Visualising 1,051 Visual Programs
Module Choice and Layout in the Nord Modular Patch Language

James Noble **Robert Biddle**

Computer Science
Victoria University of Wellington
New Zealand
{kjx,robert}@mcs.vuw.ac.nz

Abstract

The Nord Modular music synthesiser system comprises a standalone array of digital signal processors programmed by a dataflow visual langauge and supported by a visual programming environment that runs on commodity hardware. A crucial difference between the Nord Modular and traditional modular synthesizers is that each Nord module can be positioned individually, whereas physical analogue signal processing units are typically installed in fixed racks. We have used information visualisation techniques to investigate the layouts and programming style of 1,051 Nord Modular programs. We found that although modules could be positioned freely within a program, particular types of modules were generally found in sterotypical locations.

Keywords: Software Visualisation, Corpus Analysis.

1 Introduction

Domain-specific dataflow visual programming languages are now commonplace throughout the world of computing. Even in just one domain, computer music, several dataflow visual languages have been successful as products over relatively long terms, including Max [Cycling '74, 2001], Bars-n-Pipes [Hagen, 1990], and the visual programming language underlying the Nord Modular synthesizer [Clavia DMI AB, 1999].

The Nord Modular synthesizer system, by Clavia AB of Sweden, is a digital (re)creation of the traditional analogue modular synthesizer systems common in the 1970s. A traditional modular synthesizer consists of a number of modules such as voltage controlled oscillators (VCO), voltage controlled filters (VCF), envelope generators (EG) and low-frequency oscillators (LFO) fixed in position to a rack and connected together using "patch cords" to produce sounds.

Figure 1 shows a very basic Nord Modular patch. This comprises four modules: an input module named Keyboard1, an oscillator (OscB1), an envelope generator (ADSR-Env1) and an output module (2 outputs1). Figure 2 shows a more complex patch.

The Nord Modular system includes approximately a hundred different types of modules, including oscillators, filters, clocks, and so on: a patch generally uses around 20 modules. Module positions are constrained to one of four columns and about thirty rows without scrolling: all modules are one column wide, but different modules have different heights.

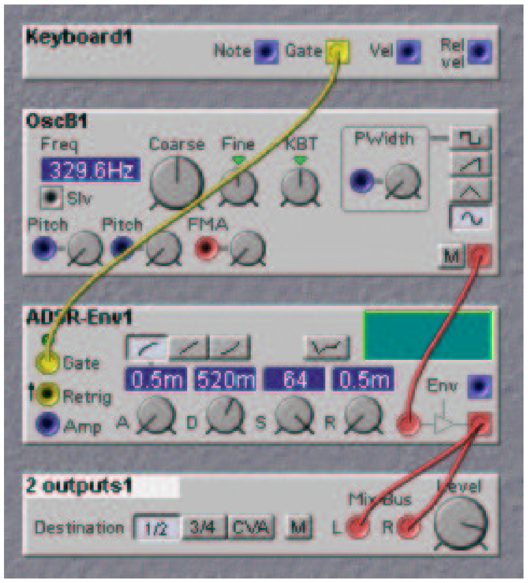

Figure 1: A Simple Modular Patch

1.1 Nord Modular Programming

Patches are produced using a programming environment called the Nord Modular Editor[1]. Using the Modular Editor is different from using a physical analogue modular system in a number of important ways. The most critical is that programming analogue systems is embodied in the physical world, connecting physical patch cables to physical modules, whereas in the Nord this experience is recreated in software for general-purpose computers. Because the Nord is a software simulation of a modular synthesizer, some constraints are different. In a physical modular synthesizer, users are restricted to a certain number of modules of a given type; if you run of out oscillators, you must do without or buy a new module. In the Nord Modular, patches are limited by the amount of DSP processing power: you can choose to trade off oscillators against filters or envelope generators, but only up to a fixed limit. Crucially, users can choose the layout of the modules in a Nord modular patch, whereas physical module positions are fixed in a traditional modular synthesizer.

As well as providing the interface for users to create and edit patches, the Modular Editor also allows patches to be stored into standard file systems. Unlike many other synthesizers, where patch information is only available encoded in MIDI System Exclusive formats, the Nord Modular patches are stored in standard ASCII files (one of the advantages of a system that relies on commodity computer support). This file format is quite simple, and has been designed to be easy to exchange between Modular users.

[1]The Nord Modular Editor is available for download free from http://www.clavia.se/nordmodular.

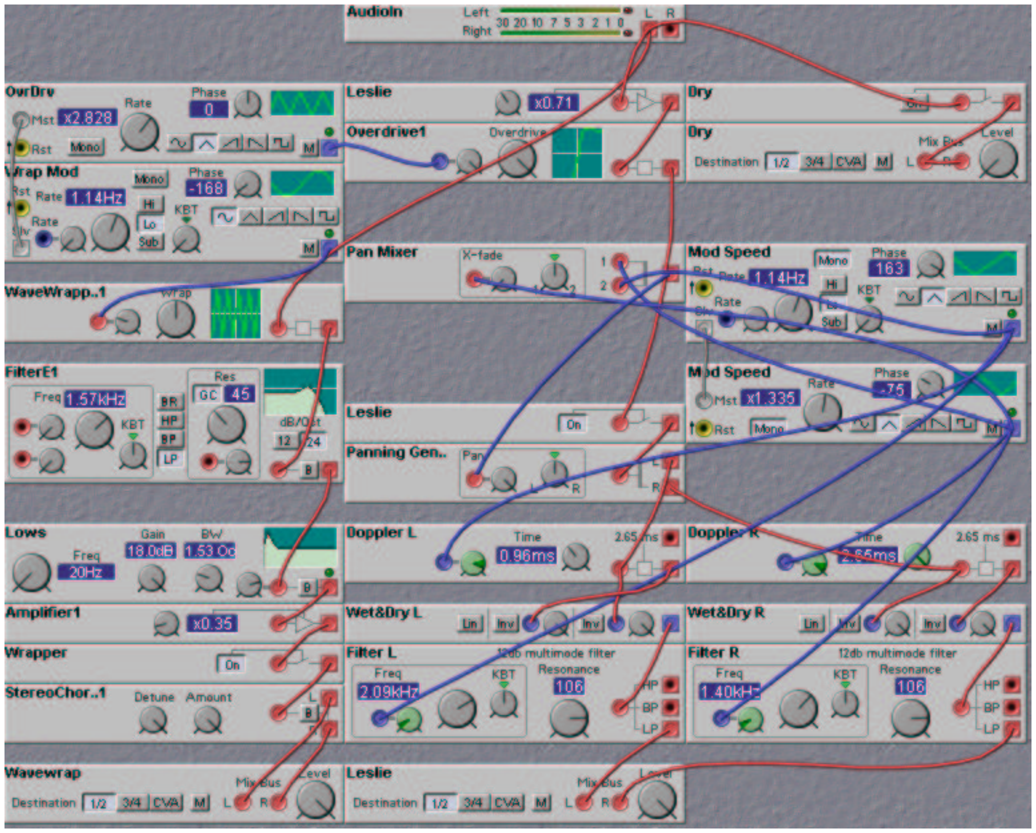

Figure 2: A More Complex Patch

For example, on a web page, a textual link (or snapshot image of a patch in the Editor) can be linked to the patch file. Clicking on the patch will then automatically open the patch file in the editor and load it into a Modular synthesizer, assuming one is attached.

1.2 The Nord Corpus

The ease with which patches can be archived and shared has meant that many Modular users have made patches available; the manufacturer, Clavia, also collects user-contributed programs on their web site. This ensures there is a readymade corpus of publically available Modular patches for visualisation and analysis, probably more so than with any other visual programming language. In this paper, we apply program visualisation techniques to the Nord Modular programming language to analyse the choice and layout of modules within patches: previous work has investigated visualising whole patches and the vectors of the patch cords within then [Noble and Biddle, 2002]. The corpus used in this paper contains 467 patches supplied with the Nord Modular ("Factory Patches") plus 584 user contributed patches [2].

2 Module Choice

We begin this investigation by considering the number of modules of each type that are chosen by programmers for use in patches. Figure 3 shows the number of times modules of each category are used across the corpus. From this visualisation, we can readily see that every patch includes one input and one output module (in fact, a few patches must include more than one output module: the extra gridline shows one module per patch). Many patches must include several oscillators, envelope generators and

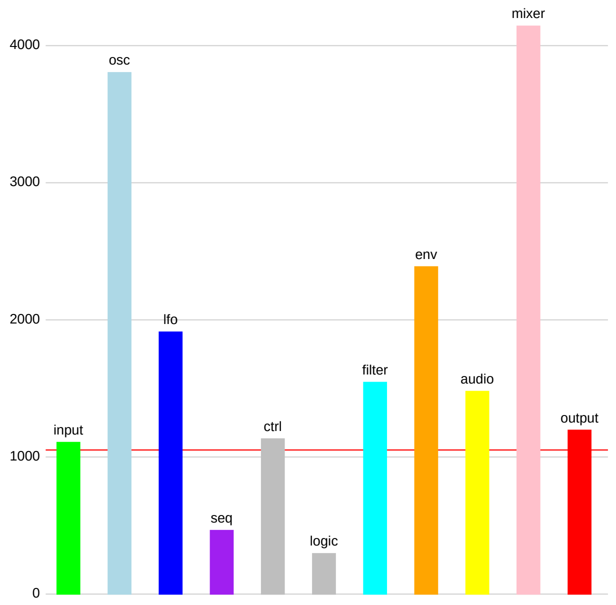

Figure 3: Module Categories

mixers, with relatively fewer audio and control (ctrl) modules. Sequencer (seq) and logic modules are comparitively underutilised.

We can also show the number of times any particular module is used. Each point in Figure 4 represents an individual module (colour-coded to the module categories in Figure 3). In this figure it can be seen that module utilisation is rougly exponential. The most commonly used modules (top down in the upper right of the diagram) are the ADSR envelope generator, the mixer, the standard output module, the control mixer, and the keyboard input module.

[2]The "V3.0 Factory Patches" and "V3.0 Contributed Patches" are available for download from http://www.clavia.se/nordmodular

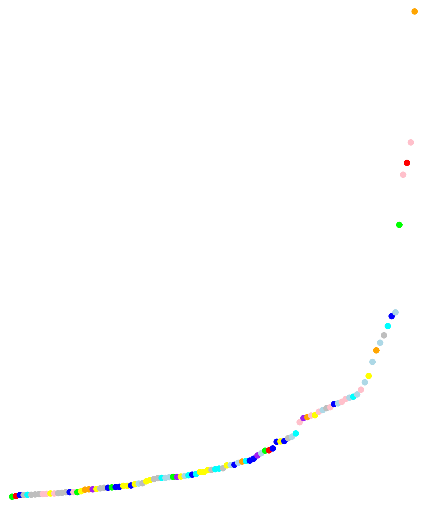
Figure 4: Module Utilisation

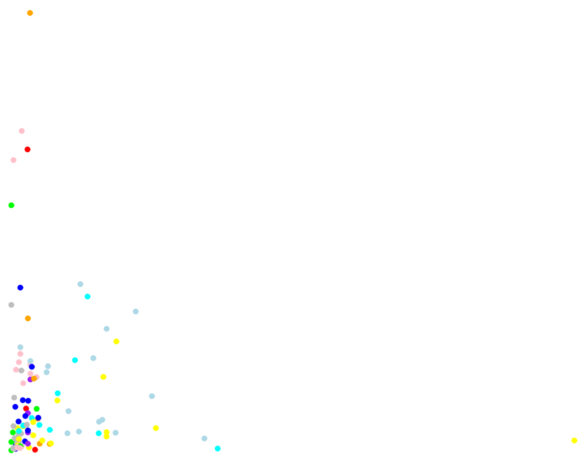
Figure 5: Module Utilisation vs. Module Power

2.1 Module Power

Different modules require different amounts of DSP processing power. Since the Nord Modular hardware has a fixed amount of power available (one, four, or eight Motorola 56000 DSPs depending upon the model) Figure 5 visualises a module's power requirements (on the x axis) against its utilisation (on the y axis as in Figure 4). This figure shows that most modules have low power requirements, including those modules that are used most frequently (close to the y axis). Modules with very high power requirements (such as the vocoder, filter bank, sine bank, phaser and drum synthesizer, reading right to left along the x axis) are used infrequently. Note all these high-power modules actually combine the features of several less powerful modules.

2.2 Module Size

As shown in Figure 1, different modules are different sizes, in particular, while all have the same width different modules have different heights. We hypothesized that the larger the module the more DSP power it would require. Figure 6 plots module height (on the x axis) versus power (again on the y axis), and illustates that this hypothesis holds on a large scale: bigger modules generally consume more power than smaller modules. The largest modules are the drum synth and note sequencer (size 9) and the sine bank (size 10); the most power-hungry module, the vocoder, is size 8. Note also that the smallest module occupies two rows (size 2) and the next largest module is is one and a half times as large (size 3). There are no size 1 modules.

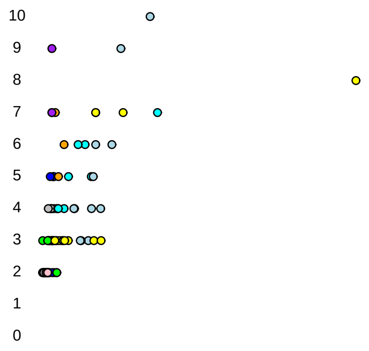

Figure 6: Module Height vs. Module Power

3 Module Layout

As well as visualising programmer's choice of modules, we also analysed modules' layout — their positions in programs. Figure 7 shows the "real estate utilisation" for the first five columns and fifty rows for all patches in the corpus. This image is an approximation to printing all 1,051 corpus patches onto acetates and then placing all the acetates into a large stack. The top left of the image is dark, thus almost every patch includes a module in that position: the bottom and right of the image is light, because few patches have modules there.

This diagram illustrates several large-scale features of Modular patches. First, most modules are positioned to the top and to the left of the patch layouts, and most substantially to the top. Second, patches are longer than wide — the figure is drawn with the same aspect ratio used by the modular editor (16:1). This implies that modular programmers prefer to scroll vertically, rather than horizontally, as the editor displays four columns and about forty rows on a 1024x768 pixel screen, three columns and thiry rows on a 800x600 screen, and two columns and twenty rows on a 640x480 screen. (Subsequent real estate diagrams will be cropped to the first three columns and thirty-five rows).

Third, the image shows prominent banding after the fourth row from the top: ever odd-numbered row is darker (more often occupied) than the even numbered rows. We hypothesize that programmers treat modules as if their size was always a natural multiple of the size of the smallest modules, that is, size 2 (see Figure 6), and lay them out on an imagined grid where rows are double height, with an aspect ratio of 16:2 rather than 16:1. Odd sized modules will be undersized in this grid, leaving a gap of one (Nord) row between the end of one module and the start of the next. Testing this hypothesis would require some kind of interaction with Modular progrmamers, such as a usability evaulation or questionnaire: while a corpus study is excellent for finding such effects, by its nature it cannot verify explanations for them.

3.1 Module Position

We have also considered the positions occupied by particular categories of modules. Figure 8 shows similar diagrams (draw with an 8:1 aspect ratio), each displaying a particular module type.

From this figure, we can see that input modules are tightly clustered along the top row, and the next two locations of the leftmost column, but rarely appear elsewhere (input modules are mostly size two). Oscillators are centred at the top left, however are more often positioned in the left column (presumably just under the input modules).

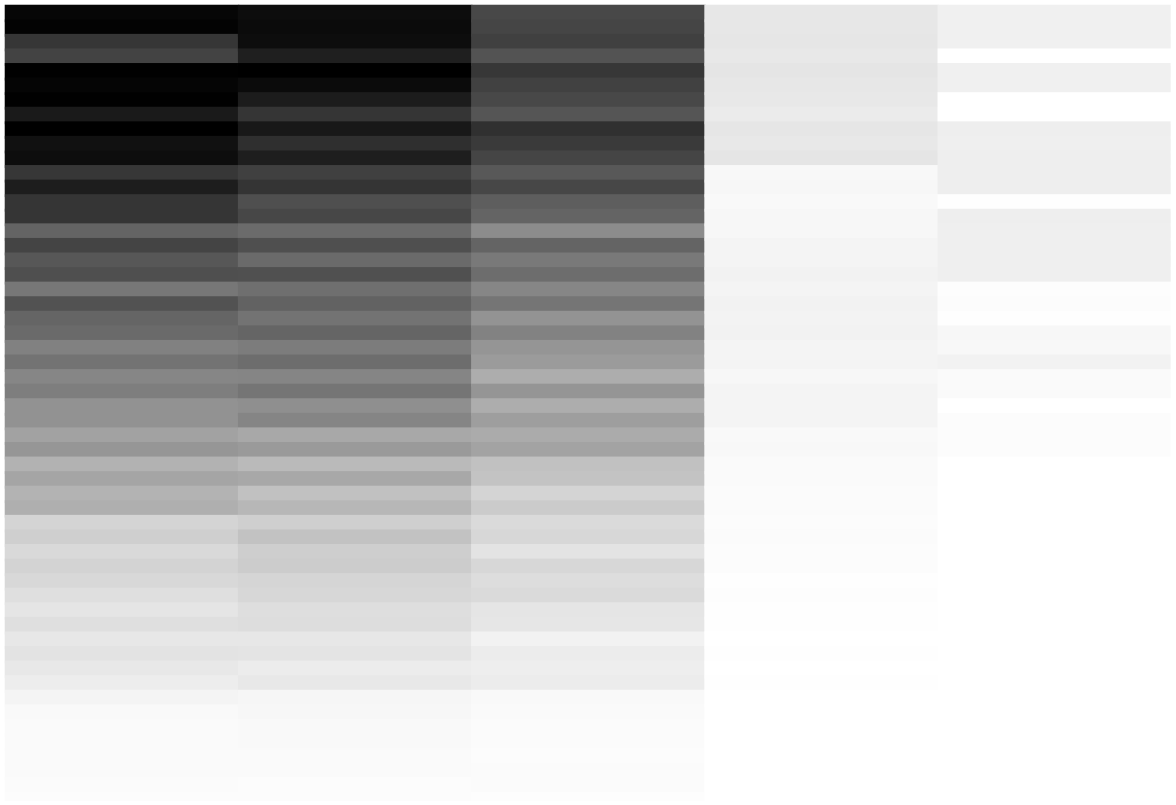

Figure 7: Real Estate Utilisation

Low frequency oscillators also appear primarily in the left-most column, either under the input modules or under the oscillators, further down. Note that the two top left positions are only occupied by input modules, oscillators, or LFOs.

The Nord documentation treats sequencer modules (seq) as special kinds of LFOs, however we have analysed them separately. Certainly they are positioned differently, mostly in the right column, rather than on the left. Filters, meanwhile, are mostly in the center column at at the top. Envelope generators are mostly to the top right of patches — compared with other modules, envelope generators and audio modules appear widely distributed throughout patches. Mixer modules are also quite widely distributed (although not as much as audio modules) and tend towards the middle of the left column. Output modules appear in the right column towards the bottom of the patch.

3.2 Location Use

Figure 9 is a complementary visualisation to Figure 8, showing the aggregate use made of each location in a patch. Colour coding and order is again taken from Figure 3. We see that the two top left positions are quite distinctly occupied by input modules, with the rest of the top of the left and middle rows occupied primarily by oscillators and LFOs. Filters are predominant in the middle of the second column, envelope generators at the top of the third, and output modules are lower down.

4 Related Work

Dataflow visual languages are arguably the most common form of visual language, and there are a number of commercial systems based upon such languages, including the IRIS Explorer [NAG, 2000], LabView [Baroth and Hartsough, 1995, NI, 2000], VEE [Helsel, 1997], CAPRE, [Hansen, 1997], and MAX [Cycling ’74, 2001, Desain et al., 1993], as well as the Nord Modular patch language [Clavia DMI AB, 1999]. Given this widespread practical acceptance, it is perhaps surprising that there has been little standalone research on visualising programs in these languages or analysing the kinds of programs these languages are acutally used to write.

Most research on visualisation of visual programs is generally subsumed with research on the visual programming environments themselves — indeed, one of the reasons for the software visualisation research community moving away from dataflow languages is that the execution of these programs is not easy to visualise. Rather, following Sketchpad [Sutherland, 1963] once again, many modern (non-dataflow) visual languages incorporate dynamic visualisations directly into the programming model, so that whenever a program runs it is visualised: Toontalk [Kahn, 1996], Agentsheets [Repenning and Sumner, 1995], and VIPR [Citrin et al., 1998] are just three examples of this approach.

There has been some work on specialised visualisation of visual programs, however. Burnett has applied software visualisation techniqes to support testing of Forms/3 programs [Rothermel et al., 2000], and Grundy and Hosking have applied some program visualisation techniques to software engineering modelling languages [Grundy and Hosking, 2000, Grundy et al., 1996] — in one case, successfully visualising a “gedanken” notation that was never designed to be executed [Grundy and Hosking, 1995].

Probably because most visual languages do not have a large user base, the practice of the visual languages community has been to adopt empirical usability evaluations to understand how languages are used, or to measure the effectiveness of individual small details of language designs [Rothermel et al., 2000, Blackwell, 2001], or researchers may participate in programming communities to evaluate their use of langauges [Carroll and Rosson, 1987, Nardi, 1993].

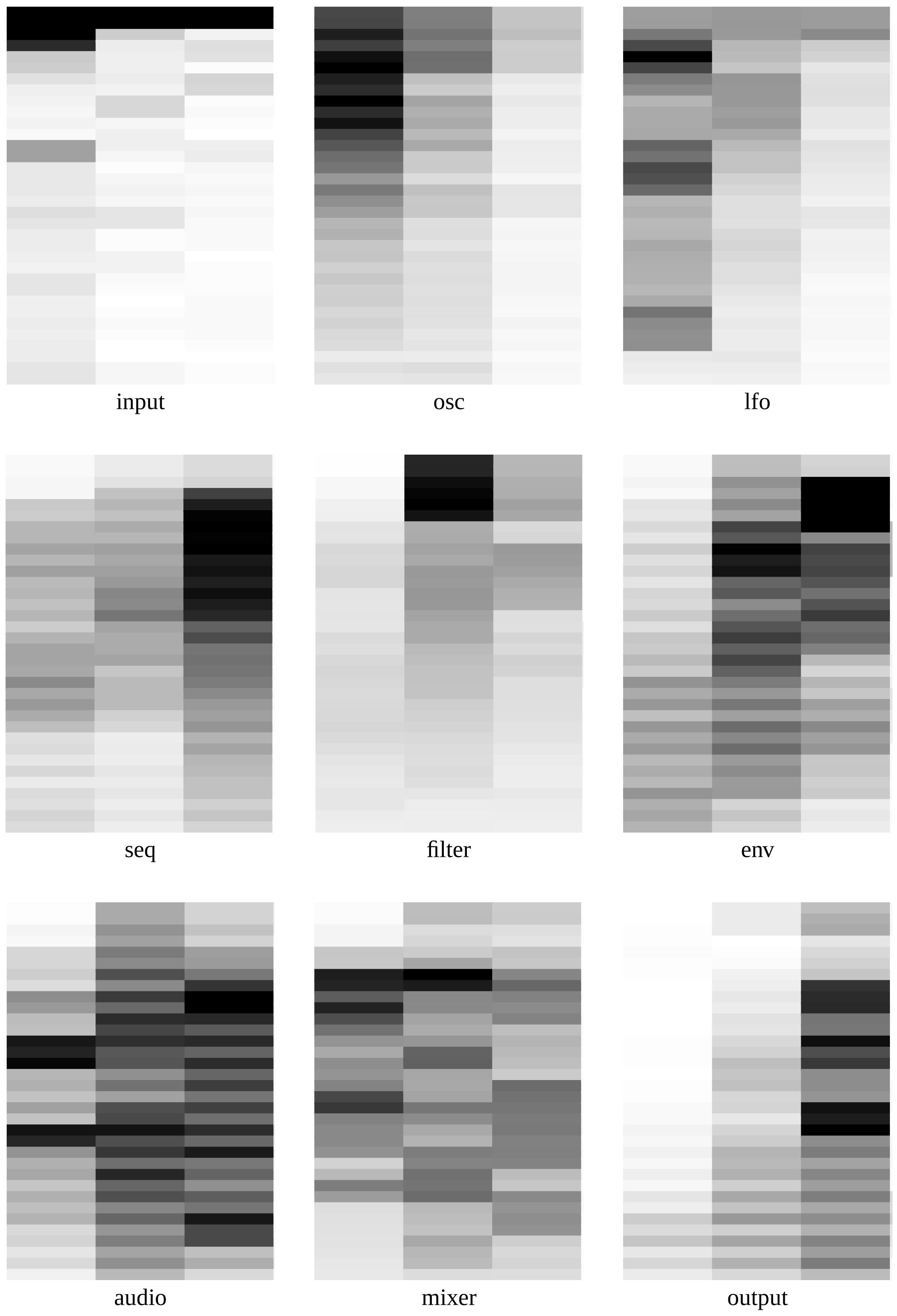

Figure 8: Module Position by Category

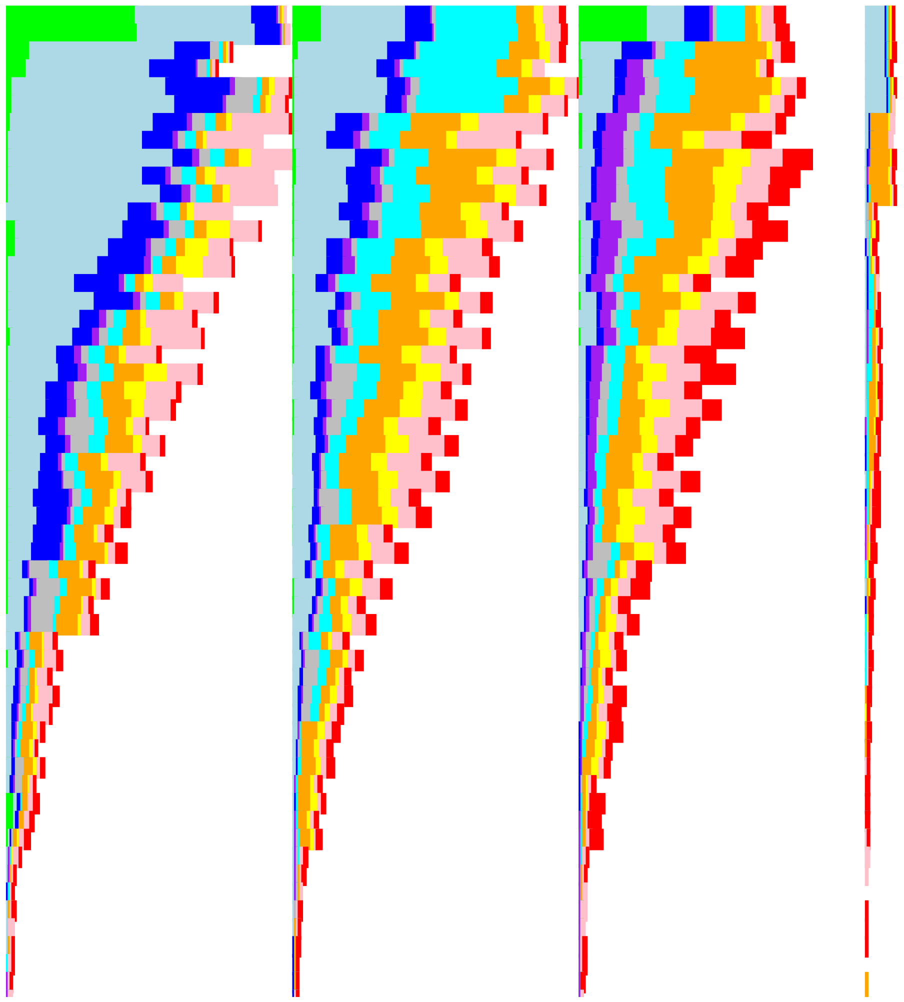

Figure 9: Module Category by Position

Because they are time consuming, usability evaluations and participant observation are generally limited to tens of subjects working on tens of programs. Surveys can provide information from many more subjects, but surveys cannot engage with actual programs, only programmers' opinions and beliefs about their programs [Whitley and Blackwell, 2001]. Probably closest in spirit to our work is the empirical analysis of spreadsheet programs, where accountants or auditors work through a corpus to identify features of programs, such as cell error rates [Panko, 1998]. In the mainstream textual language community, analyses of programs are carried out mainly to improve implementations: analyses and critiques of programming style are generally based on single examples, drawing on literary critisism [Kernighan and Plauger, 1974, Skublics et al., 1996] or the patterns movement [Beck, 1997] for models.

In comparison with usability evaluation or participant observation, an approach based on corpus analysis requires a sample of several hundreds of programs, but does not require detailed analysis of the process by which those programs were written. Corpus analysis is best suited to investigating the *parole* [de Saussure, 1916] of a language — the way it is used in practice — while other techniques can provide more specific information about the design of languages themselves. Corpus analysis has the problem of bias in the selection of the corpus: human studies have analagous problems with the selection of test subjects.

5 Discussion and Conclusion

In this paper we have applied program visualisation to the dataflow visual language for the Nord Modular synthesizer, to investigate modules choice and layout in Modular patches.

Considering the choice of modules within patches, we found oscillators and mixers are used several times in a patch, while sequencers and logic category modules are used rarely. Each patch generally has one input and one output module. Regardless of module category, modules which impose high processing loads are used less frequently than lower power modules.

Considering the layout of those modules, we found that patches tend to be long and thin, promoting vertical rather than horizontal scrolling, and possibly laid out on an 16:2 grid. Within a patch, modules tend to be placed towards the top left, with the absolute top left position typically occupied by an input module. Oscillators and lfos are placed to the left and below them, with filters, envelope generators and mixers in the middle of the patch. Output modules are placed towards the bottom right. This tallies with our previous work on cable direction in modular patches, which found that cables generally flowed rightwards and downwards [Noble and Biddle, 2002].

The key technical advantage allowing us to produce these visualisations, and to perform corpus analysis in particular, is that the Nord Modular patch files are stored in a simple ASCII format. Performing a similar analysis on many other visual languages would be much more difficult, because we would first have had to parse a much more complex binary file.

We do need to note that these results are somewhat preliminary. Each Nord Modular program actually consists of two separate patch areas — a polyphonic voice area (PVA) where modules are duplicated for polyphonic patches and a common voice area (CVA) which is shared across all polyphonic voices. For technical reasons due to the Modular patch file format, our current tools analyse only the polyphonic voice area. We plan to extend our tools to analyse both areas, but do not expect this to markedly impact our results.

We plan further visualisation work on Nord Modular programs: indeed, there seems quite some scope for research since only a small amount of standalone visualisation has been performed upon dataflow visual languages, and very little corpus analysis has performed upon visual langauges of any type. We plan to analyse the use of secondary notation, particularly the names programmers assign to modules. We would like to experiment with providing automatic layout support for modules (to reorganise patches to minimise cable length and cable crossings) and with program slicing (so that all the modules producing one part of a patch could be automatically extracted from a patch making multiple sounds). Finally, we hope to extend this work to analyse many more patches.

References

[Baroth and Hartsough, 1995] Baroth, E. and Hartsough, C. (1995). Visual programming in the real world. In Burnett, M. M., Goldberg, A., and Lewis, T. G., editors, *Visual Object-Oriented Programming*. Prentice-Hall.

[Beck, 1997] Beck, K. (1997). *Smalltalk Best Practice Patterns*. Prentice-Hall.

[Blackwell, 2001] Blackwell, A. F. (2001). Pictorial representation and metaphor in visual language design. *Journal of Visual Languages and Computing*, 12(3):223–252.

[Carroll and Rosson, 1987] Carroll, J. and Rosson, M. (1987). Paradox of the active user. In Carroll, J., editor, *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. MIT Press.

[Citrin et al., 1998] Citrin, W., Ghiasi, S., and Zorn, B. G. (1998). VIPR and the visual programming challenge. *Journal of Visual Languages and Computing*, 9(2):241–258.

[Clavia DMI AB, 1999] Clavia DMI AB (1999). *Nord Modular Manual*. Clavia DMI AB, Sweden, v3.0 edition.

[Cycling ’74, 2001] Cycling ’74 (2001). *MAX Reference*. Cycling ’74.

[de Saussure, 1916] de Saussure, F. (1916). *Cours de linguistique générale*. V.C. Bally and A. Sechehaye (eds.), Paris/Lausanne.

[Desain et al., 1993] Desain, P., Honing, H., Rowe, R., and Garton, B. (1993). Putting Max in perspective. *Computer Music Journal*, 17(2).

[Grundy and Hosking, 1995] Grundy, J. and Hosking, J. (1995). ViTABaL: a visual language supporting design by tool abstraction. In *IEEE Symposium on Visual Languages*.

[Grundy and Hosking, 2000] Grundy, J. and Hosking, J. (2000). High-level static and dynamic visualisation of software architectures. In *IEEE Symposium on Visual Languages*.

[Grundy et al., 1996] Grundy, J. C., Hosking, J. G., and Mugridge, W. B. (1996). Serving up a Banquet: Towards an environment supporting all aspects of software development. In *Software Engineering: Education and Practice (SE:E+P)*, Dunedin.

[Hagen, 1990] Hagen, R. (1990). Blue ribbon soundworks’ bars and pipes professional. `http://www.richardhagen.org`.

[Hansen, 1997] Hansen, G. A. (1997). *Automating Business Process Re-Engineering: Using the Power of Visual Simulation Strategies to Improve Performance and Profit*. Prentice Hall PTR, 2nd edition.

[Helsel, 1997] Helsel, R. (1997). *Visual Programming with HP-VEE*. Prentice Hall PTR.

[Kahn, 1996] Kahn, K. (1996). Toontalk — an animated programming environment for children. *Journal of Visual Languages and Computing*.

[Kernighan and Plauger, 1974] Kernighan, B. and Plauger, K. (1974). *The Elements of Programming Style*. McGraw-Hill.

[NAG, 2000] NAG (2000). *IRIS Explorer User’s Guide*. The Numerical Algorithms Group Limited, Oxford, 5.0 edition.

[Nardi, 1993] Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press.

[NI, 2000] NI (2000). *LabView User Manual*. National Instruments Inc.

[Noble and Biddle, 2002] Noble, J. and Biddle, R. (2002). Program visualisation for visual programs. In Grundy, J. and Calder, P., editors, *Third Australian User Interface Conference (ACUI 2002)*, volume 7 of *Conferences in Research and Practice in Information Technology*, Melbourne, Australia. Australian Computer Society.

[Panko, 1998] Panko, R. D. (1998). What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15–21.

[Repenning and Sumner, 1995] Repenning, A. and Sumner, T. (1995). Agentsheets: A medium for creating domain-oriented languages. *IEEE Computer*, 28(3):17–25.

[Rothermel et al., 2000] Rothermel, K. J., Cook, C. R., Burnett, M. M., Schonfeld, J., Green, T. R. G., and Rothermel, G. (2000). Wysiwyt testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239.

[Skublics et al., 1996] Skublics, S., Klimas, E. J., and Thomas, D. A. (1996). *Smalltalk with Style*. Prentice-Hall.

[Sutherland, 1963] Sutherland, I. E. (1963). Sketchpad: A man-machine graphical communication system. In *Proceedings AFIPS Spring Joint Computer Conference*, volume 23, pages 329–346, Detroit, Michigan.

[Whitley and Blackwell, 2001] Whitley, K. and Blackwell, A. F. (2001). Visual programming in the wild: A survey of LabVIEW programmers. *Journal of Visual Languages and Computing*, 12(4):435–472.