linux Audio Conference*, 242–249.

Bau, O., A. Tanaka, and W. E. Mackay. 2008. The a20: Musical metaphors for interface design. In *Proc. of International Conference on New Interfaces for Musical Expression*, 91–96.

Bertin, J. 1983. *Semiology of Graphics*. The University of Wisconsin Press.

Booth, G. 2010. Inclusive interconnections: Towards open-ended parameter-sharing for laptop ensemble. Master's thesis, University of Huddersfield, Huddersfield, England.

Bullock, J. 2008, March. Braun. Last accessed June, 19 2013, `http://sourceforge.net/projects/braun/`.

Bullock, J., D. Beattie, and J. Turner. 2011. Integra live: a new graphical user interface for live electronic music. In *Proc. of International Conference on New Interfaces for Musical Expression*, 387 – 392.

Chadabe, J. 2000, February. The electronic century part i: Beginnings, electronic musician. *Electronic Musician*: 74–90.

Cleveland, W. S., and R. McGill. 1984, September. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association,* 79 (387): 531–554.

Cook, P. R. 2009. Re-designing principles for computer music controllers: a case study of squeezevox maggie. In *Proc. of the International Conference on New Interfaces for Musical Expression*, 262–263.

Corry, M. D., T. W. Frick, and L. Hansen. 1997. User-centered design and usability testing of a web site: An illustrative case study. *Educational Technology Research and Development* 45 (4): 65–76.

Goudeseune, C. 2002. Interpolated mappings for musical instruments. *Organised Sound* 7 (2): 85–96.

HakenAudio. 2013, April. Eagan matrix. Last accessed June 19, 2013, `http://www.hakenaudio.com/Continuum/eaganmatrixoverv.html`.

Halmos, P. R. 1970. *Native Set Theory*. Springer-Verlag.

Höllerer, T., J. Kuchera-Morin, and X. Amatriain. 2007. The allosphere: A large-scale immersive surround-view instrument. In *Proc. of Workshop on Emerging Displays Technologies*, 21 – 28. ACM Press.

Holten, D. 2006, September/October. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphiscs* 12 (5): 741–748.

Hunt, A., and R. Kirk. 2000. Mapping strategies for musical performance. *Trends in Gestural Control of Music*.

Hunt, A., and M. M. Wanderley. 2002. Mapping parameters to synthesis engines. *Organised Sound* 7 (2): 97–108.

Hunt, A., M. M. Wanderley, and R. Kirk. 2000. Towards a model for instrumental mapping in expert musical interaction. In *Proc. of International Computer Music Conference*, 2–5.

Hunt, A., M. M. Wanderley, and M. Paradis. 2003, December. The importance of parameter mapping in electronic instrument design. *Journal of New Music Research* 32: 429–440.

Imhof, E. 1982. *Cartographic Relief Presentation*. ESRI Press.

Jensenius, A. R., T. Kvifte, and R. I. Godøy. 2006. Towards a gesture description interchange format. In *Proc. of the International Conference on New Interfaces for Musical Expression*, 176–179.

Kling, R. 1977, December. The organizational context of user-centered software designs. *MIS Quarterly* 1 (4): 41–52.

Krasner, G., and S. Pope. 1988. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming* 1 (3): 26–49.

Kvifte, T. 2008. On the description of mapping structure. *Journal of New Music Research* 37 (4): 353–362.

Mackinlay, J. 1986, April. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics* 5: 110–141.

Malloch, J., S. Sinclair, and M. M. Wanderley. 2008. A network-based framework for collaborative development and performance of digital musical instruments. *R. Kronland-Martinet, S. Ystad, and K Jensen. (Eds.): CMMR 2007, - Proc. of Computer Music Modeling and Retrieval 2007, Conference, LNCS 4969. Berlin Heidelberg: Springer-Verlag*: 401–425.

Malloch, J., S. Sinclair, and M. M. Wanderley. 2013. Distributed tools for interactive design of heterogenous signal networks. In *Multimedia Tools and Applications*.

Mehlhorn, K., and P. Sanders. 2008. *Algorithms and Data Structures: The Basic Toolbox*. Springer.

Momeni, A., and C. Henry. 2006. Dynamic independent mapping layers for concurrent control of audio and video synthesis. *Computer Music Journal* 30 (1): 49–66.

Nort, D. V. 2010, January. *Modular and Adaptive Control of Sound Processing*. Ph. D. thesis, McGill University, Montreal, Canada.

Place, T., and T. Lossius. 2006. Jamoma: A modular standard for structuring patches in max. In *Proc. of International Computer Music Conference (ICMC 2006)*.

Robillard, D. 2011, January. Patchage. Last accessed June 19, 2013, `http://drobilla.net/software/patchage/`.

Rudraraju, V. 2011, December. A tool for configuring mappings for musical systems using wireless sensor networks. Master's thesis, McGill University, Montreal, Canada.

Schulze, A. N. 2001. User-centered design for information professionals. *Journal of Education for Library and Information Science,* 42 (2): 116–122.

Silberschatz, A., P. B. Galvin, and G. Gagne. 2003. *Operating Systems Concepts* (6 ed.). John Wiley and Sons, Inc.

STEIM. 2004, Summer. Junxion - products of interest. *Computer Music Journal* 28 (2): 105–107.

Tuckey, J. W. 1965, April. The technical tools of statistics. *The American Statistician* 19 (2): 23–28.

Tufte, E. R. 1990. *Envisioning Information*. Graphics Press.

Tufte, E. R. 2006. *Beautiful Evidence*. Graphics Press.

Wanderley, M. M., and P. Depalle. 2004. Gestural control of sound synthesis. In *Proc. of the Institute of Electrical and Electronics Engineers*, Volume 92, 632 – 644.

Wildora. 2012, May. Osculator. Last accessed June 19, 2013, `http://www.osculator.net/`.

Wolek, N. 2010. The mpg carepackage: coordinating collective improvisation in max/msp. In *Proc. of the Society for Electro-Acoustic Music in the United States*.

Wright, M., and A. Freed. 1997. Open soundcontrol: A new protocol for communicating with sound synthesizers. In *Proc. of International Computer Music Conference*, 101–104.