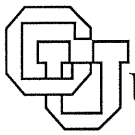
Visual AgenTalk: Anatomy of a Low Threshold, High Ceiling End User Programming Environment

Alexander Repenning & James Ambach

CU-CS-802-96



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Visual AgenTalk: Anatomy of a Low Threshold, High Ceiling End User Programming Environment

Alexander Repenning and James Ambach

Department of Computer Science
Center for LifeLong Learning and Design
University of Colorado, Boulder CO 80309-0430
(303) 492-1349, ralex@cs.colorado.edu
(303) 492-1503, ambach@cs.colorado.edu
Fax: (303) 492-2844
http://www.cs.colorado.edu/~ralex/
http://www.cs.colorado.edu/~ambach/

Abstract

Typical approaches to end user programming involve design trade-offs between ease of use and expressiveness. End user programming environments are either easy to use and not very expressive (low threshold/low ceiling) or more difficult to use but more powerful (high threshold/high ceiling). We propose the development of end user programming environments that are both low threshold and high ceiling by combining a collection of mechanisms that address the issues of program comprehensibility, language tailorability, and interactive multimodality. In this paper, we describe the layered anatomy of a low threshold/high ceiling environment that is usable by both end users and language designers. We then illustrate our theory with a description of a new, low threshold/high ceiling end user programming environment called Visual AgenTalk.

Keywords

graphical rewrite rules, end user programming, scripting, direct manipulation, agents, object-oriented programming, visual programming

1. The End User Programming Challenge

The biggest challenge in creating applications that can be programmed by end users is to find a balance between ease-of-use and expressiveness. The perfect end user programming environment would make it easy for an end user to create simple programs, and at the same time provide capabilities to allow the definition of more complex programs. Different approaches to end user programming tend to fall into one of two categories. Low threshold/low ceiling approaches allow end users to quickly acquire the necessary skills to create simple programs, but as the user becomes a more sophisticated programmer, these environments become inadequate. On the other hand, high threshold/high ceiling programming environments initially require a much greater knowledge of programming, but once these skills are mastered, allow for much more complex programs. Unfortunately many end user programmers simply do not have the time or the interest to acquire these advanced skills.

We have explored, within the context of the Agentsheets programming substrate [18], a number of different approaches to programming that were either low threshold/low ceiling or high threshold/high ceiling. For instance, graphical rewrite rules [1, 6, 9, 10, 16, 21] and AgentBuilder [19], an iconic language, are both examples of easy to use programming approaches that allow end users to create simple animations. In contrast, AgenTalk, a textual, object-oriented programming language based on LISP has, in the hands of more experienced programmers, been used to create much more complex applications within Agentsheets.

These different approaches represent the extreme end points of the programming spectrum. The question is whether it is possible to combine these two approaches in order to define an environment that is at once low threshold and high ceiling. The answer is not to simply add more features to a low threshold/low ceiling language because the added functionality will eventually interfere with the conceptual clarity of low threshold languages. On the other hand simply defining domain-oriented subsets of high threshold/high ceiling languages does not necessarily make the language any easier to learn and will not result in a language useful for end users.

In order to create an end user programming environment that is both low threshold and high ceiling, we believe that the environment must embody mechanisms that promote program *comprehensibility*, language *tailorability*, and interactive *multimodality*.

> The goal of this paper is to describe why comprehensibility, tailorability and multimodality are necessary to create evocative and engaging low threshold/high ceiling environments.

The paper describes an anatomy of programming layers to support these concerns and introduces the Visual AgenTalk environment as special instance of this anatomy.

2. The Anatomy of Low Threshold/High Ceiling Programming Environments

Low threshold/high ceiling implies both ease of use and expressiveness. The idea is to make it easy for end users to program simple things while still providing the capabilities for more complicated programs. Although this may be too great a challenge for a single programming language, we feel that by creating a collection of tightly integrated mechanisms it is possible to create a low threshold/high ceiling environment that is comprehensible, tailorable, and multimodal.

- ***Comprehensibility:*** To lower the threshold of programming, an environment must allow users to easily comprehend how programs and program elements will map to the behavior of application objects.
- ***Tailorability***: Environment must be flexible enough to allow the creation of new, domain specific, programming elements that can be utilized by end users.
- ***Multimodality***: Environments must provide access to all capabilities of the underlying system hardware and software.

2.1. Comprehensibility

The issue of program comprehensibility involves the ability to understand how program primitives and collections of primitives will map to the behavior of a particular application. This problem is exacerbated by the barriers that exist between the things to be programmed and the programs themselves [15]. In typical end user programming environments two worlds exist:

- ***Application World***. The application world contains the objects of direct interest to the end user. Examples of application world objects include things like text, buttons, fields and cards in HyperCard® stacks (Figure 1). Application world objects are the things that are manipulated or viewed by the end user of the application.
- ***Programming World***. The programming world contains objects that describe the behavior of application world objects. In HyperCard, programming world objects consist of a textual programming language and individual scripts that describe how application world objects will behave (Figure 1).

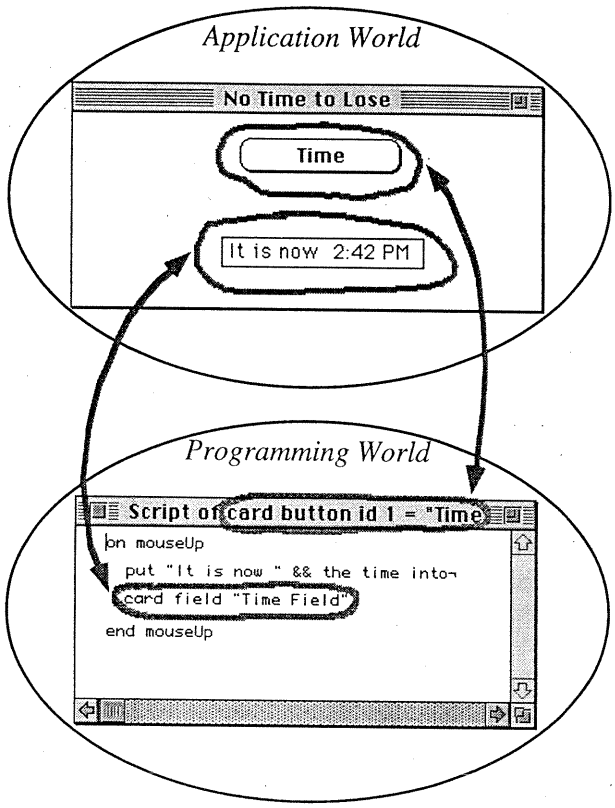


Figure 1: A HyperCard Example

A major difficulty in programming is to comprehend the *mapping* between the application world and the programming world. The issue is to determine what programming world object or collection of objects needs to be employed in order to achieve the desired behavior in the application world. Figure 1 shows a sample script created in HyperTalk and demonstrates the problem of mapping between the two worlds. The programming world is represented by the window entitled "Script of card button id 1 = "Time"". This script specifies the behavior of the Time button that appears in the AW (top of Figure 1). The script dictates that when the Time button is clicked, a message should appear in the Time Field pictured below the Time button. Although the Time Field is an object in the AW that can be dragged, resized and edited, within the programming world, the user refers to it by the abstract name "card field "Time Field"". The user is forced to make the mapping between "card field "Time Field"" in the programming world and the Time Field in the AW. Environments that support comprehensibility must provide links between AW objects and programming world objects.

2.2. Tailorability

A challenge in developing programming languages for end users is to find the right level and kind of programming primitives. A useful repertoire of language primitives

should be closely related to the concepts that end users typically deal with. End users should not be forced to solve frequent problems in their domain by combining a large number of low level, general purpose programming constructs in intricate ways. However, language designers cannot anticipate all the problems that might be tackled with a particular language [13]. Furthermore, the problem domain itself may be a moving target. This presents a difficult design constraint that can only be resolved if the programming language is allowed to evolve over time.

To deal with this problem, it is important that the end user programming language be part of a layered environment that provides facilities for language designers to easily evolve the language. Although these lower levels may not be appropriate for end user programmers, they can be used by language developers to develop new language primitives as the understanding of the problem domain evolves. This tailorability is vital for creating end user languages that do not become obsolete and yet are specific enough to be useful within a particular domain.

2.3. Multimodality

The expressiveness of a programming language determines *what* kind of things end users can verbalize using the computer as a medium. An important aspect of this verbalization is the ability to provide programmers different communication modalities. As computational hardware and software become more powerful, they allow for more sophisticated means of interaction. Computer users do not interact with their applications just by typing in commands anymore, instead they use a combination of typing, mouse gestures and even voice. In response, computers can communicate with users via text, still images, sound, speech and animation. These different communication mechanisms are ubiquitous in modern computer applications, and end user programming environments should not limit the programmer to particular modalities.

The ability to define *when* a program should execute can be as complex as defining what the program should do. It is often difficult to specify the circumstances that describe when certain activities should be executed. A classic example of *when* complexity is the VCR. Most VCR owners are capable of playing a tape and even recording a program directly by pressing either the play or record button. However, the intricacies of "programmed" recording are due to establishing *when* the event should occur. Thus, setting the right start time, stop time, and channel become the real source of programming frustration. End user programming environments need to provide access to different communication modalities (mouse clicks, timers, speech input, etc.) in order to let end users specify exactly when different programs should be invoked. These capabilities provide for more sophisticated, expressive programs.

2.4. An Anatomy of Layers

An end user programming environment that adequately addresses the issues of comprehensibility, tailorability and multimodality will provide both a low threshold and a high ceiling at the same time. An architecture supporting this kind of environment is pictured in Figure 2. Application world objects and programming world objects are available from within an end user programming substrate that can be tailored for a particular domain, and the end user programming substrate is defined from a rich programming substrate useful for language designers.

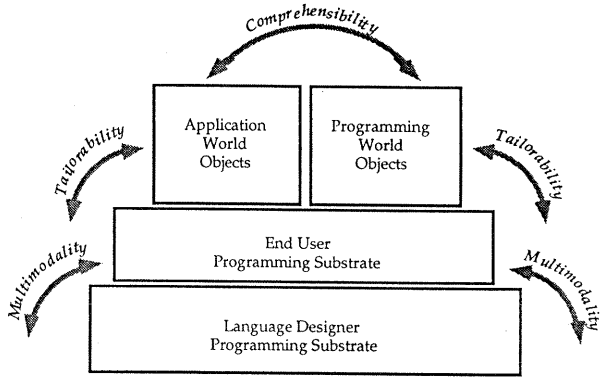


Figure 2: A Low Threshold/High Ceiling Anatomy

Comprehensibility is facilitated through the easy interchange of objects from the application world and the programming world. This is possible if both types of objects support user control via direct manipulation [20], and the objects are allowed to interact with each other. Through this interaction, end users experience the mapping between the two worlds, helping them to learn about programming primitives and to test and debug their programs. Tailorability and multimodality are provided through the integration of lower levels and upper levels, and are supported by a low level programming substrate that provides language designers with a rich set of functionality that they can package into programming world objects for the end user. Visual AgenTalk is an end user programming environment that instantiates the anatomy described above.

3. Visual AgenTalk at a Glance

Visual AgenTalk is a new programming mechanism for the Agentsheets [18] programming substrate used to develop simulations and design environments such as the CityTraffic environment (Figure 4). Applications developed in Agentsheets typically consist of a large number of autonomous, communicating agents organized in a grid called the *agentsheet*. Agents can communicate amongst

themselves via the grid, or with the user through different modalities including animation, sound and speech. Users can communicate with agents through the use of direct manipulation tools, the keyboard or even voice.

3.1. Application World Objects: Agents

Agentsheets and Visual AgenTalk are combined into a layered architecture (Figure 3) that allows language designers to create tailored programming environments for end users.

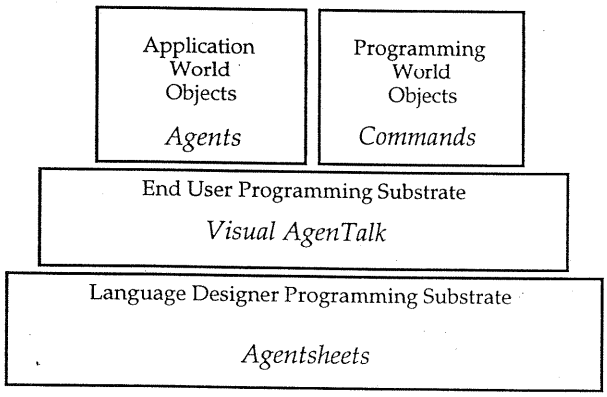


Figure 3: Layered Language Architecture

End-users of Agentsheets applications create simulations and designs by selecting application world objects called *agents* from a Gallery (Figure 4 (1)) and placing them into a worksheet (Figure 4 (2)). Once a worksheet is populated, users can start simulations by selecting a menu item. Starting a simulation activates agent behavior, and users can continue to interact with the agents while the simulation is running [17]. The behavior of application world objects can be altered with programming world objects. In Figure 4, the end user is designing a traffic simulation. Roads, cars, tracks, trains and stoppers are all agents, and when the user starts the simulation, the agents will act according to their programmed behavior: cars will move on roads, and trains will move on tracks.

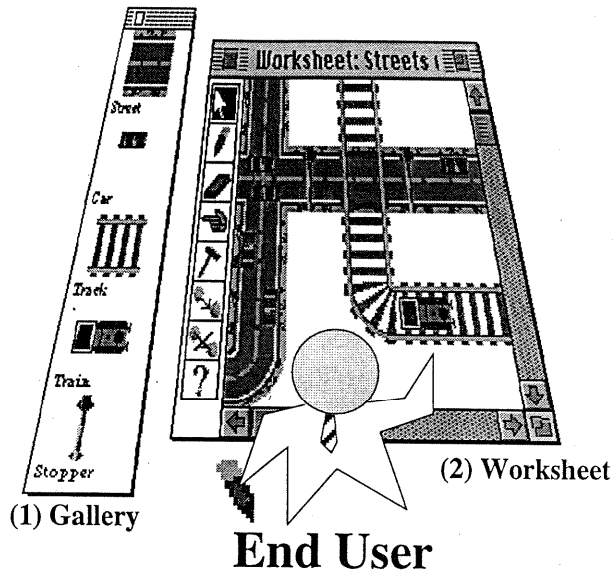


Figure 4: The City Traffic Simulation

Designers create application world and programming world objects for environments such as the City Traffic simulation (Figure 4) by collaborating with end users [18]. Agentsheets provides tools that allow language designers to create both the look and behavior of agents. In the case of the City Traffic simulation, designers have to design and implement the behavior of application world objects such as cars and trains as well as to provide a basic set of programming world objects to allow end user programming.

3.2. Programming World Objects: Commands

Programming world objects in Visual AgenTalk are called *commands*. Commands are small interactive forms representing programming primitives that can be manipulated by end users. Commands can be both general purpose as well as domain specific, and have interfaces that consist of familiar direct manipulation widgets.

A command (Figure 5) consists of a name and an arbitrary number of parameters. End users set the values of the typed parameters via *type interactors* such as number fields, text fields, check boxes, textual and iconic pop up menus. type interactors can limit user input to valid choices. For instance, a sound type interactor is a pop up menu offering only the names of sounds that are available in the system.

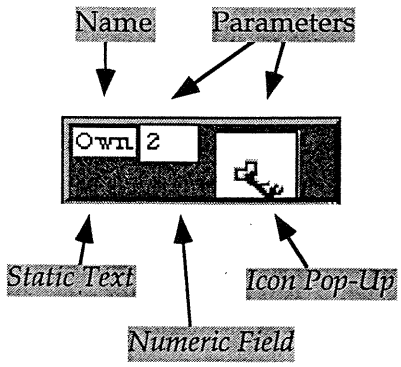


Figure 5: A Visual AgenTalk Command

Parameters, such as the key in Figure 5, can be references to application world objects. The ability to have application world objects appear in the programming world as they do in the application world significantly helps end users to map between the two worlds [8].

The following sections indicate how end users learn about the functionality of programming world objects by making them interact with application world objects, illustrate how end users create programs by composing programming world objects, describe how language designers create new programming world objects for end users, and explain how Visual AgenTalk supports end users in creating multimodal applications.

4. Comprehensibility: Connecting the Worlds

Elevating programming world objects onto the level of a direct manipulation drag and drop user interface, turns them into *tactile objects* that are no longer confined to individual windows which, in traditional programming interfaces, are used as rigid barriers between the application world and the programming world.

4.1. Breaking down the Barriers

Visual AgenTalk supports the direct interaction between application world and programming world objects through a flexible drag and drop mechanism. Along the lines of dynamic programming languages such as Lisp and Smalltalk, any command or command composite can be executed at any point in time. A command is applied to an agent by selecting the command from a palette, specifying the command's parameters, and dragging and dropping the command onto the agent in the worksheet. Figure 6 shows a simple Turing machine implemented in Agentsheets. The

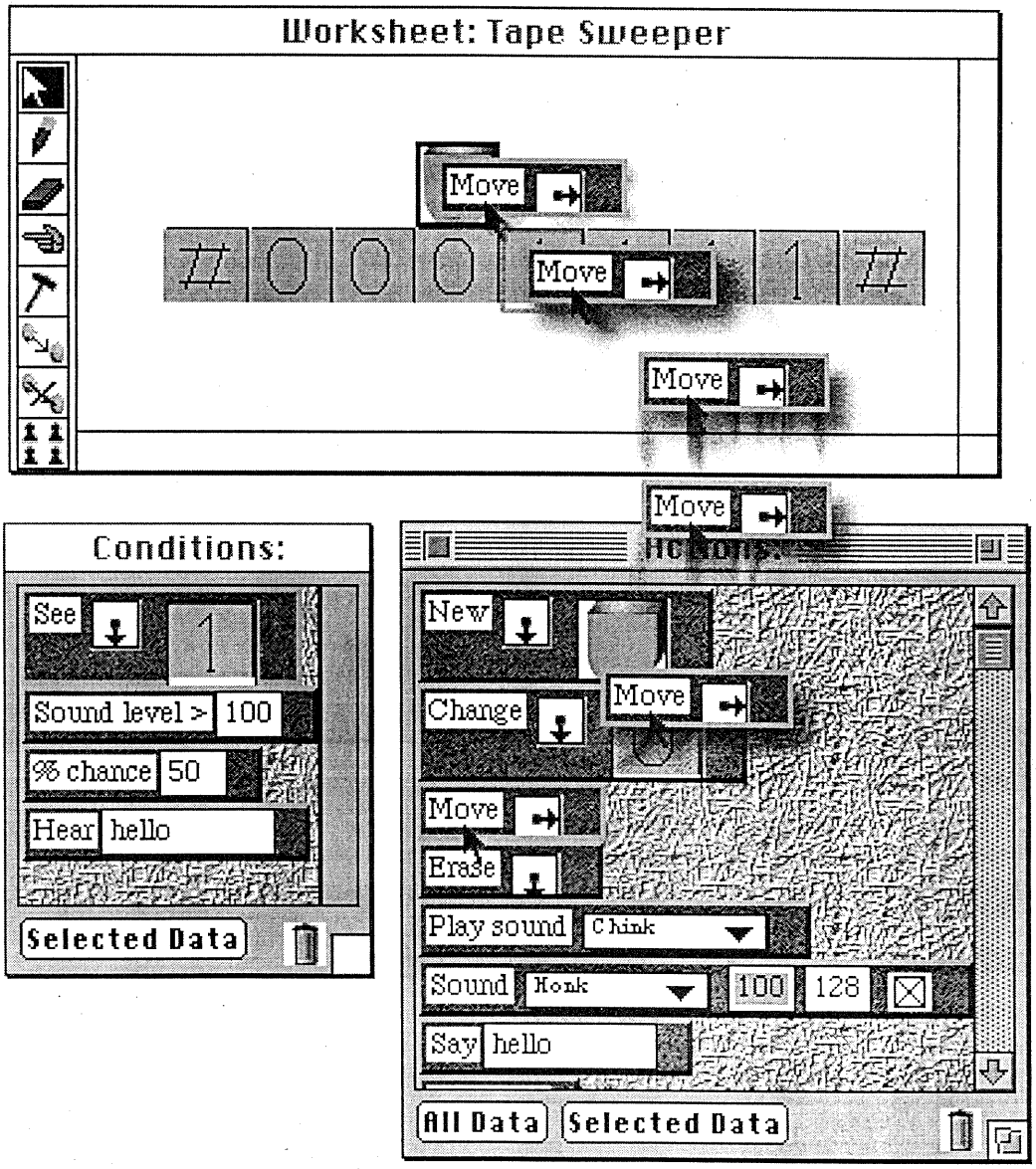


Figure 6: Commands can be dragged and dropped onto agents to explore their function and to modify agents

top window is the worksheet representing the application world containing a number of agents representing the tape pieces and the Turing machine head. Below the worksheet are two command palettes representing part of the programming world. The condition command palette, on the left, holds commands to query the state of the application world, and the action command palette, on the right, holds commands that can change the application world. In the figure, the "move" action command was selected, its direction parameter was set to ➡ (right) via a graphical pop up menu (Figure 12), and the command was then dragged on top of the Turing machine head. This powerful application and execution paradigm allows the end user to explore and test the repertoire of commands and parameters in the *context* of any agent.

Dropping the command on the Turing head results in the command's execution. The Turing head will move according to the direction indicated in the command, in this case one position to the right (Figure 7).

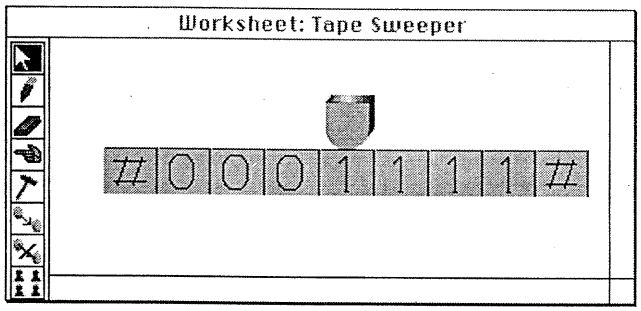


Figure 7: The Turing machine head after moving one position to right.

This tactile nature of programming world objects, enabled by the drag and drop interaction mechanism [22], extends the notion of object-orientation. Tactility encourages a more exploratory style of programming in which end users perceive the functionality of programming world objects by "touching" them in a direct manipulation sense. As we will see later this tactility has positive consequences for other aspects of programming as well.

4.2. Composing Programming World Objects

The notion of programming captures a range of activities including the exploration of programming world objects to asses their suitability for solving specific problems, and the process of *composing* programming world objects into programs. Visual AgenTalk supports rule-based programming treating rules as special kinds of commands.

Rules are edited through rule editors (Figure 8). Similar to the notion of nested boxes in Boxer [5], Visual AgenTalk makes use of containment to indicate part-whole relationships between commands. The IF part, on the left, and THEN part, on the right, of each rule are represented by flexible size containers into which commands can be dragged and dropped. The IF part is an implicit conjunction of all the conditions and the THEN part is an implicit action sequence. Rule sets are interpreted from top to bottom. If the rule interpreter finds a rule for which all the conditions are true then it will execute all of its actions, again from top to bottom, and return from one matching cycle.

The agent representing the Turing machine head gets programmed with two rules (Figure 9) capturing the functionality specified in this table:

If	Then
there is a "0" below	move right
there is a "1" below	change it to "0" and move right

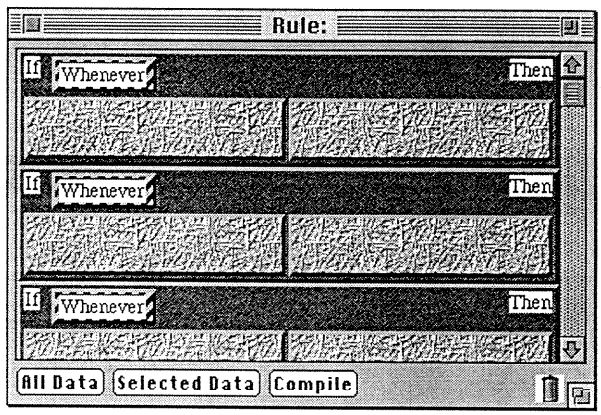


Figure 8: Empty Rule Editor.

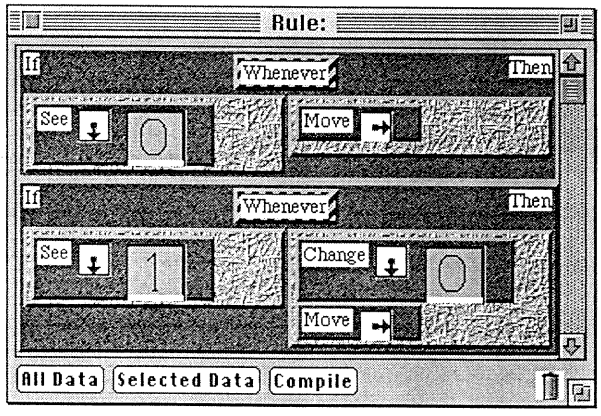


Figure 9: Turing machine rules.

Tactility in Visual AgenTalk does not only permit the composition of programs, in the form of rules, but also supports the comprehension of these programs. Problems identified regarding rule comprehension [7] and debugging [2, 12, 14] stand in contrast with claims of modularity of

rule-based approaches [3]. We believe that this problem is at least partly due to the lack of the ability to test individual rules. In Visual AgenTalk each part of the rule, the IF part, the THEN part, individual commands, and even entire rules can be dragged and dropped onto agents to test the rule. Feedback is then provided indicating whether the rule would match and what will happen if the actions get executed, or, if the rule does not match, feedback is provided explaining why.

5. Tailorability: Expanding the Worlds

The layered architecture of Agentsheets and Visual AgenTalk (Figure 3) helps language designers tailor programming and application worlds so that they feature objects that are directly relevant to end users. Designers create new commands in Visual AgenTalk by specifying their interface and by implementing their functionality in the textual AgenTalk language. AgenTalk is an extension of lisp and is part of the language designer programming substrate. Designers create new agents by subclassing existing agent classes and by defining their look. Visual AgenTalk offers *embedded fallback* and *command generators* as mechanisms to bridge the syntactic, semantic and interactive gaps between layers.

5.1. Embedded Fallback

The most primitive interaction mechanism that provides lower-level substrate access to end users is called embedded fallback. Embedded fallback allows knowledgeable end users to escape into the language designer programming substrate layer from the end user programming substrate layer. Visual AgenTalk provides a command that allows users to evaluate any valid Lisp expression (with or without side effects) within a Visual AgenTalk program. For instance, the command 

will compute the sin of the number 1.5. Since this command is part of Visual AgenTalk, it can be dragged and dropped just like any other Visual AgenTalk command in order to test it or to make it part of a rule.

5.2. Command Generators

Command Generators are used by language designers to create new language primitives for end users. Embedded fallback mechanisms, such as the "Lisp" command in Visual AgenTalk, are of limited use to most end users because they require a substantial knowledge of the lower level programming language. Command generators, in contrast, allow language designers to create new commands for end users who have no knowledge of the language designer programming substrate.

Command generators take compact specifications by the language designer and generate new programming world objects. In doing so, command generators bridge the gap between the language designer programming substrate and the programming world. This is not only a process of abstraction but at the same time a mechanism to generate concrete user interface snippets. An example will illustrate the generation process.

WebQuest is a game design environment implemented in Agentsheets, used by school children . WebQuest features a number of characters such as knights and princesses, that are either controlled manually or can be programmed.

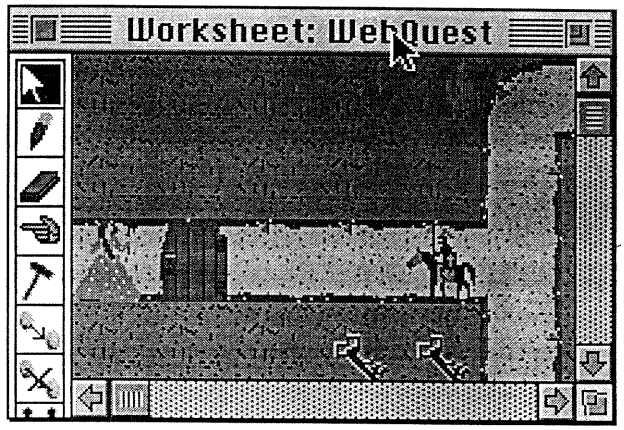


Figure 10: WebQuest Scene

A typical game playing activity is the acquisition of objects, such as keys, that allow characters to overcome obstacles. In the scene in Figure 10 keys are required to open the gate. To express these kinds of programs, new commands must be added to Visual AgenTalk. A "Take" action command takes an object and adds it to a list of objects owned. An "Own" condition command allows users to express rules that depend on the ownership of certain objects.

The language designer specifies a new command in terms of its parameters and language designer programming substrate code (Figure 11). The grayed out code is only provided to illustrate the small amount of code required to define new commands.

```
(defcommand TAKE ((Direction direction-type))
  :macro '(progn
            (push (effect ,Direction 'depiction) Possessions)
            (effect ,Direction 'erase)))
```

Figure 11: The Take Action Command Definition.

This declarative specification automatically generates the 

command. The Direction parameter is of type direction-type. The command generation uses the command declaration to create a new drag and droppable user interface snippet by automatically instantiating and laying out type interactors such as the graphical direction pop up menu (Figure 12).

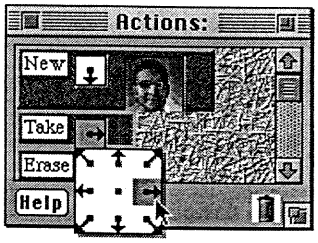


Figure 12.: A Pop up menu to interact with direction types.

Immediately the new command can be utilized by the end user either by dragging it onto an agent, or by dragging it into a rule (Figure 13).

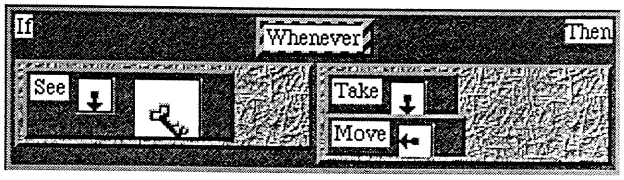


Figure 13: If the condition in this rule is satisfied, the agent will take the key and move left

Applying this rule, the knight takes two keys and moves in front of the gate (Figure 14).

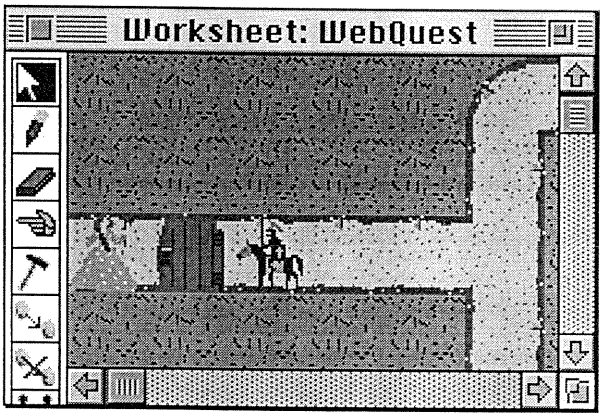
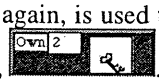


Figure 14: WebQuest scene with knight advancing towards princess.

The next step for the language designer is to define the "Own" condition command to test for possessions. The command definition (Figure 15) contains two parameters. The first one, How-Many is of type number-type and describes how many objects of a certain kind need to be owned. The second parameter, What, points to an application world object represented by a graphical pop-up menu containing the objects that can be owned.

```
(defcommand OWN ((How-Many number-type) (What depiction-type))
  (macro '(>= (count :What Possessions) :How-Many)))
```

Figure 15: The Own Condition Command Definition.

The definition, again, is used to automatically generate the Own command, 

.

The end user now composes a new rule containing an "Own" condition (Figure 16). When the rules gets executed, because there is a door to the left, and the knight has two keys he says "I'm in" and moves to the left.

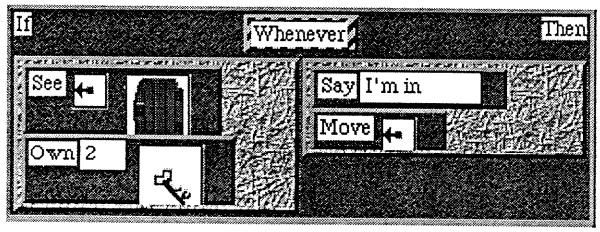


Figure 16: Rule to open door with at least two keys.

6. Multimodality: Words User Communication

End user programmable applications such as simulations, games and visual programming environments can be made more evocative and engaging by providing users multimodal paradigms to interact with the application world. By this we do not necessarily mean multimodal input and output devices, such as data gloves, that are often used in full immersion applications. Even *off-the-shelve multimodality* as it is supported by current multimedia computers can significantly enrich the human-computer communication. Adding speech input, sound recording, speech output, and sound playing to mouse, keyboard input and screen output can add new dimensions of interactivity to end user environments. However, the integration of multimodality poses additional challenges:

- *What* are the modalities offered in a programming substrate and how can they be packaged up into a useful and usable programming interface for end users? Visual AgentTalk includes a number of programming world objects to provide end users access to the multimodal functionality provided by the Agentsheets programming substrate.
- *When* are these programming world objects to be executed? Multimodal interaction introduces new timing and synchronization problems. Visual AgenTalk uses the concept of *triggers* to provide users control over program execution.

The idea is to move from Single modality In Single modality Out, *SISO*, to Multimodal In Multimodal Out, *MIMO*, models. The use of single modalities such as the visual representations used in graphical rewrite rules may lead to end user programming environments that are very powerful in the realm of this single modality but it may be hard to incrementally add new ones. Vampire [11] adds simple mouse clicks to rewrite rules, Science Sheets and

KidSim [21] add the ability to play sounds to rewrite rules. Visual AgenTalk provides a flexible architecture to introduce new modalities when they become available by defining new triggers, conditions and actions. Currently Visual AgenTalk supports the following interactions:

Output	
Pictures:	The look, color and position of agents can be controlled.
Sound:	Sounds can be played at different pitches and amplitudes.
Speech:	The voice synthesizer can speak arbitrary strings in a number of voices.
Input	
Sound:	The sound level picked up by the microphone can be queried Sounds can be recorded
Keyboard:	Individual keys and key chords can be tested for.
Mouse:	Mouse clicks with modifier keys can be tested for.
Speech:	Speech produced by the voice synthesizer can be matched.
Time:	Rules can execute at time intervals specified by user

The following sections describe in the context of examples how end users can employ these modalities.

6.1. Creating a Clapper: Using triggers and conditions

The combination of triggers and conditions allows end users to specify complex circumstances under which rules execute. The Agentsheets Electric World application serves as example.

The Agentsheets Electric World (Figure 17) contains switches that turn electricity on or off. How should a user interact with the simulation to turn individual switches on and off? A plausible interaction mode for this kind of activity would be to click the switch with the mouse. This mode would be hard to implement in graphical rewrite rules since they do not typically provide a good way to specify mouse input.

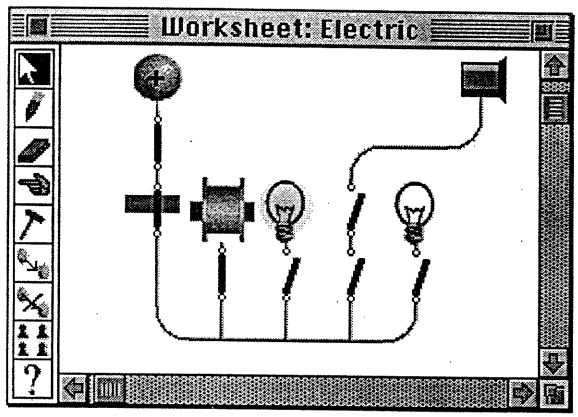


Figure 17: Electric World with Switches.

Triggers allow end users to specify the execution of rules based on events. Two rules with click triggers, 

, are used to toggle between the on state and the off state (Figure 18). The check boxes included in the click trigger are used to further qualify the trigger with click modifiers keys (shift, control, option, and command key). Depending on what state the switch is currently in it either switches to the on or off position with an appropriate sound.

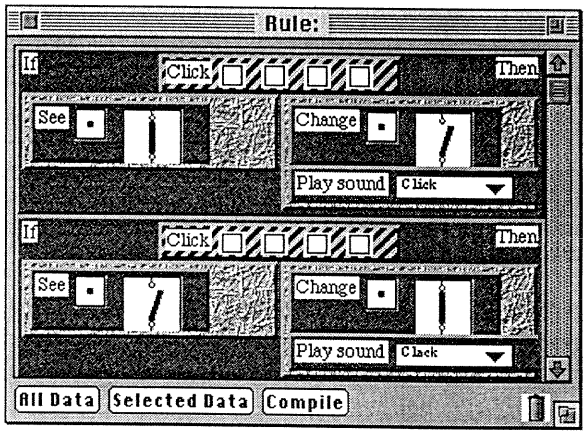
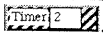


Figure 18: Mouse click triggered rules to operate switch.

An autonomous switch, such as a blinker relay used for a car turn signal, can be implemented by having the switch automatically change its state from on to off every fixed number of seconds. This is achieved simply by using a timer trigger type, 

. To be able to distinguish the appearance of the timer switch from the ordinary switch it is marked with a "T".

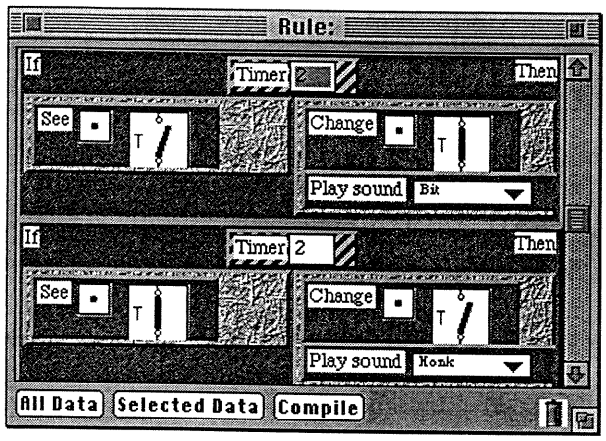


Figure 19: Implementation of Timer Switch with Timer Trigger.

Conditions can be used to allow different communication modalities as well. For instance, to implement a clap on, clap off type of switch, the sound level condition, Sound level > 200, can be used. In the rule (Figure 20), the sound level condition is used to determine if the sound level picked up by the computer's microphone is larger than 200 (on a scale of 0 - 255). Again, to distinguish the clap activated switch it is marked with a "C".

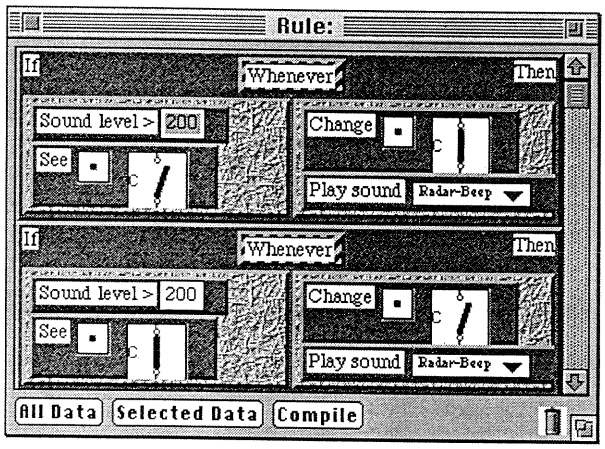


Figure 20: Clap on, Clap off Switch Implementation.

Finally, we can replace some of the ordinary switches from Figure 17 with timer switches and clappers.

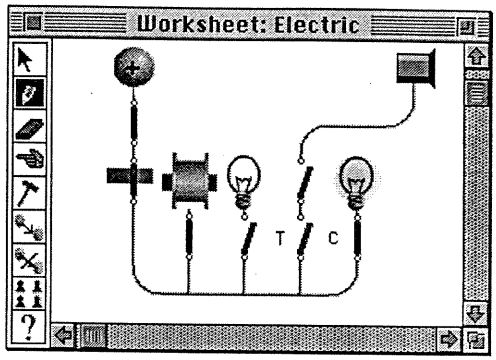


Figure 21: Mouse click, Clapping, and Time Controlled Circuit.

The circuit is now controlled by the user clapping his hands, clicking the switches, as well as by time.

6.2 Talking Agents

Speech input and output can be used to program complex interactions between application world objects. Figure 22 shows 3 characters called, from top to bottom, Hans, Bob and Franz. The three characters live in a maze consisting of walls and a few dollar bills. The characters can speak. When they speak, a simple animation of expanding circles indicates who speaks and also indicates the scope in which other characters can hear them.

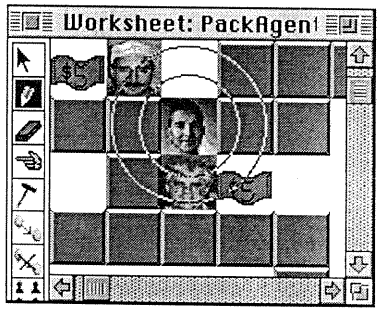


Figure 22: Talking Agents.

All three characters share the same rule set. The "Hear" condition is used to match any fragments of sentences spoken. If a character hears the words "right" and "money" and sees a dollar bill to the left, then it will speak its name. A second rule tests for the presence of the words "left", the word "money" and a dollar bill to the right.

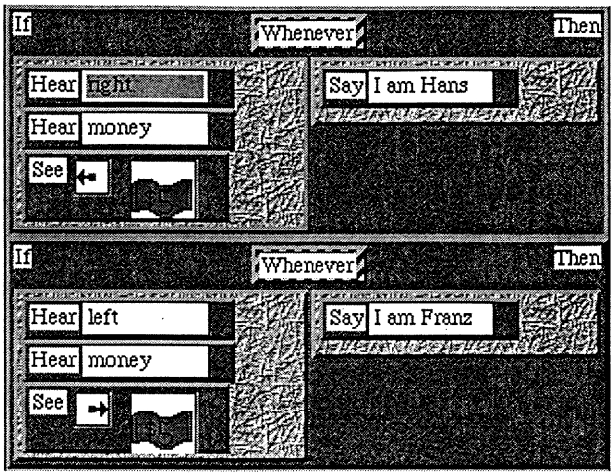
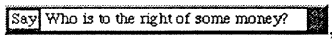


Figure 23: Rules for Listeners

When a "say" command, 

, is dragged and dropped onto Bob he will broadcast the question (Figure 22) to all of his immediate neighbors. The neighbors parse the sentence according to the rule and reply if the conditions match. In this case only Hans replies since he is to the right of some money.

This example illustrates how modalities can be combined to implement complex communications. Employing these different modalities within a language greatly improves the expressiveness of that language. Visual AgenTalk provides a flexible architecture that can not only take advantage of new communication modalities but through it's structure of conditions, actions, triggers and rules, Visual AgenTalk can provide end users with the ability to take advantage of these modalities in ways that is both easy to use and to test.

7. Conclusion and Future Work

Although most end user programming environments are either easy to use, but not very expressive (low threshold/low ceiling), or more difficult to use, but more expressive (high threshold/high ceiling), it is possible to define environments that are both low threshold and high ceiling. The anatomy of such an environment needs to adequately address the issues of comprehensibility, tailorability and multimodality. This is achieved by combining a multi-layered software architecture with different mechanisms that allow end users to easily combine programming world and application world objects, and language designers to smoothly integrate functionality from lower level programming substrates to higher level, end user substrates. Visual AgenTalk is an example of an end user programming environment that embodies this anatomy.

Visual AgenTalk effectively reduces the barrier between the programming world and the application world by allowing all objects to appear and interact in both worlds. This increases the comprehensibility of programs and program primitives. Visual AgenTalk's tailorability mechanisms, including embedded fallback and command generators, provide facilities for end users that need more sophisticated, general purpose primitives, as well as for language designers who need to evolve the end user language. Furthermore, Visual AgenTalk's repertoire of commands, triggers and rules allows end user to create sophisticated and engaging applications that are starting to blur the boundaries between end user programming and multi media authoring environments.

Although the current instantiation of Visual AgenTalk addresses the concerns of comprehensibility, tailorability and multimodality, we believe that steps can be taken to lower the threshold and raise the ceiling of the programming environment even further. For instance, graphical rewrite rules, such as the ones used in Agentsheets and other systems, have proven successful at lowering programming thresholds, and we would like to create new Visual AgenTalk commands that allow the inclusion of graphical rewrite rules. Furthermore, the self disclosure of low level programming primitives [4] is an interesting learning mechanism that we would like to incorporate to assist end user programmers interested in exploring the more powerful Agentsheets layer. Both of these approaches can be implemented within the anatomy described above.

Acknowledgments

We wish to thank Gerhard Fischer and the Center for LifeLong Learning and Design for all of the thoughtful discussions. We also want to thank Corrina Perrone and David Clark for their excellent work, and Kurt Schneider, Kumiyo Nakakoji and Tamara Sumner for their feedback. This work has been supported by the Advanced Research Projects Agency under Cooperative Agreement Number CDA-940860, and the National Science Foundation under grant number RED925-3425.

References

1. Bell, B. and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," *1993 IEEE Workshop on Visual Languages*, Bergen, Norway, 1993, pp. 188-195.

2. Carver, S. M. and S. C. Risinger, "Improving Children's Debugging Skills," in *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard and E. Soloway, Ed., Ablex Publishing Corporation, Norwood, New Jersey, 1987, pp. 147-171.

3. Cooper, T. and N. Wogrion, *Rule-based Programming with OPS5,* Morgan Kaufman Publischers, Inc., San Mateo, CA, 1988.

4. DiGiano, C. and M. Eisenberg, "Self-disclosing Design Tools: A Gentle Introduction to End-User Programming," *Designing Interactive Systems*, Ann Arbor, Michigan USA, 1995, pp. 189-197.

5. diSessa, A. A., "An Overview of Boxer," *Journal of Mathematical Behavior,* pp. 3-15, 1991.

6. Furnas, G. W., "New Graphical Reasoning Models for Understanding Graphical Interfaces," *Proceedings CHI'91*, New Orleans, LA, 1991, pp. 71-78.

7. Gilmore, D., K. Pheasey, J. Underwood and G. Underwood, "Learning graphical programming: An evaluation of KidSim," *Proceedings of the Fifth IFIP Conference on Human-Computer Interaction*, London, 1995, pp. .

8. Halbert, D. C., "SmallStar: Programming by Demonstration in the Desktop Metaphor," in *Watch What I Do: Programming by Demonstration*, A. Cypher, Ed., The MIT Press, Cambridge, MA, 1993, pp. 103-124.

9. Kirsch, R., A., "Computer Interpretation of English and Text and Picture Patterns," *IEEE Transactions on Electronic Computers,* Vol. 13, pp. 363-376, 1964.

10. Lieberman, H., "An Example-Based Environment for Beginning Programmers," in *Artificial Intelligence and Education*, R. W. Lawler and M. Yazdani, Ed., Ablex Publishing, Norwood, NJ, 1987, pp. 135-151.

11. McIntyre, D., "Design and Implementation of Vampire," in *Visual Object-Oriented Programming*, M. Burnett, A. Goldberg and T. Lewis, Ed., Manning Publications Co., Greenwich, 1995, pp. 129-159.

12. Nanja, M. and C. R. Cook, "An Analysis of the On-Line Debugging Process," in *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard and E. Soloway, Ed., Ablex Publishing Corporation, Norwood, New Jersey, 1987, pp. 172-184.

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

14. Papert, S., *Mindstorms: Children, Computers and Powerful Ideas,* Basic Books, New York, 1980.

15. Pennington, N., "Comprehension Strategies in Programming," in *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard and E. Soloway, Ed., Ablex Publishing Corporation, Norwood, New Jersey, 1987, pp. 100-113.

16. Repenning, A., "Bending the Rules: Steps toward Semantically enriched Graphical Rewrite Rules," *Proceeding of Visual Languages*, Darmstadt, Germany, 1995, pp. 226-233.

17. Repenning, A. and T. Sumner, "Programming as Problem Solving: A Participatory Theater Approach," *Workshop on Advanced Visual Interfaces '94*, Bari, Italy, 1994, pp. 182-191.

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

19. Robinson, R., D. Cook and S. Tanimoto, "Programming Agents with Visual Rules," *Proceeding of Visual Languages*, Darmstadt, Germany, 1995, pp. .

20. Shneiderman, B., "Direct Manipulation: A Step Beyond Programming Languages," in *Human-Computer Interaction: A multidisciplinary approach*, R. M. Baecker and W. A. S. Buxton, Ed., Morgan Kaufmann Publishers, INC. 95 First Street, Los Altos, CA 94022, Toronto, 1989, pp. 461-467.

21. 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.

22. Wagner, A., P. Curran and R. OBrien, "Drag Me, Drop Me, Treat Me Like an Object," *CHI '95*, Denver, CO, 1995, pp. 525-530.