Authoring Interactive Behaviors for Multimedia

Brad A. Myers

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

Presented at the 9th NEC Research Symposium
August 20-September 1, 1998
Nara, Japan

Abstract

The tools for authoring multimedia presentations start with sophisticated interactive tools like Director and ToolBook. However, to make the presentations truly interactive requires programming in "scripting languages." These languages have generally been difficult to learn for non-programmers. "Interactive behaviors" allow users to click on, move, or otherwise interact with objects on the screen, as opposed to just watching the presentation like a TV show. Behaviors range from simply clicking on buttons or links, to sophisticated interactions with computerized characters. This paper presents a variety of ways we are studying to make authoring of these interactive behaviors more accessible to non-programmers. One approach is "demonstrational" techniques, where the author gives examples of the desired actions and results, and the system generates the code to perform the same actions at run time. Using demonstrational techniques has proven successful for specifying simple behaviors. To represent the behaviors and allow the author to edit them, we are investigating new languages which are designed to be more "natural" because they are based on how non-programmers actually think about these tasks. Human-factors studies have been performed to investigate how people naturally express algorithms. These studies have revealed some general principles which can be applied to the design of new languages, such as that a general case is often expressed first, with exceptions afterwards, and that loops are avoided by applying operations to sets of objects. Using these new results, along with results from the fields of Empirical Studies of Programmers and Human-Computer Interaction, we can create languages that are easier to learn and more effective to use. This will enable a wider range of people to read, generate and modify the code.

Introduction

We are working to develop new methods, techniques and tools that will make it significantly easier for people to have the capabilities of programming. We are particularly targeting applications where people who are not professional programmers are expected to write the programs. One of the applications for this is to *author interactive behaviors for multimedia*. But what do these terms mean?

Definitions

Authoring

"Authoring" is creating the content for any kind of presentation or document. This ranges from what everyone does when they write text, including authoring a letter or a novel. This might use paper or a direct manipulation click and edit tool like Microsoft Word. For authoring World-Wide-Web pages, tools for creating the text include WYSIWYG editors like Adobe PageMill and Microsoft's FrontPage. Authoring for pictures uses tools like paper, or drawing programs like Adobe Illustrator or MacDraw. To create more sophisticated behaviors, the author might need to use scripting languages like the "Lingo" language used in Macromedia's Director, or the author might program in a professional programming language like C++.

The term "authoring" is being used here to cover a wide range of kinds of creating, from just typing text, to drawing, all the way to writing programs which control the content. Authoring also covers an enormous range of expertise.

Interactive Behaviors

Some multimedia projects are just designed to be passively watched, like a television show. However, I am particularly interested in authoring of *Interactive Behaviors*, which is when the user can click on or move an object on the screen. In multimedia presentations that include interactive behaviors, the user must *get actively involved*. This is the hardest kind of multimedia production to create, and also the most engaging for the user. Examples of multimedia with interactive behaviors include all games, most educational software, and web pages where the user clicks on links or fills in fields.

Multimedia

Multimedia is the use of text, graphics, video, photographs, audio, and animations together on a computer. I am using this very broad definition to include many different kinds of systems and situations.

Why is Authoring Important?

We believe that there will be increased demand and opportunity for universal authoring of multimedia. About 89% of households in the United States have a still camera, 80% have a tape recorder, and about 38% have a camcorder.[1] All of these technologies are becoming widely available in digital form (digital cameras are now comparably priced to conventional cameras, and digital video cameras are appearing). When people create their scrapbooks and home movies in the near future, they might be creating digital multimedia productions. While there may be

[1] These statistics are taken from my "Computer Almanac: Numbers about Computers." See http://www.cs.cmu.edu/~bam/numbers/. Unfortunately, I do not have the corresponding numbers for Japan.

specialized viewing and editing hardware, there is also the opportunity to more effectively allow people to use their general-purpose computers to author multimedia productions.

Access and authoring for the world-wide-web is increasing exponentially. *Universal authoring* has been one of the basic tenets of the world-wide-web from the beginning. One of the chief reasons that the WWW started to become popular was that it allows everyone to author. HTML was simple enough that tens of thousands of people created web pages in the first few years of the WWW. Now, there are many interactive programs like Microsoft FrontPage and Adobe PageMill that make it even easier to author, at least for static text and pictures. The digital cameras mentioned above will make it easy for people to include still pictures and video into their WWW pages. However, it remains difficult to make the pages interactive with more than just clicking, as is needed to support forms, draggable items, or any kind of game. In this case, programming in Java, Perl, Visual Basic, or some other programming language is often required.

Another popular use for multimedia is in educational software. An article from a recent issue of the *Communications of the ACM* claims says that “Digital multimedia offers the key to educational reform” [6]. Certainly, just as there are text books created by experts and produced by professional publishing houses, there will be multimedia publications created by large teams of experts. However, teachers must prepare lessons that are individualized to each classroom and even to each student, and they do this today by picking and choosing from their text books, and by creating a lot of their own material and handouts. If teachers are to use multimedia effectively, then the teachers *themselves* will similarly have to be able to refine existing material and create their own multimedia material, so that it will be geared to their particular needs and their particular classes. Furthermore, educational multimedia is often interactive, with small games, quizzes, and user-selectable options. Therefore, teachers are going to have to be authors of interactive multimedia material, but they do not have the resources to spend a lot of time learning how.

Another reason for supporting *general authoring* is that people learn better when they *construct* things than when they just use them [33]. Therefore, providing an environment where people can *create* many interesting kinds of multimedia applications will enhance learning. This is the topic of a recent issue of *Communications of the ACM* [11], and has been promoted by many researchers from Papert [26] to Soloway [32].

Why are *Interactive Behaviors* Important?

The importance for *interactivity* in multimedia was highlighted by a recent New York Times article that summarized the conclusions of the 1997 Roundtable in Multimedia [1]. It concluded that “customers are not interested” in many of today’s multimedia products, so “money is not being made.” This is because it is not sufficient just to have good material: “content is most certainly not king. Interaction matters an order of magnitude more than content.” Furthermore, “people liked interactive media best when they were able to create something, or participate in the process of creation” [1]. Thus, it is not sufficient just to present the content in a static way. People want to provide interesting and sophisticated experiences for the viewer by allowing the viewer to interact with their productions.

Roger Schank calls for more interactivity in educational multimedia:

> “Creating educationally effective multimedia programs means taking seriously the idea of learning by doing. Good educational software is active, not passive, and ensures that users are doing, not simply watching.” [29, p. 69]

Goal: More People Able to Author Interactive Behaviors

One important issue is the *range of people* who can be authors. Who are the authors going to be? Is it just professional programmers, or can everyone create multimedia productions that include interaction? How much extra training does a design major need in order to be able to create multimedia that interacts with the user? An important theme of this paper is that authoring of multimedia should be accessible to everyone. This should even include allowing everyone to be able to author interesting interactive behaviors, not just have this be limited to professional programmers.

This comes under the category of “End User Programming,” since we are aiming to allow end users to write programs.

Approach

To address these problems, we are working to decrease the difficulty of programming. One approach is to eliminate the appearance of programming where possible. However, users still need the *capabilities* of programming, as we have argued above, such as conditionals, iterations, etc. Therefore, we use direct manipulation and “demonstrational” techniques, where users can give examples of how they want the interface to work. Our second approach is to try to make the programming languages themselves easier to learn by being more “natural.”

Why study programming languages at all today? More than a decade ago, Allen Newell and Stu Card pointed out:

> “Millions for compilers but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use. ... The human and computer parts of programming languages have developed in radical asymmetry.” [22, p 212-3]

This situation still holds. Somewhat surprisingly, it even applies to multimedia scripting languages, which are programming languages designed for use by people who are not professional programmers, and therefore are most likely to benefit from research on the human side. There are surprising gaps in the knowledge about how to make programming languages effective for people, and we are working to fill those in.

Gentle Slope Systems

Our research is closely aligned with the concept of “Gentle Slope Systems” [4] [21] which are systems where for each incremental increase in the level of customizability, the user only needs to learn an incremental amount. This is contrasted with most systems which have “walls” where the user must stop and learn many new concepts and techniques to make further progress (see Figure 1). We use direct manipulation and demonstrational techniques to lower the initial starting point (so users can get useful work done immediately), and we are creating a language that is easy to learn so the number and height of the walls is minimized, if they cannot be eliminated entirely.

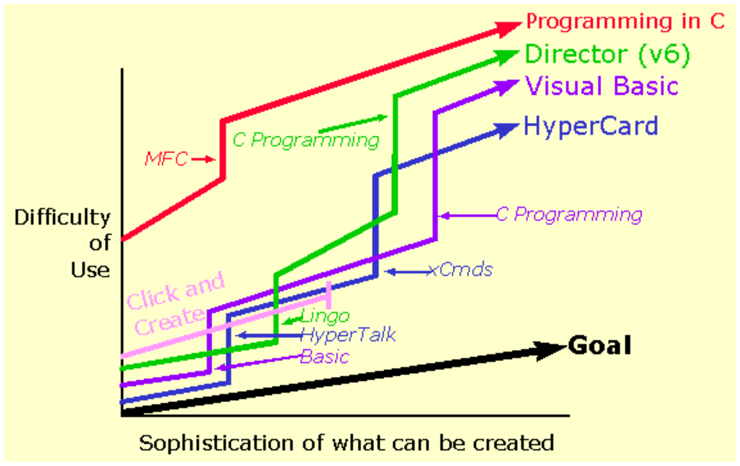

Figure 1: The intent of this graph is to try to give a feel for how hard it is to use the tools to create things of different levels of sophistication. For example, with C, it is quite hard to get started, so the Y intercept is high up. The vertical walls are where the designer needs to stop and learn something entirely new. For C, the wall is where the user needs to learn the Microsoft Foundation Classes (MFC) to do graphics. With Visual Basic, it is easier to get started, so the Y intercept is lower, but Visual Basic has two walls—one when you have to learn the Basic programming language, and another when you have to learn C. Click and Create is a menu based tool from Corel [2], and its line stops because it does not have an extension language, and you can only do what is available from the menus and dialog boxes.

Programming By Demonstration Research

Systems that are easy to learn and easy to use often have a *direct manipulation* front end which significantly reduces the amount of necessary scripting. For example, in Visual Basic, users can place widgets using the mouse and set their properties using dialog boxes, and in Director, many simple movements can be specified by dragging objects with the mouse. One focus of much of our previous work has been how to extend the range of what can be performed by direct manipulation, by allowing more behaviors to be specified *by demonstration* [17]. We have created many systems which have explored various aspects of this problem. A partial list of these systems includes: Peridot [15], Lapidary [34], Tourmaline [16, 35], Marquise [20], Pursuit [14], Silk [10], Topaz [19], and Turquoise [13]. Some of these and many other demonstrational systems are described in a recent book [3].

Our latest system is "Gamut" which allows complete games to be created entirely by demonstration without scripting [12]. Gamut is the PhD research of my student Rich McDaniel and stands for Games Are Made Using This. The game author gives examples of what the end user will do, and then gives examples of what the system will do in response. These examples take the form of editing the objects using the mouse in the usual direct manipulation way. For instance, if after rolling a dice, a piece is supposed to move, the designer would click on the button that rolls the dice, and then select the piece and move it the appropriate number of squares. Gamut uses various Artificial Intelligence algorithms, including plan recognition and decision trees, to generalize from the user's examples. Previous research has shown that systems cannot create correct programs from just a few examples, so Gamut asks for additional information. The designer can draw *guide objects* to show the system important paths and properties that people see implicitly, but which the system cannot notice. The guide objects disappear at run time. The user can also give *hints* to the system about which objects are important, which helps the inferencing algorithm figure out what to do. Since often the previous state is important for determining the current state, Gamut explicitly represents the previous state in the form of a "temporal ghost," which is a dimmed version of the object shown in its former state. Although guide objects and hints were proposed in previous systems, Gamut is the first system to make them work effectively. More information about Gamut is available at http://www.cs.cmu.edu/~richm.

In summary, we have shown that many interesting behaviors can be demonstrated from examples, and that this work can be useful in many different domains. However, it often is necessary to use fairly sophisticated inferencing techniques so that the system will guess correctly and be able to handle realistic situations. Having a static, editable representation of what is inferred is important so that the users know what is going on and can repair it if incorrect.

Natural Programming

Although direct manipulation and demonstrational techniques are powerful and easy to use, they have well-known limitations. In particular, complex sequences of actions are hard to perform accurately, there is no way to represent abstractions, iterations and conditionality, and many repetitive actions are often required. Furthermore, if there is no *static* representation of the program, then the users cannot go back and see what they have done, to revise, edit and reuse prior work. The lack of an editable static representation has been a chief failing of many previous demonstrational and direct manipulation systems [17].

Why Natural?

We are investigating new representations of programs (which includes multimedia scripts) that use textual and graphical elements, and are designed to be more *natural*. We define "natural" as "faithfully representing nature or life," which here implies that it works in accordance with the ways people expect. By "natural programming" we are aiming for the language to work in the way that people who do not have programming experience would expect. Why would this make the programming easier? One way to define programming is the process of transforming a mental plan in familiar terms into one that is compatible with the computer [8]. The closer the language is to the user's original plan, the easier this refinement process will be. This is closely related to the concept of *directness* which, as part of "direct manipulation," is a key principle in making user interfaces easier to use. Hutchins, Hollan and Norman describe directness as the distance between one's goals and the actions required by the system to achieve those goals [9]. Reducing this distance makes systems more direct, and therefore easier to learn. User interface designers and researchers have been promoting directness at least since Shneiderman identified the concept in 1982 [30], but it was not even a consideration in most programming language designs. Green and Petre also argue in favor of directness, which they call *closeness of mapping:* "The closer the programming world is to the problem world, the easier the problem-solving ought to be.... Conventional textual languages are a long way from that goal." [5, p. 146].

User interfaces in general are also recommended to be "natural" so they are easier to learn and use, and will result in fewer errors. For example, Nielsen recommends that user interfaces should "speak the user's language" which includes having good mappings between the user's conceptual model of the information and the computer's interface for it [23, p. 126]. One of Hix and Hartson's usability guidelines is "Use Cognitive Directness," which means to "minimize the mental transformations that a user must make. Even small cognitive transformations by a user take effort away from the intended task" [7, p. 38]. Conventional programming languages require the programmer to make tremendous transformations from the intended tasks to the code design.

For example, to add a set of numbers uses 3 kinds of parentheses and 3 kinds of assignment operators in 5 lines of C code, whereas a single "SUM" operator is sufficient in a spreadsheet [5].

Background Research

Our first step in thinking about the design of new easy-to-learn languages was to thoroughly study the Empirical Studies of Programmers (ESP) and Human-Computer Interaction (HCI) literature. It is somewhat surprising that in spite of 30 years of research in these areas, the designs of new programming languages have generally not taken advantage of what has been discovered. In particular, most languages for scripting multimedia still use features that have been shown to be particularly difficult to learn and use. We cataloged many results which can be used to guide the design of a new programming system [25]. For example:

- The syntax in many languages is a significant barrier, as evidenced by the special symbols needed in the SUM example above.
- One way to ease the entry into programming is to capitalize on the beginner's knowledge about the world. Many languages are based on a **metaphor**, which should be drawn from a concrete real-world system that is familiar to the user audience [31]. Director tries to use the metaphor of a musical score, but it breaks down when applied to interactive situations [36]. We will investigate other metaphors that might be better suited for interactive interfaces.
- When they are stumped, beginners will attempt to transfer knowledge from other domains even if they are not appropriate [8]. This is a problem when the language uses words and symbols in ways that are different from English or math. For example, "AND" is often read to mean "THEN" as in: "We went to the store **and** bought milk," whereas in computers, AND is always used between two things that must both be true at the same time. People often use "AND" when a computer would require the use of "OR," as in: "All people whose names begin with 'A' **and** 'B' should be in the first line." Another example is that many languages use "=" for assignment, but "a = a+3" makes no sense if read as in mathematics. These kinds of features should be avoided in a new language.
- The object-oriented style seems to be harder to learn for novice programmers, and a full inheritance hierarchy has been shown to be too complex for novices, but a fixed two-level inheritance hierarchy is understandable [27].
- … and many others. See [25] for details.

However, there are many significant gaps in the knowledge about how people reason about programs and programming, and how languages can be made more effective. In particular:

- What programming paradigm works best for non-programmers? Professional languages like Java and C++ are object-oriented, but most novice languages, like Visual Basic and HyperTalk are not. And, should the language be textual or graphical?
- How can difficult constructs like iterations and conditionals be minimized and made easier?
- What is the tradeoff between ease of use and correctness? In particular, what is the role of type checking?
- How should abstractions, such as variables, procedures and modules, be presented? To what extent do non-programmers focus only on the concrete examples?
- How can the reuse of procedures, modules and other components be facilitated?

- What terminology and syntax should be used? HyperTalk, AppleScript, and other recent languages from Apple have used a verbose style with words like "the" and "set." Is this better than the more conventional language design using a terse syntax and special symbols like `= := == {}[]()"'` ? Director's Lingo language uses the verbose style ("`set the visible of sprite P1 to false`"), whereas ScriptX and GLPro use the terse style (GLPro: "`layer hey$@loop wait to @loop*30 @loop*15 10 5`").
- Are there other ways to express Boolean concepts without using AND, OR, and NOT, since our results show that people often use these incorrectly?
- What is the role of the environment in overcoming problems in the language? For example, syntax editors can overcome some problems with the language syntax.

Our work in the Natural Programming project [18] is beginning to answer these questions. This forms the basis for John Pane's PhD thesis, and Chotirat "Ann" Ratanamahatana's BS thesis [28].

First study: PacMan

To find out what is *natural*, we are asking people to describe in their own words how they would express algorithms. We have performed two studies so far, and the details of both studies are in [28]. The first study was conducted with 14 fifth graders at East Hills International Studies Academy (a public K-5 school) in Pittsburgh. The children were evenly divided by gender and were racially diverse. They were asked to describe how they would make PacMan move about the screen, eating dots and killing or being killed by monsters. A real risk in designing a study like this is that the experimenter could bias the subjects by the language used in asking the questions. For example, the experimenter cannot just ask: "How would you tell the monsters to turn blue when the PacMan eats a power pill?" because this may lead the participants to simply parrot the question back. Therefore, the participants were shown depictions of the scenarios and asked to write down using their own words or diagrams how they would instruct the computer to implement the actions shown. This enabled the experimenter to show the images and ask general questions to prompt the participants for their responses. As responses, the participants could both draw pictures and write text on the unlined blank paper that we supplied. This allows us to look at whether linguistic (textual) or graphical notations are preferred.

To analyze the data, we gave the participants' responses to five people who are not affiliated with the project, and asked them to classify what they saw in the answers. These raters were all programmers, and were paid to participate. Among the observations from this study are:

- Much of the control (54% of all utterances) was expressed in an "event language" (also called the "production language") style, with rules to control behaviors. For example: "If PacMan loses all his lives, its game over." This result is already reflected in some of today's end-user programming languages. The event-based style used by Visual Basic, Lingo for Director, and HyperTalk for HyperCard, is a form of rule-based style, since the code is of the form "if this event happens, then execute this code."
- Iterations were usually expressed implicitly, by operating on sets of objects. In fact, 95% of the participants' utterances about multiple objects used a set/subset specification. For example, "When PacMan eats all of the yellow balls he goes to the next level." This is instead of using any form of iteration or explicit counting, as would be required in most programming languages.

- When there were multiple options for a conditional, the most frequent construction was a set of mutually exclusive rules, which appeared 37% of the time (for example: “When the monster is green he can kill PacMan. When the monster is blue PacMan can eat the monster”). The next most popular construction was to specify a general condition that was subsequently modified with exceptions, which appeared 27% of the time (for example: “When you encounter a ghost the ghost should kill you. But if you get a power pill you can eat them”). This is in contrast to conventional languages that generally require the conditional to be set up in advance using “ANDs,” “NOTs” and “ORs,” forcing the user to think about all the cases first, and resulting in a complicated Boolean expression.
- The students expected objects to be moving as their normal behavior, and wrote commands that would alter the motion (97% of the utterances). For example, “If PacMan hits a wall, he stops.” This is in contrast to some conventional languages and environments where to make something move requires setting its position at each clock timer tick.
- When inserting items, most subjects (74%) treated the data structures as a list, and just inserted the new item without making room first (as would be required with an array). To sort the items, the subjects usually inserted the item in an indeterminate place, and then specified the sort operation afterwards.

Many researchers have identified control structures as a common area of difficulty for novice programmers [8]. It is interesting that many of the strategies noted above that the subjects used serve to *eliminate* control structures by making loops and conditionals implicit. This provides further evidence that creating a new language that supports these natural tendencies may be easier to learn.

Second study: Spreadsheets

We next performed a follow-on study using database access with both children and adults, and both programmers and non-programmers. This is to investigate how well the observations generalize to other domains and to other populations. We again showed the participants pictures to avoid biasing the answers. This time the pictures were of the database tables before and after various operations, and we asked them to write how the computer should carry out the operations. This study was administered to 19 adults with various levels of programming ability, and to 21 fifth-grade children, four of whom had programmed before. We again developed categories, and the subjects answers were evaluated by 3 independent raters. The analysis of this second study is still on-going, but we do have some preliminary results:

- 90% of the time, multiple objects were handled by operating on the set as a whole, rather than iterating through the individual elements, which is consistent with our first study.
- Also, as in the first study, subjects did not construct complex conditionals using ANDs, ORs, and NOTs. Instead, they would express independent conditions (as in “Black is for G and L. Gold is for B, C, H, J, and S”) or a general case first and exceptions afterwards.
- Most mathematical operations were expressed in a natural language style, such as “*Add 10,000 points to the scores in Round 1 and Round 3*” rather than a mathematical style (“*score + 10000*”) or a programming language style (*“score =*

score + 10000"). However, this natural language style appeared to lead to more errors in the specification, such as failing to handle boundary cases in ranges.

- The subjects used the words AND, OR and THEN in various ways, often inconsistently, and usually in ways that would not work in a conventional programming language. For instance, AND often means "then," as in *"Cross out the highest score,* ***and*** *add the lower scores."*

We are now working on the implications of these results for programming language design in general. For instance, it is clear that "AND" is a problematic word to use in a language, but that built-in support for sets are likely to make the programming language easier to use. By applying these results to multimedia scripting, the result will be a language that will be significantly easier to learn than any of today's languages, and that it will enable a broader range of people to author more sophisticated behaviors.

Future Work

There are many areas to study further in the area of authoring interactive behaviors for multimedia. My groups main focus will be on issues relating to the natural programming approach. One issue that is particularly relevant to this NEC conference is to what extent our results would be different if we studied people whose native language was not English. For example, do the "natural" ways of expressing programming concepts differ for Japanese natives? I would welcome collaborators to help study these issues.

An important next step, which we have not yet begun, is to use the knowledge from the Natural Programming experiments, along with our Demonstrational techniques, to create a new multimedia authoring environment. We hope to raise funding to begin this project soon.

An important focus will be investigating how these ideas can be applied in other domains in addition to multimedia authoring. For instance, we are working on a new programming language for kids that is designed to be more natural [24]. We have also proposed to create natural programming languages for controlling robots used for assembly tasks, for authoring digital video productions, and for performing data visualization and filtering tasks. I believe there are many domains where end-users who are not professional programmers would benefit from using more natural programming techniques.

Conclusions

In conclusion, it is inevitable that consumer audio, video and still photography will migrate to be digital, and new hardware and software will make it possible to view, share and edit video and audio. With this will come an increasing demand by authors for the ability to control how people view and interact with their creations. The World-Wide-Web will continue to expand, with more and more people creating their own Web pages, and wanting to make them more interesting by adding Multimedia. So authoring will become more and more universal, including the need to support interactive behaviors.

By studying *people* and their needs, we can create much more effective authoring environments, that will empower everyone to be able to author their own multimedia presentations that incorporate interesting behaviors. But more research is needed in this area, so I encourage you to think about performing, sponsoring or using research on the human side of the authoring challenge.

Acknowledgements

Dr. Myers is supported in part by a grant from NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

References

1. Caruso, D., "Technology; Digital Commerce; The interactive media industry begins to deconstruct its self-made myths," in *New York Times*1997. New York. pp. D7.

2. Corel, "Click and Create," 1996. Corel Corporation, P.O. Box 3595, Salinas, California, 93912 - 3595. http://www.corel.com.

3. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*. 1993, MIT Press: Camb., MA.

4. Dertouzos, M. and al., e., "ISAT Summer Study: Gentle Slope Systems; Making Computers Easier to Use," 1992. Presented at Woods Hole, MA, August 16.

5. Green, T.R.G. and Petre, M., "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework." *Journal of Visual Languages and Computing*, 1996. **7**(2): pp. 131-174.

6. Hardaway, D. and Will, R.P., "Digital Multimedia Offers Key to Educational Reform." *Communications of the ACM*, 1997. **40**(4): pp. 90-91. April.

7. Hix, D. and Hartson, H.R., *Developing User Interfaces; Ensuring Usability Through Product & Process.* 1993, New York: John Wiley & Sons, Inc. 381.

8. Hoc, J.-M. and Nguyen-Xuan, A., "Language Semantics, Mental Models and Analogy," in *Psychology of Programming*, J.-M. Hoc, *et al.*, Editors. 1990, Academic Press. London. pp. 139-156.

9. Hutchins, E.L., Hollan, J.D., and Norman, D.A. "Direct Manipulation Interfaces," in *User Centered System Design.* 1986. Hillsdale, New Jersey: Lawrence Erlbaum Associates. pp. 87-124.

10. Landay, J. and Myers, B.A. "Interactive Sketching for the Early Stages of User Interface Design," in *Proceedings SIGCHI'95: Human Factors in Computing Systems.* 1995. Denver, CO: pp. 43-50.

11. Lieberman, H., ed. *Learner-Centered Design*. Communications of the ACM, ed. 39. 1996,

12. McDaniel, R.G. and Myers, B.A. "Building Applications Using Only Demonstration," in *1998 International Conference On Intelligent User Interfaces.* 1998. San Francisco, CA: pp. 109-116.

13. Miller, R.C. and Myers, B.A., *Creating Dynamic World Wide Web Pages by Demonstration.* Carnegie Mellon University School of Computer Science, CMU-CS-97-131 and CMU-HCII-97-101, 1997,

14. Modugno, F. and Myers, B.A., "Visual Programming in a Visual Shell -- A Unified Approach." *Journal of Visual Languages and Computing*, 1997. **8**(5/6): pp. 276-308.

15. Myers, B.A., "Creating User Interfaces Using Programming-by-Example, Visual Programming, and Constraints." *ACM Transactions on Programming Languages and Systems*, 1990. **12**(2): pp. 143-177.

16. Myers, B.A. "Text Formatting by Demonstration," in *Proceedings SIGCHI'91: Human Factors in Computing Systems.* 1991. N.O., LA: pp. 251-256.

17. Myers, B.A., "Demonstrational Interfaces: A Step Beyond Direct Manipulation." *IEEE Computer*, 1992. **25**(8): pp. 61-73.

18. Myers, B.A., *Natural Programming: Project Overview and Proposal.* Technical Report, Carnegie Mellon University School of Computer Science, CMU-CS-98-101 and CMU-HCII-98-100, 1998, Pittsburgh.

19. Myers, B.A. “Scripting Graphical Applications by Demonstration,” in *Proceedings SIGCHI'98: Human Factors in Computing Systems.* 1998. Los Angeles, CA: pp. 534-541.

20. Myers, B.A., McDaniel, R.G., and Kosbie, D.S. “Marquise: Creating Complete User Interfaces by Demonstration,” in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 293-300.

21. Myers, B.A., Smith, D.C., and Horn, B. “Report of the `End-User Programming' Working Group,” in *Languages for Developing User Interfaces.* 1992. Boston, MA: Jones and Bartlett. pp. 343-366.

22. Newell, A. and Card, S.K., “The Prospects for Psychological Science in Human-Computer Interaction.” *Human-Computer Interaction*, 1985. **1**(3): pp. 209-242.

23. Nielsen, J., *Usability Engineering.* 1993, Boston: Academic Press.

24. Pane, J.F. “Designing a Programming System for Children with a Focus on Usability,” in *Adjunct Proceedings SIGCHI'98: Conference Summary: Human Factors in Computing Systems.* 1998. Los Angeles, CA: pp. 62-63.

25. Pane, J.F. and Myers, B.A., *Usability Issues in the Design of Novice Programming Systems.* School of Computer Science Technical Report, Carnegie Mellon University, CMU-CS-96-132, 1996, Pittsburgh, PA. Also appears as Carnegie Mellon University Human-Computer Interaction Institute Technical Report CMU-HCII-96-101.

26. Papert, S., *Mindstorms: Children, Computers, and Powerful Ideas.* 1980, New York: Basic Books. 230.

27. Pausch, R., Conway, M., and DeLine, R., “Lesson Learned from SUIT, the Simple User Interface Toolkit.” *ACM Transactions on Information Systems*, 1992. **10**(4): pp. 320-344.

28. Ratanamahatana, C.A., *Evaluating What is Natural for Beginners.* BS Thesis, Computer Science Department Carnegie Mellon University, 1998, Pittsburgh, PA.

29. Schank, R.C., “Active Learning through Multimedia.” *IEEE Multimedia*, 1994. **Spring**: pp. 69-78.

30. Shneiderman, B., “Direct Manipulation: A Step Beyond Programming Languages.” *IEEE Computer*, 1983. **16**(8): pp. 57-69. Aug.

31. Smith, D.C., Cypher, A., and Spohrer, J., “KidSim: Programming Agents Without a Programming Language.” *Communications of the ACM*, 1994. **37**(7): pp. 54-67.

32. Soloway, E., “Learning to Program = Learning to Construct Mechanisms and Explanations.” *CACM*, 1986. **29**(9): pp. 850-858. Sep.

33. Stasko, J., Badre, A., and Lewis, C. “Do Algorithm Animations Assist Learning? An Empirical Study and Analysis,” in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 61-66.

34. Vander Zanden, B. and Myers, B.A., “Demonstrational and Constraint-Based Techniques for Pictorially Specifying Application Objects and Behaviors.” *ACM Transactions on Computer-Human Interaction*, 1995. **2**(4): pp. 308-356.

35. Werth, A.J., *Tourmaline: Formatting Document Headings by Example.* 1992, Information Networking Institute, Carnegie Mellon University.

36. Wolber, D., “An Interface Builder for Designing Animated Interfaces.” *ACM Transactions on Computer-Human Interaction*, 1997. **4**(4): pp. 347-386. Dec.