Behavior Processors: Layers between End-Users and Java Virtual Machines

Alexander Repenning and Andri Ioannidou

Department of Computer Science
Center for LifeLong Learning and Design
University of Colorado, Boulder CO 80309-0430
(303) 492-1349, {ralex, andri}@cs.colorado.edu
Fax: (303) 492-2844
http://www.cs.colorado.edu/~ralex/

Abstract

Visual programming approaches are limited in their usefulness if they do not include a profile of their users that defines exactly who is attempting to solve what kind of problems using which tools and why. Without such a definition, visual programming approaches can end up as solutions in search of problems. Reconceptualizing — programming environments as layered *behavior processors* in the context of creating SimCity™-like interactive simulations — makes end-user programming more feasible. The layered approach serves the programming needs for a range of users, including casual computer end-users and professional programmers. The extension of the Agentsheets system with the Ristretto™ *agent to Java bytecode compiler* is used to illustrate how a behavior processor enables end-users to create their own Java applets that can be embedded into web pages.

Behavior Processors: Layers between End-Users and Java Virtual Machines

Alexander Repenning and Andri Ioannidou

Department of Computer Science
Center for LifeLong Learning and Design
University of Colorado, Boulder CO 80309-0430
(303) 492-1349, {ralex, andri}@cs.colorado.edu
Fax: (303) 492-2844
http://www.cs.colorado.edu/~ralex/

Abstract

Visual programming approaches are limited in their usefulness if they do not include a profile of their users that defines exactly who is attempting to solve what kind of problems using which tools and why. Without such a definition, visual programming approaches can end up as solutions in search of problems. Reconceptualizing — programming environments as layered *behavior processors* in the context of creating SimCity™-like interactive simulations — makes end-user programming more feasible. The layered approach serves the programming needs for a range of users, including casual computer end-users and professional programmers. The extension of the Agentsheets system with the Ristretto™ *agent to Java bytecode compiler* is used to illustrate how a behavior processor enables end-users to create their own Java applets that can be embedded into web pages.

Who is the programmer?

With the next millennium in sight, the visual languages community is starting to take inventory of its successes and failures in order to develop new directions in which research can explore new ideas to increase the effectiveness of visual programming approaches. Several classification schemes [10, 18] have been proposed to structure a complex space populated with a variety of programming approaches. A nice and increasing set of cognitive dimensions [7] has not only been proposed but has also been used to evaluate and contrast both existing and hypothetical programming systems.

Taxonomies and cognitive dimensions are important instruments of analysis, but they provide little information about the actual users of visual programming environments. Specifically, it is important to understand the backgrounds of users, their motivations, and their needs, and to get at least a sense of the problems they want to solve using visual programming environments. Visual language "superlativism" — that is, the bias toward overly optimistic and general assertions about the positive value of visual programming, excellently surveyed by Blackwell [2] — is partly due to the lack of specific statements about who is trying to solve what problems and how, as well as the lack of empirical studies that evaluate the performance of users solving problems.

Knowledge of the mind sets, attitudes, and skills of end-users empowered by visual programming is important to understand the role of programming environments. For instance, most VCRs can be programmed, but VCRs are not considered programming environments, nor do most people using VCRs think of themselves as programmers enjoying the process of programming. For the majority of computer end-users the very notion of programming is daunting and, at the same time, completely secondary to the solution of a problem, such as taping a specific TV program. It is exactly this kind of programmer, who does not want to program, for whom visual programming could potentially make the biggest difference. Currently there are 90 million users of PCs — machines that are programmable — but the question is why should they want to program and what hope would they have to be able to program? Many people argue that programming a VCR is already so difficult that programming a much more complex device, such as a computer, would be an even more intricate process for that they see little hope of achieving. The following discussion refers to these kind of "forced" programmers, but by no means implies that visual programming should completely focus on them. In the past, some visual programming environments such as LabView [8] and Prograph [6] have successfully supported skilled programmers.

As a programmable machine, the computer holds enormous potential as the ultimate flexible medium — a kind of computational clay — for people to express themselves or to communicate information to other people in new ways. The idea of using or even creating a computer as a simulation or game environment is appealing to many people [21], but, unfortunately, the current programming approaches — visual or not — place the ability create these things out of reach for the majority of people.

We believe that in order to make the computer a more malleable medium in the next millennium it is necessary to get a much better sense of exactly who these people that we are trying to empower with visual (or other kinds of) programming really are. We may have to reconceptualize the very process of programming and its role in society. While cognitive dimensions and programming approach taxonomies are necessary instruments, we may have to increasingly draw from other fields of study such as anthropology, ethnography, and sociology.

This paper can be little more than a small step toward this kind of goal. However, by drawing from experiences with a diverse group of users ranging from middle-school children creating educational simulations of the world (in which they live) to environmental design professionals simulating issues of sustainability in city planning, we hope to start a process of reconceptualization. Specifically, this paper outlines the notion of *bricolage* as a way to think about programmers and programming, and uses the Agentsheets system [15-17] to illustrate how bricolage can be supported with a so-called *behavior processor*.

Bricolage

The understanding of the intricate relationships among people, tools, and problems should not be limited to cognitive and technical issues, but should include what Papert calls *intellectual style* [12]. Papert and Turkle use the term "bricolage" to describe intellectual style in an educational context in which children program in Logo. The term bricolage was originally introduced by the French anthropologist Claude Lévi-Strauss to describe the process of theoretical tinkering by which individuals and cultures use objects around them to assimilate ideas. Papert defines bricolage as an organizational style that can be described as negotiational rather than planned in advance. Bricoleurs are "tinkers using what they got, improvise, and make do."

Bricolage, the process of gradually using components and building them up to larger and larger structures, is often found in environments such as spreadsheets. From a software engineering point of view, large spreadsheets maybe be declared to be hopeless disasters because they have organically grown bottom-up into increasingly complex webs of formulas to a point where the creator is the only person able to maintain them. While this approach does lead to a number of problems [22], it generally gets the job done for millions of spreadsheet users.

The point here is not to judge bricolage as good or bad with respect to programming style, but to acknowledge that most people for whom programming is only a means to an end (e.g., the VCR or spreadsheet programmer), will resort to some kind of bricolage. This may be due to the lack of formal software engineering training but can also simply be the result of having other, more important things to do. These users are not likely to employ elaborate analytical top-down planning processes to create working programs. These circumstances leave user manuals unread, resulting in "let's see what happens if I press this button" approaches. For typical Agentsheets users, the same situation holds true. That is, their main interest is to get a simulation running and not to write an elegantly generalizable program. Could this be a taste of what is to come for the next generation of end-user programmers? Turkle maintains that in the 1990s, as computing shifts away from a culture of calculation, bricolage has been given more room to flourish [21]. The emerging question is how a programming environment can accommodate bricolage.

A first step toward creating a bricolage environment is to reconceptualize programming as a more domain-oriented process [5] that is more closely related to the actual problems to be solved. It is crucial to rethink the process of problem solving in this specific context and not just to find a substitute for the stigmatized programming word. In the context of creating simulations, we propose the term *behavior processing,* referring to the process of creating and using behavior units called agents. Analogous to a word processor, a behavior processor aids users in their process of dealing with behaviors. Word processors define a number of operations on words such as typing, spell checking, formatting, rearranging, and copying. Similarly, behavior processors define operations on behaviors as the basic units. Such operations include, but are not limited to, defining, modifying, merging, and exchanging behaviors. At the moment, these operations elevate behavior processing only slightly over traditional programming. From today's perspective WYSIWYG word processors seem to be obvious, but it took many years to develop the concept of a word processor.

Behavior Processors

The role of a behavior processor is to support bricoleurs in creating, modifying, combining, and sharing behaviors. The Agentsheets system [15-17] is a behavior processor supporting a wide range of users to create SimCity™-like interactive simulations. These simulations can be compiled down into Java applets that can be embedded directly into web pages. An agentsheet is an agent-based spreadsheet typically containing a large number of agents interacting with each other in ways similar to the interactions between cells in a spreadsheet. Each agent consists of behavior and look. Figure 1 below shows an Agentsheets application called "Der Packmann," which features a number of agent types, such as pacmans, monsters, pills, and walls.

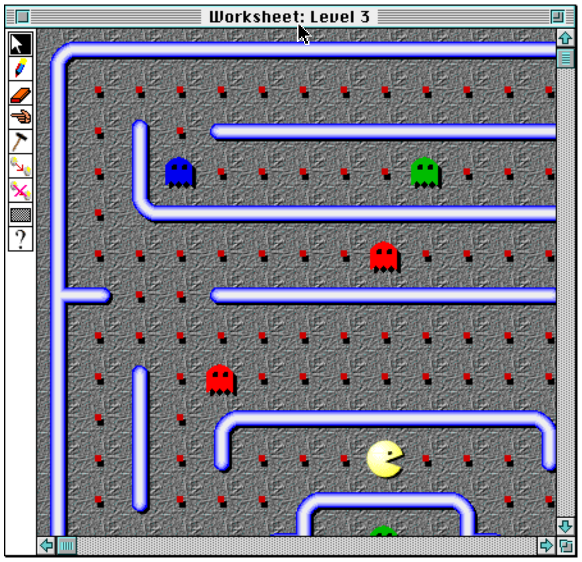


Figure 1: Agentsheet containing pacmans, monsters, pills, and wall agents

To illustrate the operation of a behavior processor, we use an analogy to word processors. Both word processors and behavior processors bridge a wide gap between human cognition and technology. The word processor supports the high-level (typically WYSIWYG) direct manipulation of words in order to create a document. The word processor can render an electronic version of a document into a more tangible form using a document description language such as Postscript. A Postscript file can be sent to a printer containing a Postscript interpreter which renders the description language into printed pieces of paper. Analogously, the behavior processor supports the direct manipulation of behaviors in order to create an interactive simulation or a game. The simulation, such as the pacman application of Figure 1, can be run in Agentsheets directly; that is, a user can plan the Pacman game or the behavior processor can render it into an applet consisting of Java byte codes. The applet, in turn, can be run on any hardware/software platform featuring a Java virtual machine [9]. In this sense it can be claimed that Java plays a role to networking very similar to the role Postscript plays to printing.

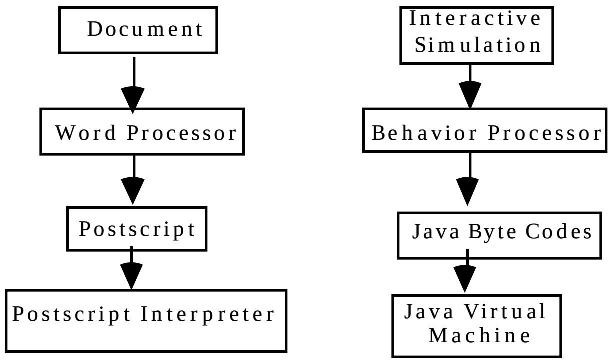


Figure 2. Layers of word processor and behavior processor

In order to be effective for a wide range of applications, a behavior processor must be:

- **usable:** allowing the bricoleur to solve simple problems in a simple way. For instance, to do frequent operations, such as moving an agent, the behavior processor must offer a recognizable solution. The behavior processor should not require the bricoleur to assemble fundamental operations first from lower-level operations.
- **expressive**: with some additional effort it should still be possible to solve nontrivial and difficult-to-predict problems.

These two characteristics represent a trade-off [15]. Programming approaches such as C++ are highly expressive, but are geared toward professional programmers. End-user programming approaches such as graphical rewrite rules [14, 20] are highly usable but are very limited in their ability to deal with more complex applications.

Instead of striving for a single point — the Holy Grail — in the usability versus expressiveness space, the position taken here is that behavior processors need to be structured as *layers* providing end-users some choices in the usability versus expressiveness space.

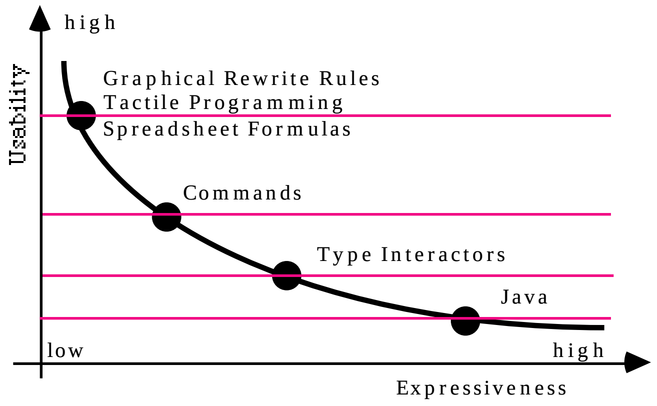


Figure 3. Four points in the Usability/Expressiveness space

VCRs are examples of programmable devices providing different usability versus expressiveness levels. Most modern VCRs include some kind of fold-down panel proving users access to a more complex set of operations. With the panel closed, the Start, Stop, Forward, and Rewind buttons are dominant in the user's view of the VCR. Once the panel is folded down, additional, more complex choices are provided, often including functionality to set the time or to record programs.

The following sections illustrate the four layers of the Agentsheets behavior processor representing four points in the usability/expressiveness space (Figure 3).

Layer 1: Visual AgenTalk and Visual AgenTalk Formulas

The highest level layer closest to the direct needs of the bricoleur. Activities on this layer would be equivalent to editing text in a word processor or defining simple formulas in a spreadsheet. The role of the behavior processor is to elevate the program representation from a textual or visual representation to the status of a user interface. In other words, in its elevated form, *the program is the user interface*. This kind of interface in support of an exploratory style of programming/behavior processing is necessary for bricolage.

To be best suited for bricolage, a behavior processor has to allow users to "play" with the language and, explore its functionality. Agentsheets includes a tactile programming language [15] called Visual AgenTalk, which provides language objects called *commands* packaged up with user interfaces. We think of tactile programming as a direct continuation of our work on graphical rewrite rules in Agentsheets [13] and of our collaboration with Apple Computer on KidSim/Cocoa [19].

Tactility is used here not in the sense of complex force feedback devices that are hooked up to computers, but much more in the sense used by Papert to explain the closeness of bricoleur programmers to their computational objects [12]. One departure from Papert's framework is that the notion of computational objects in Visual AgenTalk is not limited to the objects that are programmed, such as the Logo turtle, but also applies to the programming components themselves, which are elevated to the level of highly manipulatable objects.

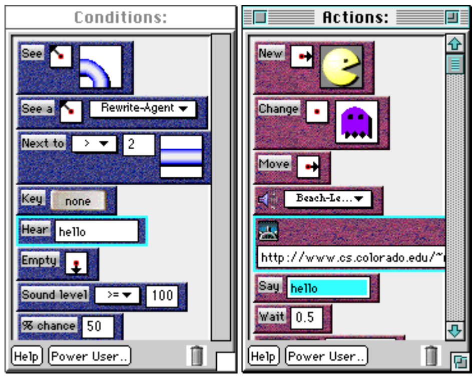


Figure 4. Command palettes: conditions (left), actions (right)

Tactile programming is more than "drag and drop" because the operations it enables are not used just as construction mechanisms (e.g., move, copy, delete language pieces) but also as a means of direct exploration. At any point in time users can explore the functionality of any command by simply dragging and dropping them onto any agent in their simulation. A simple case of exploration is, for instance, dragging the move command from the action command palette (Figure 4, right) onto a monster agent, which will make the monster move to the right.

Condition commands, when dragged and dropped onto agents, will reveal whether the condition holds for the agent testing the condition in its current context. If the Next-To condition (in Figure 4) is dragged onto the red monster in Figure 1, visual and acoustic feedback will immediately indicate that this condition would not hold.

Permanent behavior can be created when these commands — and not just playing with the conditions and actions — are combined into rules that can be tested using the same mechanism of exploration. One of the many pacman rules defines that if the cursor-up key is pressed the Pacman changes its direction to be facing up. Dragging and dropping the rule onto a pac man will test the entire rule. Step-by-step with visual feedback, all the conditions are checked (there is only one in this example rule). If, for instance, the cursor-up key on the keyboard is not pressed, acoustic feedback is provided and the condition that fails blinks to indicate the problem. Repeating the test while holding down the key will successfully match the condition and, as consequence, execute all the actions — in this case, changing the direction attribute and the depiction of the agent.

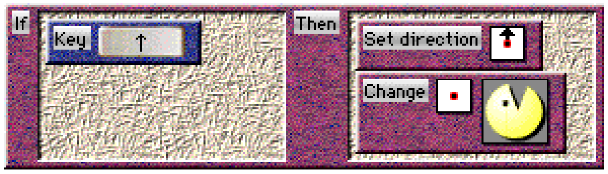


Figure 5. Rule: if cursor-up key is pressed, set new direction to up and change depiction to pacman facing up

The ability to test a single rule out of a possibly large set of rules is important for the bricoleur. Without this ability rule-based systems become difficult to debug and can quickly reach a critical mass of complexity that is not due to the rule-matching mechanism of the system but due to the cognitive complexity posed by the number of rules. Even if a user with a problem suspects a certain rule to be faulty, it can become extremely tedious to test that rule without such a mechanism. The user may be forced to set up complex situations or temporarily disable or even delete rules preceding the rule in question.

Bricolage is a form of *guided exploration* at different levels. At the level of commands, guidance arises from the quick feedback provided by the drag-and-drop testing. Very quickly, bricoleurs can determine, without having to first write a complete program, the applicability of specific conditions and actions to their problems. The same principle of guided exploration holds true for command parameters. For instance, the SEE condition (Figure 6) tests for the presence of an agent with a certain depiction in a specific direction. Direction as well as depiction are parameters to the command that are set by the user via direct manipulation. This is important because, as pointed out by Nardi [11],the way parameters are specified can affect the extent to which the programmer must learn a syntax.

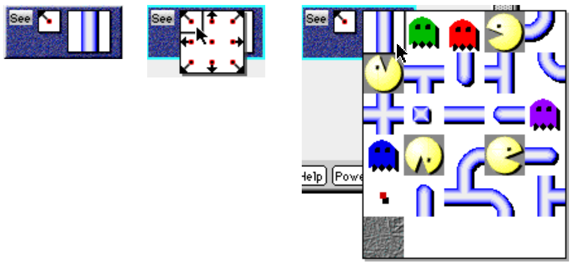


Figure 6. "See" command (left), manipulating direction parameter (middle), manipulating depiction parameter (right)

The integration of parameters that are directly manipulatable, such as the 2D pop-up dialogs for direction and depiction, elevate the program onto the level of a user interface combining ideas of form-based interfaces [1] with end-user programming.

Command parameters do not necessarily have to be constrained the way the direction and depiction 2D pop-ups work. The parameter used in the key condition is defined by clicking at it and then pressing some key on the keyboard. Even more flexible are Visual AgenTalk Formula parameters which can be defined like spreadsheet formulas. For instance, in a digital filter Agentsheets application, formulas are employed to process real-time signals.

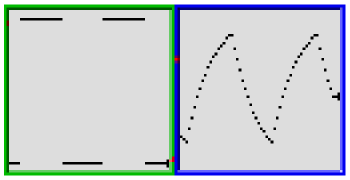
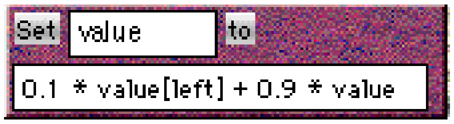


Figure 7. Two plotter agents. The left one outputs a square signal. The right one filters it.

For example, the agent to the right in Figure 7 accesses the signal to its left and filters it with a simple formula. The output value of this low-pass filter is defined as the weighted sum of the value to the left and its old value (Figure 8).



Set value to

0.1 * value[left] + 0.9 * value

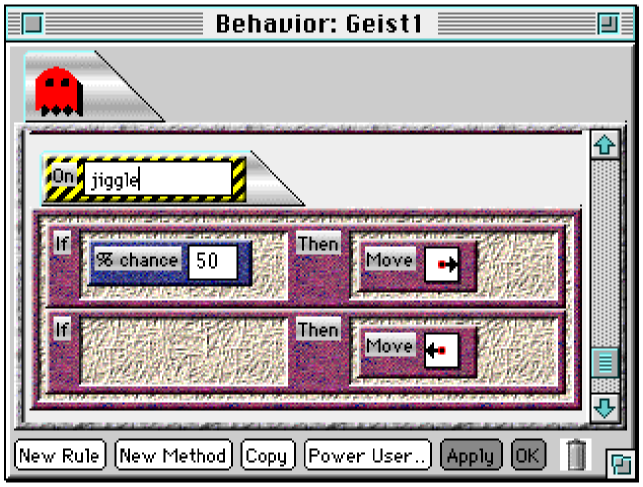
Figure 8. Attribute value is defined as formula with spatial reference to value attribute of agent to the left

Layer 2: Making Commands

Agentsheets provides a default set of commands (conditions, actions, triggers) that are expressive enough to allow the creation of fairly complex simulations, such as a Sim-City™ simulation. Even so, that set of commands is limited. As users get more experienced with the system, they tend to want to create more complex behaviors that would either be impossible with the existing set of commands or would require a round-about way, usually involving a large number of rules. In most cases, the introduction of a new command would simplify this process significantly.

The use of these custom commands is similar to the function of control panels found on some VCRs. When folded down these panels provide access to a number of additional, typically much smaller and more exotic buttons controlling less frequently used operations such as setting the time or programming a recording. Only a subset of users either requires these operations or may actually be interested in using them but is not willing to invest time into learning how to use them. The same holds true for command design. The ability to design commands results in increased expressiveness at the cost of usability.

In simple cases, bricoleurs can define their own language extensions by aggregating a number of rules into a new rule group and giving it a name. This named rule group can then be invoked with a special action such as "jiggle" in Figure 9. More complex cases require completely new commands.



Behavior: Geist1

On jiggle

If % chance 50 Then Move

If Then Move

Figure 9. A rule group named "jiggle" when called makes a monster randomly go left or right

In one of our Agentsheets test sites, the Centennial Middle School in Boulder, Colorado, a group of sixth- to eighth-grade students was working on a simulation of the Pearl Street Mall, one of the popular areas in Boulder. The students decided to build their game in the form of a scavenger hunt, in which a user-controlled tourist walks around the mall and encounters interesting characters, such as jugglers , bikers, , people who follow him around, even pick-pockets . The tourist's mission is to gather a number of items, that can be bought from a variety of stores located on the mall, given a initial budget. It was soon clear that the existing set of commands would not suffice for this game. There was a need for trade commands, such as ways for shops to set up and display their inventories and, naturally, means for the tourist to buy different things from shops. Therefore, we created a set of new commands (Figure 10), and gave them to the students to use.

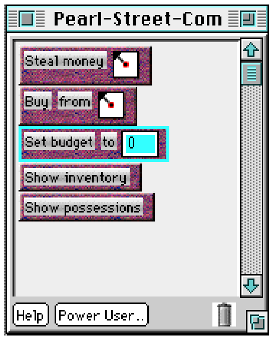


Figure 10: New commands created for the Pearl Street Project

Yet more complex cases require access to the lower levels of the behavior processor, in some cases reaching as deep as creating commands to access low-level operating system functions. These extensions, however, cannot typically be implemented by the bricoleur. An example of such a command is the WWW Read command which allows agents to read and parse web pages in real time. The command shown in Figure 11 searches a web page for the expression "Wind Speed:" and parses the following word as a number into an agent attribute called "speed." This kind of functionality allows agents to reach out from simulations to real-world signals via the internet. The web page referred to in Figure 11 is updated every 5 minutes.

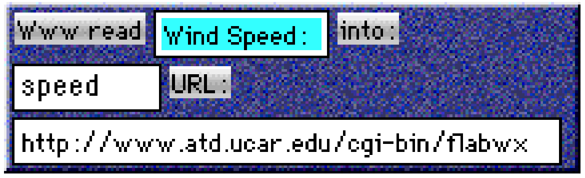


Figure 11. Command allowing agent to read web pages in real time and extract numerical value

A command designer needs to specify the command parameters and the function (in Lisp) that will be called when the command is executed. The specification of the parameter types is sufficient to generate the user interface for the command. That is, designers need not worry about how to lay out the parameters interactors' nor do they have to provide code for making parameters editable. The WWW-FIND make calls additional functions to implement TCP/IP protocols, parsing, and matching:

```
(defcommand WWW-FIND ((String string-type) "URL:" (URL url-type))
  :type condition-command
  :macro `(find-web-page-string-in-page ,URL ,String))
```

Commands of this complexity need to be created by experts. However, through the open architecture of Agentsheets, users can freely exchange such custom commands via the web or email.[16] .Commands fulfill a role comparable to macros in spreadsheets or VBXs, sharable components programmed in Visual Basic. However, the biggest difference is that, in contrast to a macro, a command not only provides raw functionality but wraps up functionality as language accessible through a direct manipulation user interface suitable for end-users. Comparable trap door mechanism in systems such as Rehearsal World [4] or ThingLab [3] also provide access to the underlying programming language — Smalltalk in both case — but do not include end-user interface generators to wrap up new language pieces. LabView [8] and Prograph [6] provide interactors for individual data types (e.g., a push button switch for a boolean type) but do not aid the mapping of entire type signatures.

Layer 3: Making Type Interactors

Agentsheets allows the design of custom type interactors. A large set of built-in parameter types, such as the URL--TYPE from the WWW read command (Figure 11), are automatically translated by Visual AgenTalk to so-called type interactors. These types interactors are direct manipulation mini-widgets used to edit values of that type. Existing type interactors are used to define sounds, MIDI values, keyboard key codes, numbers, depictions, directions, etc. With the necessary skills, users can define their own type interactors and consequently express new commands based on these type interactors.

A number of visualization projects required the definition of commands referring to color. An initial set of color commands represented color parameters as editable text fields to enter RGB values. In hindsight, and not very surprisingly, the RBG-as-numerical-values approach was not very well received by end-users and led to confusion. A new color-type interactor pops up a color palette when clicked and allows the user to sample any of the colors contained or even to sample other colors contained anywhere else on the screen.

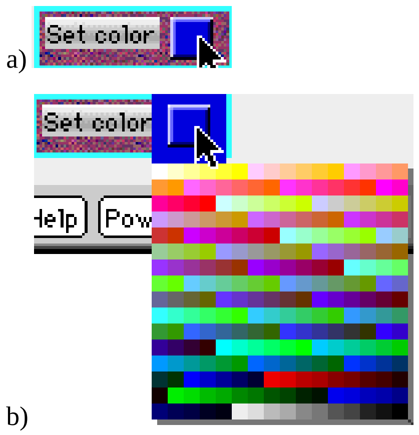


Figure 12. Clicking at Color-type interactor (a) allows interactive color definition via a pop-up palette (b)

Designers of type interactors need to have substantial knowledge of user interface implementation issues and need to adhere to a type interactor protocol by providing methods defining the size of the type interactor, click actions, and a value returned.

Layer 4: Ristretto™ the Agent to Java Compiler

While Java enables its users to create interactive applets and make them publicly available in their web pages, it is still a programming language that requires expertise that usually only professional programmers possess, and it is neither available to nor desirable for end-users. We claim that in behavior processing, Java is at the level that Postscript is in word processing. When users create a document in word processing and want to print it out, they just push a button that does exactly that. It is of little or no interest to them that, in order to be printed, the document is first translated into Postscript and sent to a printer where a Postscript interpreter creates the printed version. Just as users of word processors do not have to learn Postscript to print their document, end-users in behavior processing should not have to learn Java to create an applet out of their simulations.

As a behavior processor, Agentsheets provides the Ristretto™ compiler layer to allow end-users to create Java applets out of their simulations, without having to learn Java, or even worry about its existence. Suppose a user created a Fish Tank simulation in Agentsheets in which fish swim around and sharks eat fish. In order to create the same simulation as a Java applet for the user to include in his or her homepage, it would probably take one or two days, given that the user is a professional programmer. With Ristretto, however, this process becomes a matter of minutes for end-users as well as more sophisticated users. Once the users are happy with their simulation, by pressing a single button, Ristretto generates a complete Java applet that can be embedded in web pages and then be accessed remotely through the internet by other users. In just a few seconds, Ristretto compiles every agent behavior directly into Java class files consisting of Java bytecodes and compiles agents depictions into GIF files. Translating agent rules directly into Java bytecode results in very efficient applets because the only run-time interpretation left is in the Java virtual machine.

The applet created is ready to run on a large number of different Java runtime support environments, including Netscape Navigator (Figure 13), MS Explorer, JDK Applet Runner, and MRJ Apple Applet Runner, allowing the simulation generated to be used on a wide range of hardware platforms.

™ Ristretto is tradmaked by Agentsheets Inc.

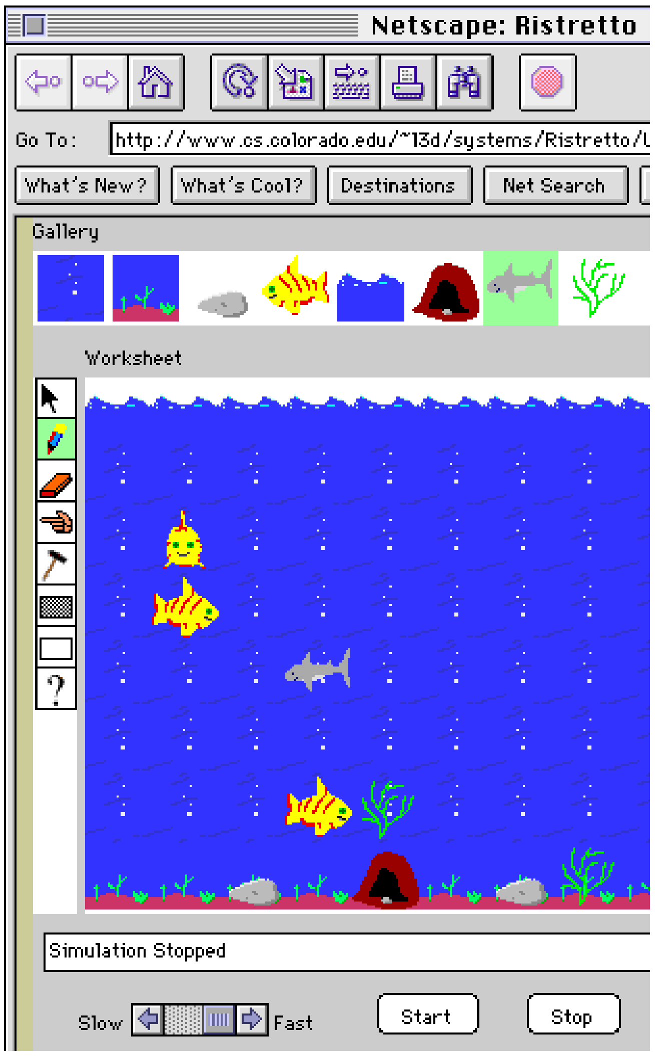


Figure 13. Fish Tank applet generated with Ristretto agent-to-Java-byte-code compiler

The Ristretto-generated applet allows its users to edit the simulation world. Users can add and remove agents (in the fish tank: water, ground, rocks, small fish, waves, caves, sharks, and weeds), can start and stop the simulation, and can change the speed at which the simulation is running. Furthermore, the Ristretto-generated applet can be controlled through JavaScript or interact with other applets through interapplet Java functions.

Elevating the process of programming a Java applet to behavior processing, as described above, presents end-users with previously unavailable opportunities to harness the power provided by the Java language, such as the cross-platform capability and the creation of applets that can run on web pages, without ever having to learn how to program in Java.

Conclusions

The claim put forward in this paper is that for visual programming — or any kind of programming, for that matter, — to be effective, it is crucial to know the context in which the programming process is taking place. In other words, what is the social context, — the circumstance in which nonprofessional programmers create programs? Why are they doing it and what kind of problems are they trying to solve?

This paper has introduced the notion of a behavior processor that allows end-users to create complex interactive simulations and turn them into Java applets without first having to become full-fledged programmers.

Acknowledgments

This work has been supported by the Advanced Research Projects Agency in the Technical Reinvestment Program, and the National Science Foundation under grant number REC-9631396. Many thanks to Jim Ambach, who has worked on the Java version of Agentsheets, and to Thilo Hübner, author of the Agentsheets pacman game featuring a still unbeaten level 3.

References

1. Ambler, A. L., and M. M. Burnett, "Influence of Visual Technology on the Evolution of Language Environments," *IEEE Computer,* pp. 9-22, 1989.

2. Blackwell, A., "Metacognitive Theories of Visual Programming: What Do We Think We Are Doing?," *Proceedings of the 1996 IEEE Symposium on Visual Languages*, Boulder, Colorado, IEEE Computer Society, 1996, pp. 240-245.

3. Borning, A., "Defining Constraints Graphically," *CHI86*, ACM, 1986, pp. 137-143.

4. Finzer, W., and L. Gould, "Programming by Rehearsal," *Byte,* Vol. 9, pp. 187-210, 1984.

5. Fischer, G., "Domain-Oriented Design Environments," in *Automated Software Engineering,* Ed., Kluwer Academic Publishers, Boston, MA, 1994, pp. 177-203.

6. Golin, E. J., "Tool Review: Prograph 2.0 from TGS Systems," *Journal of Visual Languages and Computing,* pp. 189-194, 1991.

7. Green, T. R. G., "Cognitive Dimensions of Notations," *Proceedings of the Fifth Conference of the British Computer Society,* Nottingham, Cambridge University Press, 1989, pp. 443-460.

8. Green, T. R. G., M. Petre, and R. K. E. Bellamy, "Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture," *Empirical Studies of Programmers: Fourth Workshop*, Norwood, NJ, Ablex Publishing, 1991, pp. 121-146.

9. Lindholm, T., and F. Yellin, *The Java™ Virtual Machine Specification,* Addison-Wesley, Reading, MA, 1997.

10. Myers, B. A., "The State of the Art in Visual Programming and Program Visualization," *Tech Report,* CMU-CS-88-144, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1988.

11. Nardi, B., *A Small Matter of Programming,* MIT Press, Cambridge, MA, 1993.

12. Papert, S., *The Children's Machine,* Basic Books, New York, 1993.

13. Repenning, A., "Programming Substrates to Create Interactive Learning Environments," *Journal of Interactive Learning Environments, Special Issue on End-User Environments,* Vol. 4, pp. 45-74, 1994.

14. Repenning, A., "Bending the Rules: Steps toward Semantically Enriched Graphical Rewrite Rules," *Proceedings of Visual Languages*, Darmstadt, Germany, IEEE Computer Society, 1995, pp. 226-233.

15. Repenning, A., and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," *Proceedings of the 1996 IEEE Symposium of Visual Languages*, Boulder, CO, Computer Society, 1996, pp. 102-109.

16. Repenning, A., and J. Ambach, "The Agentsheets Behavior Exchange: Supporting Social Behavior Processing," *CHI 97, Conference on Human Factors in Computing Systems, Extended Abstracts*, Atlanta, Georgia, ACM Press, 1997, pp. 26-27.

17. Repenning, A., and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer*, Vol. 28, pp. 17-25, 1995.

18. Shu, N., *Visual Programming,* Van Nostrand Reinhold Company, New York, 1988.

19. Smith, D., "Making Programming Easier for Children," *Interactions,* Vol. III, pp. 59-67, 1996.

20. Smith, D. C., A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM,* Vol. 37, pp. 54-68, 1994.

21. Turkle, S., *Life on Screen,* Simon & Schuster, New York, 1995.

22. Wilde, N., and C. Lewis, "Spreadsheet-based Interactive Graphics: From Prototype to Tool," *Proceedings CHI'90*, Seattle, WA., ACM Press, 1990, pp. 153-159.