# Work in Progress Report: Nonvisual Visual Programming

Clayton Lewis

*Department of Computer Science*
*University of Colorado, Boulder*
*Clayton.Lewis@colorado.edu*

Keywords: POP-I.A. Learning to program; POP-I.B. Barriers to programming; POP-III.C. Visual languages; POP-IV.B. User interfaces

## Abstract

Visual programming systems are widely used to introduce children and other learners to programming, but they cannot be used by blind people. Inspired by the ideas of blind computer scientist T.V.Raman, the Noodle system provides nonvisual access to a dataflow programming system, a popular model for visual programming systems. This paper describes the design and implementation of Noodle and some of its capabilities. The paper suggests that the same approach used to develop Noodle could be applied to increase the accessibility of other visual programming systems for learners with disabilities.

## 1. Visual programming systems are popular ways of introducing programming to children and other learners.

Despite repeated arguments for their usefulness (e.g. Glinert, 1990; Victor, 2013) visual programming systems have yet to find widespread adoption in a world still dominated by textual languages. But in one important application such systems are now dominant: the introduction of computing to children. Scratch, a visual adaptation of the textual Logo language, has millions of users (Resnick et al., 2009, http://scratch.mit.edu/). Agentsheets, a rule-based language presented in a visual environment rather than textually, is also widely used (Repenning and Ioannidou, 2004, http://www.agentsheets.com/). LabView, a visual programming system originally developed for the control of electronic instruments, is provided as the programming medium for the Lego Mindstorms robotics kits, including WeDO (http://www.ni.com/academic/wedo/), aimed explicitly at younger children (ages 7-11).

These visual programming systems aim to exploit the capabilities of vision to support understanding of program structure and function. Graphical representations and layouts are used to help users recognize the parts of programs, and how they are connected. How well visual systems deliver on these intentions can be questioned; Green, Petre, Blackwell, and others have analysed visual programming systems, with mixed findings (see e.g. Green and Petre, 1996; Blackwell, et al. 2001). Likely the popularity of these languages for children and other learners owes a good deal to the relative simplicity and small scale of the programs typical created by learners, for which the strain on representational systems is reduced.

## 2. Visual programming systems cannot be used by learners who cannot see.

As noted as early as 1990 (Glinert, 1990, p. 4) visual programming systems present a serious challenge for users who cannot see, for obvious reasons. (In fact, visual programming systems also present barriers to sighted users with motor limitations, because of their typical reliance on mouse interaction. The work to be discussed in this paper addresses these barriers as well, but more modest adaptations of existing systems could likely remove or at least greatly reduce these problems, by permitting navigation of their user interfaces via keyboard commands.)

The limitations of these systems mean that blind children are excluded from learning activities that use them. This is deplorable, since programming is an activity that is in other ways quite accessible to

blind people, because of the flexible access provided by machine-readable representations of textual information, including programs. But, one might feel, if visual programming tools make programming easier to learn for other children, surely it must be right to use them. Equally, one might feel, there just isn't any way to deliver the benefits of "visual" representations to learners who cannot see. But there is a way to attack this painful dilemma.

## 3. The Raman Principle suggests that no activities are intrinsically visual.

Blind computer scientist T V Raman has suggested (personal communication, 2009) that a way to think about the visual system is as a way to answer queries against a spatial database. If you have an alternate way to ask the queries and get the answers, you don't need the visual system. While Raman has not expressed this principle, in this explicit form, in his writings, related ideas can be found in Raman (1996) and Raman and Gries (1997). See also Lewis (2013).

A clear illustration of the Raman Principle is an adaptation of the Jawbreaker computer game, created by Raman and Chen (n.d.). As discussed in Lewis (2013), playing Jawbreaker involves clicking on coloured balls, making clumps of balls of the same colour go away, and earning points in a way that depends on the size of the clumps. Anyone seeing the game would imagine that playing it is an intrinsically visual activity, and that no one who cannot see the balls and their arrangement could possibly play the game. But Raman, who is totally blind, can play the adapted version of the game.

To create that version, Raman and Chen analysed the questions that sighted players use their eyes to answer, and added keyboard commands, with speech output, that allow players to ask these questions and receive the answers. For example, the scoring system of the game rewards a player who determines which colour of ball is the commonest when the game begins, and avoids eliminating any balls of that colour until the end of the game. That strategy yields the largest possible clump for removal with one click, an effect that dominates the scoring, since slightly larger clumps give much larger scores. A sighted player would judge which colour is commonest by "eyeballing", if playing casually, or by counting, if playing more carefully. A blind player in the adapted game does it by typing "n", and listening to a spoken report, that there are 13 red balls, 15 blue ones, and so on. Other commands and responses allow the blind player to learn what they have to know about the arrangement of the balls in the game.

The Raman Principle does not assert that the mental processes associated with an activity (playing a game, in the example) will be identical for different presentations. Nevertheless it suggests a way to make an activity supported by visual representations possible for non-sighted users.

### 3.1 The Raman Principle suggests that any visual programming system could have a nonvisual counterpart that is operable without vision.

The logic of the Jawbreaker example can be extended to visual programming systems. For any such system, one can ask, "What information is a sighted user obtaining from the visual display?" One can then provide a means for a non-sighted user to pose a corresponding question, and to receive an answer, using nonvisual media.

## 4. Noodle is a nonvisual dataflow programming system.

In dataflow systems, one popular form of visual programming system, programs consist of functional units, whose values (outputs) may be connected to the inputs of other functional units, and whose inputs may be connected to the outputs of other functional units. Connections are shown visually as lines or curves, representing paths along which data flow from one functional unit to another. Programs are built by selecting available functional units from a palette, placing them in a construction area, and connecting their inputs and outputs.

Noodle provides a system of keyboard commands with speech output that supports creation, execution, and modification of dataflow programs, with visual presentation being optional. That is, Noodle provides a simple diagrammatic representation of programs, mainly for expository purposes,

but it is intended to be used without any use of these diagrams. Noodle is implemented in JavaScript as a Web application.

Noodle's design is based on a task analysis of dataflow programming, influenced by the Raman and Chen work. Their Jawbreaker user interface supports three kinds of user actions: navigation actions, that do not modify the state of the game, but move a point of focus from one location on the game screen to another; queries, that also do not modify the state of the game, but provided particular information to the user; and operations, actions that do modify the state of the game. Adapting this framework to dataflow programming, the design for Noodle provides navigation actions that move a focus between different parts of the programming environment, for example, between the palette and the construction area (called a *workspace* in Noodle), or between functional units (called *blocks* in Noodle) already placed in the workspace. Noodle also provides operations, including commands that cause blocks to be placed in the workspace, that cause data paths (called *edges* in Noodle) to be placed between blocks, that cause blocks or edges to be deleted, and so forth. The current design includes no queries. Instead, all navigation actions produce a description of the current entity of focus (as discussed later, it may be appropriate to add some queries to the design.) All of these descriptions, and in fact all feedback provided to the user, are presented in audio form, using text-to-speech conversion (using the speak.js package, https://github.com/kripken/speak.js).

Table 1 shows a trace of the creation and execution of a trivial Noodle program.

| | |
|---|---|
| user types "n" [“new” command] | user types → |
| system says:type name of new function | system says:palette playSound |
| user types "foo" | user types → |
| system says:creating new workspace foo | system says:heading sound operations |
| user types ← | user types → |
| system says:palette noteC | system says:workspace foo playSound |
| user types / [select] | user types → |
| system says:adding block noteC after externalIn | system says:input connector buffer on block playSound with no edge. |
| user types → | user types / [select] |
| system says:palette noteC [see note in caption] | system says:connector selected |
| user types ← | user types ← |
| system says:heading sounds | system says:input connector buffer on block playSound with no edge. |
| user types ↓ | user types ← |
| system says:heading sound operations | system says:workspace foo playSound |
| user types ← | user types ↑ |
| system says:palette amplify | system says:workspace foo noteC |
| user types ↓ | user types → |
| system says:palette attenuate | system says:output connector buffer on block noteC with no edge. |
| user types ↓ | user types / [select] |
| system says:palette mixSounds | system says:adding edge from connector buffer on block noteC to connector buffer on block playSound |
| user types ↓ | user types "r" [“run” command] |
| system says:palette concatSounds | system plays tone for middle C |
| user types ↓ | |
| system says:palette playSound | |
| user types / [select] | |
| system says:adding block playSound after noteC | |

*Table 1. Transcript of interaction during creation and execution of a program that plays a tone. Note: When an arrow key is pressed after a command other than a navigation command, the current location is described but not changed, as an aid to orientation.*

## 5. Design and Implementation of Noodle.

### 5.1 Noodle uses a model-view architecture, with a declarative representation of programs.

The foundation of Noodle is an interpreter for dataflow programs that accepts programs represented as JSON (JavaScript Object Notation) strings. This declarative representation, or model, can be viewed graphically, and in fact could be created and manipulated via mouse interaction, in early versions of Noodle. This same representation can be inspected and manipulated via a completely different user interface, the completely nonvisual interface just described, that uses only keyboard interaction and audio output. Thus a single model can be the subject of multiple views.

This architecture permits multiple user interfaces to be explored easily. Work on Noodle has included four different interfaces: a conventional visual interface, now available for output only, for exposition; the completely nonvisual interface that is the main subject of this paper; an earlier nonvisual interface that used a larger, more complex set of keyboard commands (described in Lewis, 2013); and a button-oriented interface intended for use on phones (currently incomplete.)

### 5.2 Noodle uses a pseudospatialized navigation scheme to reduce the number of keyboard commands.

The first version of Noodle used a large number of keyboard commands for navigation. For example, separate commands were used to move up and down in the palette, to move up and down in the program construction area, to move among the connectors on blocks, and to follow edges. These operations required a total of eight commands.

To reduce this load (both from learning the commands, and from locating the commands on the keyboard, during use) four generic spatial movement commands (using arrow keys) are defined in the current Noodle user interface. Roughly, the horizontal movement keys move the focus between virtual columns (no spatial layout is actually shown), and the vertical keys move the focus up and down the columns.

The arrangement is "pseudospatial" rather than fully "spatial", not only because no spatial layout is shown, but also because some uses of the keys do not correspond to spatial moves in any simple way. For example, one could think of the palette and workspace as "columns", one to the left of the other, so that one moves “right” or “left” between them, and “up” or “down” within them. But moving to the "right" from a block in the workspace leads to the connectors available on that block for the attachment of edges. Moving further "right" from a connector, if there is a edge attached, leads onto that edge, and moving "right" again leads to the connector on the far end of the edge, which is on some other block in the program construction area. This is plainly not actually "spatial", since moving to the "right" can eventually bring one to a point "above" or "below" the starting point.

### 5.3 Following an edge shows a contrast between nonvisual and visual dataflow programming.

A key operation in understanding a dataflow program is tracing the data paths that connect the blocks in a program. In visual representations of dataflow programs this is done by tracing lines or curves from one unit to another. While conceptually straightforward, this tracing is often quite difficult to do in practice. Paths are generally not straight, and, worse, frequently cross one another. In complex programs several paths may run in parallel, close together, making it easy to slip over from one path to a different one, while tracing.

Noodle's navigation scheme eliminates this problem. Tracing an edge from one unit to another can be done directly, by moving onto the relevant connector, thence onto the edge, and on to the connector on the other end of the edge. The situation of other edges cannot interfere at all with this process.

## 6. Capabilities of Noodle

### 6.1 Noodle supports simple sound processing.

To learn to program one needs to know what programs are doing. Guzdial's (2003) work on media computation suggests that good learning environments allow learners to create media content, such as graphics and sound. For blind learners sound is an attractive choice, and for that reason Noodle includes primitive support for sound processing. Available blocks include ones that produce common musical notes, that mix and concatenate sounds, and that allow the frequency and length of sounds to be under program control.

### 6.2 Noodle offers the potential of programming on small screen devices.

Current programming systems for phones require large screens. While at least one effort offers application development in Android devices (https://play.google.com/store/apps/details?id=com.aide.ui) the aim seems to be to support tablets with at least modest screens, rather than phones with small screens. The Noodle user interface could be supported with no screen at all, and work has been done on a version intended to be run as a mobile Web app for phones.

## 7. Nonvisual versions of other visual programming systems should be developed.

Task analysis, as suggested by the Raman Principle, could be applied to the design of alternative user interfaces for visual programming systems that use paradigms other than dataflow. Here are a few illustrative suggestions.

Scratch uses visual cues to distinguish semantic categories and simplify syntactic constraints. These visual cues are the shapes, like the shapes of jigsaw puzzle pieces, that allow a visual judgement to be made that a given piece of program will or will not fit into a program under construction at a given place. If the user tries to place a piece of program in a wrong place, it will not fit. A major reason for the popularity of Scratch, as compared to the older Logo language, is that users do not have to learn complex textual syntax rules, and are not vulnerable to typing errors that produce syntactically invalid code. How could these benefits be secured for users who cannot see?

In a nonvisual version of Scratch, nonvisual navigation commands could traverse gaps in a program, that is, places in a program under construction where new material could be added. From such a gap, other nonvisual navigation commands could directly access palettes of compatible program elements. That is, instead of the user identifying the shape of a gap, and then visually matching the shape of the gap against the shapes of available units in part of the palette, the user could move directly to the relevant section of the palette. Navigation in this nonvisual scheme could be easier than in the current visual scheme, for all users. If this proved to be the case, it could be added to the current Scratch user interface, rather than creating a different, alternative interface.

Another aspect of Scratch that is not workable for blind users is program assembly by visually guided dragging. For example, given a sequence of commands, the user can drag a "repeat" unit up to them, so that it deforms to fit around them, forming a loop with the sequence of commands as the body. Nonvisual placement actions could be provided to support this action. A user would select the beginning and end of the loop body, and then select the repeat unit, and the loop structure could be completed automatically.

Most uses of Scratch today produce graphical, animated output, which would be problematic today for blind learners. In time, support could be contrived for interpreting graphical output nonvisually, as was suggested above for Noodle.

Like Scratch, Agentsheets uses separate palettes for program pieces that play different roles, for example separating conditions and actions for rules (Agentsheets programs are collections of if-then rules.) As for Scratch, nonvisual navigation commands could support this access.

As for Noodle and Scratch, supporting understanding of dynamic graphic behaviour, as well as static arrangements of elements, is an important challenge. Agentsheets has "conversational programming" features that provide (visual) explanations of some aspects of dynamic program behaviour, such as which conditions of rules are currently satisfied. These features might be adapted to provide spoken commentary on program execution, and this enhancement could be of value to sighted users, too. More generally, other design features of nonvisual presentations, such as easier tracing of connections between program elements, or easier access to relevant palette items from a program under construction, could be added to existing visual languages.

## 8. Conclusion

Nonvisual visual programming is possible, and offers potential benefits to learners who cannot see. Its development may also offer benefit to other users, as is common for work on inclusive design.

## 9. Acknowledgements

Thanks to Antranig Basman, Colin Clark, Greg Elin, Jamal Mazrui, T V Raman, and Gregg Vanderheiden for useful discussions and encouragement, and to anonymous reviewers for helpful suggestions.

## 10. References

Blackwell, A.F., Whitley, K.N., Good, J. and Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review* 15(1), 95-113.

Glinert, E. (ed.) (1990) *Visual programming environments: Paradigms and systems.* Los Alamitos, CA: IEEE Computer Society Press.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2), 131-174.

Guzdial, Mark (2003.)A media computation course for non-majors. *SIGCSE Bull.* 35, 3 (June 2003), 104-108.

Lewis, C. (2013) Pushing the Raman principle. In *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility* (W4A '13). ACM, New York, NY, USA, Article 18, 4 pages.

Raman, T.V. (1996) Emacspeak-- A speech interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '96), Michael J. Tauber (Ed.). ACM, New York, NY, USA, 66-71.

Raman, T.V. and Gries, D. (1997.) Documents mean more than just paper! *Mathematical and Computer Modelling*, Volume 26, Issue 1, July, 45-53.

Raman, T.V. and Chen, C.L. (n.d. ) AxsJAX-Enhanced Jawbreaker User Guide. Retrieved from http://google-axsjax.googlecode.com/svn-history/r540/trunk/docs/jawbreaker_userguide.

Repenning, A., & Ioannidou, A. (2004). Agent-based end-user development. *Communications of the ACM*, 47(9), 43-46.

Repenning, A. (2013) Conversational programming: exploring interactive program analysis. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (*Onward!* '13). ACM, New York, NY, USA, 63-74.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009) Scratch: programming for all. *Communications of the ACM* 52, 11 (November 2009), 60-67.

Victor, B. (2013) The future of programming. Retrieved from http://worrydream.com/#!/TheFutureOfProgramming