A Programming System for Children that is Designed for Usability

John F. Pane

Computer Science Department and Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
pane+esp7@cs.cmu.edu

Abstract

This paper proposes a new programming language and environment for children. This system will be designed to be easy to learn and use, without sacrificing the power necessary to create sophisticated programs that rival commercial software such as games and simulations. Throughout the design and refinement of this system, I will apply prior results from empirical studies of programmers and the psychology of programming, my own empirical studies about the ways that non-programmers naturally express solutions to programming tasks, and usability testing.

Introduction

Most children who use computers are not programmers. Yet many would like to create their own software, and to customize the software that they have. Typically, they would like to create programs that are similar to the ones that they use, such as games and educational simulations, which are highly interactive and graphically-rich. But there are no suitable tools for this kind of programming task that are easy enough for beginners to pick up and use. The powerful programming environments used by commercial software developers are clearly inappropriate for beginners because they are very complex and assume that the user has extensive programming expertise. Other tools that were designed for beginners are effective only in limited domains, such as Visual Basic for creating applications that are based on forms and dialog boxes, or spreadsheets for tabular numeric applications. More general languages for beginners, such as Logo, are easier to learn than languages for professionals, but are not powerful enough to create highly interactive graphically-rich programs like the commercial software that children use frequently. This gap, between the quality and sophistication of commercial software and the programs that children can create themselves, leads to disappointment and frustration.

There is opportunity to narrow this gap, and my research will use three complementary strategies:

1. Develop a *card game* metaphor as a new way of thinking about programming. This familiar real-world activity will provide concrete analogs for many of the essential concepts of programming that are currently abstract and difficult to learn.
2. Apply results from research on the psychology of programming and human-computer interaction, supplemented by my own empirical studies, to influence the design of a programming language and environment so that it will be easier to learn and use.
3. Use modern software technology to simplify the programming process, including direct manipulation and demonstrational techniques, programming environment technologies like structure editors, and mechanisms for sharing software components and integrating them into programs.

The result will be a programming language and environment based on the card game metaphor, targeted at the needs and abilities of children and other non-programmers.

Metaphor. An appropriate concrete model can have a strong positive effect on the usability of a programming language [Mayer 1989]. Perhaps the most successful end-user programming system to date is the spreadsheet, due in part to its familiar and effective metaphor of financial tables [Nardi 1993]. Unfortunately, the spreadsheet metaphor lacks generality, and thus many kinds of programs are not well-suited to implementation in a spreadsheet. In contrast, many programming languages for professional programmers are based on the very general computational model of the von Neumann machine, which has no familiar physical world counterpart. This research will introduce a new computational model that is amenable to general purpose computation, and yet has a strong metaphorical tie to a familiar real-world system.

I am developing a metaphor for programming around the general concept of a *card game* because it is familiar to children, and it provides a framework that matches surprisingly well to many of the essential concepts of programming. This metaphor is described below in “Preliminary Design: the Card Game Metaphor”.

Human-Centered Approach. More than a decade ago, Allen Newell 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. [Newell 1985, p 212-3]

I will avoid making this same mistake in my project, by acknowledging the importance of the human aspect of pro-

gramming, and taking an approach that focuses on usability throughout the design process. I will utilize studies of the human side of the programming process, as well as general Human Computer Interaction (HCI) principles. I have summarized research about beginner programmers, from the fields of The Psychology of Programming and Empirical Studies of Programmers (ESP), in [Pane 1996]. Surprisingly, these results do not seem to have influenced the design of popular new languages such as Java. In contrast, the design of my system will be heavily influenced by the prior work, and my research will contribute additional empirical studies to the field. For example, I have already begun using empirical studies to investigate how non-programmers express their solutions to a set of programming tasks. I found that most people use a rule-based or event-based approach, so my system will be designed to support this tendency.

Technology. This research will apply a broad set of technologies to making the system easy to use. For example, the system will take advantage of user-interface technologies such as direct manipulation and demonstrational techniques, and programming environment technologies such as structure editors to reduce tedious operations and make it easier to build error-free programs. It will also apply software engineering technologies such as component integration mechanisms to enable users to extend their programs by harnessing and reusing existing components, resources and infrastructures, such as other programs or data on the worldwide web.

Domain. My system will be capable of producing a full range of programs. However, the most successful end-user programming systems are domain-specific, so my system will focus on the class of software that children would like to create, such as games and educational software. These are highly-interactive and graphical, and contain animations, simulations and multimedia. The card game metaphor, along with the graphics primitives that I choose to build into the system, will facilitate the creation of programs within this domain without sacrificing the general capabilities of the underlying programming language. This is similar to the way that the turtle metaphor in Logo transformed a general purpose language into one that facilitates the building of programs based on “turtle” graphics.

My thesis is that a programming system that uses a card game metaphor for computation, and embodies principles from the psychology of programming and human-computer interaction, will enable children and other non-programmers to create sophisticated programs containing animations, simulations and multimedia. The new metaphor and language, with the supporting technology in the environment, will make the essential concepts of programming easy to learn, and yet will be more powerful than existing systems for beginners.

To evaluate this thesis, I will develop a new programming language and environment called HANDS, which stands for **H**uman-centered **A**dvances for **N**ovice **D**evelopment of **S**oftware. This system will be based on the card game metaphor, and will be designed with usability as a fundamental goal. I will utilize user studies to evaluate and refine the system so that it is easier to learn and use than existing systems for beginners, and so that users can create sophisticated programs that rival commercial software such as games and simulations.

Preliminary Design: the Card Game Metaphor

In my review of research examining beginner programmers, one of the prominent themes to emerge is the importance of a familiar concrete model for the computational system. Many languages fail in this respect because they are based on models that are too abstract or unfamiliar to beginners. One way to obtain these critical attributes of concreteness and familiarity is to use a well-known real-world system as a metaphor for the computational machine. In exploring alternatives for the HANDS system, I found that a card game metaphor has many compelling attributes. First, it is a familiar concept for my target audience. Second, the basic elements of a card game (cards, players, hands, piles, table, face values, suit, etc.) are concrete and do not require abstract thinking or imagination. Finally, there seems to be a straightforward mapping between a card game and the essential concepts of programming, so that programs can be represented in a complete and consistent manner within the metaphor. Thus I hope many of the difficult aspects of programming will be easier because they are more concrete in the HANDS system, allowing beginners to quickly get started in programming by transferring their knowledge from card games.

Motivated by some preliminary analysis of the way non-programmers describe solutions to programming tasks, HANDS will use a rule-based (or production system) style. In such systems, a program is a collection of rules, each consisting of a condition and one or more actions. When a rule is evaluated, the actions may be performed depending on whether the condition holds true. Apple Computer’s programming system for children named Cocoa [Smith 1996] also uses a rule-based approach, although Cocoa primarily uses graphical rewrite rules, while HANDS will use more conventional text-based rules.

The HANDS system will support the development of interactive graphical interfaces. Many modern development systems for this kind of software, such as Microsoft’s Visual Basic [Halvorson 1996] and the HyperTalk language in Apple’s HyperCard [Goodman 1987], use an event-based model of interaction. The event model permits asynchronous actions from outside the program to be dispatched to the appropriate handlers within the program, although these dispatch mechanisms are sometimes quite complicated. The event paradigm is close to the rule-based paradigm in HANDS (the conditions of rules can be predicated on events).

Elements of a Card Game. A program in the proposed HANDS system will consist of one or more *players*, representing modules or processes, sitting at a *table* with a *game board* on it. The table represents the mechanism for sharing data among players, in the style of “blackboard architectures” from AI and the “tuple space” of Linda [Carriero 1989]. The game board represents the mechanism for displaying information to the screen and receiving input from the user.

Each player will hold a *hand* of active *cards* and will also have a *pile* of inactive cards. Cards represent pieces of programs and

data. The cards in a player's hand and pile are private, providing information hiding. There also may be cards on the table, which are visible to all players, providing global or shared memory. A new card can be brought into the game by getting one from an unlimited supply of *blank cards* and adding information to it. A card can be removed from the card game by placing it into a *discard pile*. These operations are analogous to memory allocation and deallocation.

There will be four *suits*, or types of cards: *rule cards*, *data cards*, *event cards*, and *game cards*. Rule cards contain instructions that are executed by the system, in the form of if-then rules. Data cards will have one or more named slots for holding information, and thus are analogous to variables or records. Event cards are special data cards that are managed by the system. They will be placed on the table by the system to notify players about events that have taken place, such as input from the user, or collisions between objects. Game cards provide an encapsulation mechanism whereby the details of a game are abstracted into an active card that can then be used in other games. This abstraction mechanism will be discussed below.

When the program is being developed, all the elements will be visible, so the user can create and operate on them by direct manipulation. For example, a card can be placed at a particular location on the board by dragging it with the mouse. When the finished program is running, however, only the game board will be visible to the user. The programmer will be able to switch between running and developing at any time, allowing smooth transitions between editing, running, testing, and debugging programs. Cards that are located on the table (but not on the board) will be displayed as little card icons. However, when data cards are on the board they will be displayed in a unique way – only their appearance and/or value will be shown. Actually, every graphical object on the board will really be a data card that is being displayed in this special way. If there is a picture in the appearance slot, the appearance will be rendered on the board. Optionally, the contents of the value slot can be displayed. In this case, if there is an appearance the value will be superimposed on it. Otherwise, the value will be displayed alone. In either case, the value will be shown exactly as it is appears on the card. This makes it easy to output strings and numbers, and attributes such as font, size, alignment, color can be set by directly manipulating the value field on the card. A double-click on a graphic on the board will bring up a window showing the full card.

Mechanisms. HANDS will provide built-in support for handling input from the user, for moving and animating graphical objects, and for detecting collisions among those objects. Each of these built-in features will be represented as players at the card table. For example, the *event manager* player will receive input from the user such as by mouse clicks and keystrokes, post event cards onto the table representing these events. Similarly, the *collision detector* player will notice when graphical objects have collided, and will post collision events onto the table. If other built-in capabilities are provided in the HANDS system, additional system players will represent them at the table. Similarly, this mechanism could be used for importing functionality into a card game from libraries, and for importing functionality from other existing programs or components. In the latter case, the player could act as a translator between the protocols and data of an external component and the HANDS system where all program data is represented by cards.

Players correspond to processes or modules in conventional programming systems, and will not necessarily correspond to actual people playing the game. For example, in a PacMan implementation, there might be one player for the PacMan, one for each of the monsters, and possibly one to handle all the dots on the board since some of them blink and they must be counted to see when the last one is eaten (to go to the next level). Distributed programs could be represented with multiple boards, one for each display, and players could transparently be either be local or on a remote machine.

When the program is running, each player around the table is given a turn, beginning with the first system player (the animator player) and moving clockwise from player to player. During a player's turn, each rule card in the player's hand is evaluated, in the order that they appear in the hand at the beginning of the turn. Only the cards that are in the player's hand at the beginning of the turn will be evaluated during that turn, and each one will be evaluated exactly once during the turn. This evaluation order corresponds to the ordering of rules in event-based languages like Sassafras [Hill 1986]. During debugging mode, the system will indicate whose turn it is by giving the player a yellow halo. In that player's hand, the currently-executing card is also given a yellow halo.

Naming and Visibility. Cards in the following locations are visible to a player: the player's hand, the player's pile, the table and the board. However, the cards in the hands or piles of other players are not visible to the player, to help enforce modularity. Of course, all cards will be visible to the programmer so that the program can be constructed and debugged, but each player will have to send messages to the other players to retrieve or modify the data on those players' cards.

In the course of evaluation, a rule may act upon the visible cards in the following ways:

- examine the information on the cards,
- modify the information on the cards,
- move cards from one visible location to another
- move cards to the discard pile
- draw blank cards and put information on them,
- copy an existing card, which creates a new card that differs from its prototype only in its name,
- give a data card to another player, causing it to immediately become a part of the other player's hand.

In addition to its contents, each card will have identifying information at the top: its name (which must be unique), the name of a group that the card is a member of (optionally left blank), and its suit (rule, data, event, or game). This provides a limited three-level naming hierarchy for cards, which permits the programmer to not only refer to individual cards, but also to collections of cards. Full inheritance has been shown by previous systems to be too complex for novices, but a fixed 2-level inheritance hierarchy has been shown to be understandable [Pausch 1992].

For example, suppose the event manager player posts a mouse event card to the table as a result of the user clicking on the "+" button of a calculator. Here are four ways that a rule card could refer to this card:

- *an event*, in which case other mouse clicks, and keystrokes and other events would also match.
- *a click*, in which case other mouse clicks, but not non-click events, would match.
- *click12345*, in which only a click with this name would match.
- *a click whose target is +*, which corresponds to the associative memory access style of Linda.

The hierarchical naming structure can be used to manipulate groups of items together, and to set the context of a search. This mechanism will handle many of the situations where traditional programming languages would require explicit looping. For example, *move all cards with value = 8 to my hand*.

Events will be automatically posted to the table by the event manager player during his turn at the beginning of each round, and removed from the table during his next turn if they haven't already been removed by one of the other players. Players that respond to the event have the option of leaving the event on the table for other players to see, or to remove it. This very simple rule is in contrast to event based systems like HyperCard and Visual Basic, which have complex rules about how events are dispatched and handled. The system automatically fills in the target field of an event card. For example, when there is a click on the board, the card representing the frontmost graphical object under the click will be the target of the event. A system inquiry can be made in cases where the programmer wants a longer list of possible targets, such as all of the graphical objects under the click instead of the frontmost one.

Domain Specific Functions. In addition to handling user input events, the system will provide players that handle the automatic maintenance of several other details that commonly occur in many programs today, especially the games, simulations and animated stories that kids want to create. The system will provide a *timer* player who can be used for coordinating actions and specifying rates and durations. The system will automatically maintain a *location* slot containing the coordinates of the card's on the board. An *animator* player will automatically move cards around on the board if they have a value in the *velocity* slot, updating the location fields and the rendering of the appearance on the table. I will investigate ways to specify more complex motions by filling an animation slot with appropriate information such as path and duration, based on earlier work on easily specifying animations [Myers 1996]. Furthermore, the *collision detector* player will detect collisions of objects, posting collision events so that the objects can respond accordingly. While the built-in functionality of these system players will probably not be viewable as rule cards, any configuration options will be represented by cards in these players' hands.

Abstraction Mechanism. One of the difficult challenges of designing this new system is to find a good way to provide an abstraction mechanism within the card game metaphor. It appears that it will be necessary to violate the metaphor in order to provide this feature, since real-world card games do not already have a feature that I can use to represent this. Thus I am trying to invent a concrete representation for abstraction that is a simple extension of the existing card game model. While this is still to be finalized, I am currently exploring the idea of using a *game card* to represent a "recursive" card game, which I will call a *subgame*. Internally, the subgame would be complete with players, a table and a game board, while the external view would be simply a card with an appearance and the other attributes of visible data cards. The appearance of the game card would always reflect the appearance of the game board in the subgame. Parameter passing could be provided by mapping additional slots on the game card to data cards on the table of the subgame that have the same names and values. The appearance and the parameters would be the only parts of the subgame that would be visible via the game card. All other aspects of the subgame, including the other cards and the players, would be invisible from the outside, providing abstraction barriers.

The recursive card game would run in parallel to the game that contains it (and any other subgames that exist in the program). From the outside, a game card would appear to be very similar to other data cards on the game board, except they will seem to be autonomous or "alive", since externally they can appear to change without explicit action by any visible players.

In this scenario for implementing abstractions as parallel processes in subgames, I must still work out several issues. One example is synchronization: when a player writes into the parameter slots on a game card she would be passing parameters to the subgame. However, the subgame would need to "know" when all of the parameters are ready before beginning to evaluate them. A similar synchronization problem occurs when the subgame is returning results to the enclosing game.

Sharing and Reuse. Cards, and thus abstracted card games, can be shared with other users, by importing them into other card games. The system is intended to be an open system. To support simulations, there will be a simplified interface to mathematics and arithmetic, possibly using a calculator like in Cocoa or the dialog boxes used in Model-It [Soloway 1996]. There will be a standard interface with which expert programmers can code new primitives in high-performance languages such as C or C++ and encapsulate one or more of them into game cards for import into future programs. This is a way to provide libraries for specific domains which would appear to the novice user of HANDS as built-in primitives.

Technology

In building the HANDS system, I will take advantage of existing technologies wherever possible to improve the usability of the system. For example, direct manipulation and demonstrational techniques can reduce the effort required to specify portions of the program or data. Programming environment technology can simplify tedious programming tasks and reduce syntax errors, provide active assistance in helping the user to create programs that are semantically correct, and provide sophisticated query and navigation mechanisms to assist the user in understanding and debugging the parts of the program that are already written.

The system will allow the user to take advantage of existing components that are already available on their computer or on the worldwide web, by providing mechanisms for harnessing them within their HANDS program. The game card mechanism will be used to represent interfaces to these external resources, thus allowing them to be integrated into the user's program.

Use of a Textual Language

There has been a widespread expectation that visual programming systems are superior to textual systems. However, to date nobody has been able to prove that a visual language is superior to textual languages for all tasks. To the contrary, often textual languages are actually superior to visual languages [Green 1992]. It is not necessarily always difficult to learn to program in a textual language. In fact, the most successful end-user programming system (among adults) is spreadsheets, which uses a text-based language for formulas [Nardi 1993].

To minimize the difficulty of the textual language, I will incorporate knowledge about how people naturally express problem solutions, and knowledge about aspects of current programming languages that are particularly confusing. I will also create an editor and evaluation environment (interpreter and debugger) that overcome many of the problems that have been noted with textual languages. For example, many of the problems that beginners have with textual languages can be attributed to problems with punctuation and other syntactic embellishments that are used to allow the compiler to efficiently and correctly parse a stream of text. The new language will eliminate the need for most of this punctuation by using structure-editor or form-based technology, which provides slots for the unambiguous placement of program elements.

Even in textual languages, it is helpful to permit the user to define parts of the program by direct manipulation or by example [Lewis 1987]. So, HANDS will support direct manipulation. For example, the user will be able to draw a picture and use it as program data, rather than having to write program code to generate the picture. The user will be able to move objects and change their attributes directly like in a drawing package, and have the environment notice these manipulations and include them into the program. This is a simple form of "programming by demonstration" [Myers 1992] which has been shown to be a good way of allowing non-programmers to get started with programming [Modugno 1994] but which tends to be limited to only small programs. Incorporating these syntax editor and demonstrational techniques are expected to alleviate many of the problems that researchers have identified with prior textual languages.

Domain

I expect that middle school children, as well as older children and their teachers, will be able to use HANDS to create complete and interesting games and educational simulations of the form of PacMan, Number Munchers, Living Books, PlayRoom, BusyTown, SimCity and its siblings, and Model-It. I expect that HANDS will capable of implementing all of the kinds of simulations that can be programmed in Cocoa, as well as the kinds of animations that can be set up in KidPix. Simple direct manipulation programs or hypertext programs, like the ones that can be created using Visual Basic and HTML editors, will be easy to produce in HANDS. Multi-user games and activities should be easy to program in HANDS since it will allow some of the card game players to be on remote machines. Text editing and clickable links (such as for Living Books and HTML pages for the WWW) will also be supported in HANDS. Initially, HANDS will only provide 2-D graphics, but there is nothing inherent in the language that precludes extension to 3-D.

The end-user programming community has developed a repository of Problem-Centered Visual Language Benchmarks [Hübscher 1996], which are intended for the visual languages and the end user programming communities to test the effectiveness of new programming approaches. The benchmarks span many different kinds of problems which highlight strengths and weaknesses of various programming languages. Thus far, no single language has provided solutions to every problem in the benchmark. It is my goal that HANDS will handle all of these problems, demonstrating that the system is powerful, complete and general purpose.

Background Research

The following paragraphs summarize the most important findings from an extensive review of research relevant to beginner programmers from the Empirical Studies of Programmers and Psychology of Programming communities [Pane 1996].

Metaphor. With its card game metaphor, HANDS will attempt to capitalize on the beginner's knowledge about the world. The card game seems to be a good choice because it is drawn from a concrete real-world system that is familiar to the user audience, as recommended by [Lewis 1987]. It will provide a framework for understanding the system, instead of requiring the user to learn a collection of rules that may otherwise seem arbitrary, and it will allow the user to infer how parts of the system work. There will be a close correspondence between the system and this metaphor: features in one will exist in the other, and these features will behave consistently in the two systems, as recommended in [Halasz 1982].

Human Interpreter Problem. When they are stumped, beginners will attempt to transfer knowledge from other domains even if they are not appropriate [Hoc 1990]. One example is the human interpreter problem, where the program is read in a natural language manner, leading to incorrect interpretation [Bonar 1988]. HANDS will attempt to reduce this problem by carefully selecting the syntax and keywords of the language to avoid the use of words and phrases that might conflict with natural-language interpretations.

Misinterpretation and Typos. Programmers are susceptible to incorrect interpretations due to misleading appearances. One example is indentation, which if incorrect can mislead the programmer about whether or not code is within a control structure [du Boulay 1989]. Since HANDS will use a structure editor, indentation will always correspond to the computer's interpretation of the program. Correct indentation has been shown to improve comprehension [Cunniff 1989]. HANDS may also use

additional cues to indicate nesting, similar to the notations used in outlines. A related problem is that many textual languages assign vastly different interpretations to constructs that have only subtle distinctions in syntax [Fitter 1979]. The HANDS language will be designed so that a typographical error or cognitive slip will likely result in an invalid program, so that the system will detect the error for the user, as suggested by [Green 1996].

Memory Burden. Human working memory limitations account for a large part of the inferior performance of novice programmers [Anderson 1985]. HANDS will use typography, color and graphical embellishments to attract attention to, and improve comprehension of, important information, as suggested in [Fitter 1979]. We will try to avoid using these signals to highlight information of questionable importance, such as the keywords of the language [Baecker 1986].

Immediate Feedback. HANDS will provide immediate feedback and encourage incremental testing of programs as recommended in [Green 1996].

Debugging. Novices tend to introduce new bugs while they are debugging [Gugerty 1986]. HANDS will try to help the programmer track recent changes, and to back out of them when it becomes clear that they are not beneficial, through use of multi-level undo and checkpointing.

High-level Domain-specific Primitives. One way to define programming is the process of transforming a mental plan in familiar terms into one that is compatible with the computer [Hoc 1990]. The closer the language is to the user's original plan, the easier this refinement process will be. A very low-level language with many simple primitives requires the user to synthesize higher-level operations. This is one of the great difficulties in programming [Lewis 1987]. When there are many different choices, more planning is required, and this increases the likelihood of backtracking and revision, which slows the programmer [Gray 1987]. HANDS will provide high-level, domain-specific primitives for the highly-interactive visually-rich applications that I am targeting, and will be extensible so that similar high-level primitives can be provided for other domains.

Natural Language vs. Natural Expressions. The HANDS programming language will not be based directly on natural language, since the computer cannot reliably understand human conversation. Natural-language conversation among humans is not precise, and relies on shared context, cues that indicate misunderstanding, and clarification dialogs [Grice 1975]. However, in designing the HANDS language, I look at how people express their ideas. When asked to write step-by-step informal natural language procedures, non-programmers tend to use certain phrases to indicate program constructs such as loops [Bonar 1988]. However, they omit many details, expecting a human-like interpreter to fill them in. The system can support the programmer by providing constructs that are as similar as possible to the common natural language phrases, and directing the programmer's attention to necessary details that may otherwise be overlooked.

Terseness and Conciseness. HANDS will not strive for elegance and terseness in its language, because these features are not useful for beginners [Mendelsohn 1990]. Conciseness may be helpful in some cases, such as eliminating redundancies and excessive punctuation and allowing optional information that has intelligent defaults [Cordy 1992], but this will not be taken to extremes. Novice programmers are more verbose than experts in describing tasks to computers or to humans [Onorato 1986], so HANDS will try to accommodate this characteristic of beginners.

Notation. The HANDS language will be self-consistent, and its rules will be uniform, the meanings of keywords will be independent of context, and exceptions will be minimized, as recommended in [du Boulay 1989]. HANDS will provide extra cues in control structures, since these help beginners [Sime 1977].

General HCI Principles. In addition to the above, the HANDS design will apply general Human-Computer Interaction (HCI) principles. These include such recommendations as: be consistent, provide feedback, prevent errors, etc. [Nielsen 1993, pp. 115-163].

Empirical User Studies

I will conduct new empirical studies to fill in important missing areas of research about novice programmers. This series of studies will be mostly with elementary school and middle school children who do not already know how to program, but who have familiarity with a variety of computer applications. The studies will be designed to discover how they think and talk about programming concepts. The first area I've chosen to look at is the language and structure of the solutions that non-programmers give to programming tasks. Throughout the design of the system, I will conduct other studies to focus on additional questions that arise, and to test my design ideas.

Pilot Study. To see if a study would be likely to get useful results a pilot study was conducted with 11 fifth and sixth graders at Winchester-Thurston School in Pittsburgh in June, 1996. Students were asked to describe how they would make a PacMan move about the screen, eating dots and killing or being killed by monsters. Among the observations were:

- The students expected objects to be moving as their normal behavior, and wrote commands that would affect the motion. For example, "If PacMan hits a wall, he stops." This is in contrast to conventional systems where to make something keep moving requires the continuous issuing of commands.
- Most of the control was expressed in an "event language" or "production system" style, with rules controlling behaviors. For example, "If PacMan loses all of his lives, it's game over."
- The students preferred to express the general case first, and then later modify it with exceptions. For example, "When you encounter a ghost the ghost should kill you. But if you get a little pill you can eat them." This is in contrast to conventional languages that generally require the conditions to be written before the actions, requiring the user to think about all the cases before beginning to construct the expression.
- Operations like counting, which are traditionally expressed with iteration, were usually expressed implicitly by operating on sets of objects. For example, "When PacMan eats all of the

yellow balls he goes to the next level." This capability is not provided by any of today's novice languages.

Subsequent Study. As a first step in a subsequent study, the programming task was broken down into the essential and useful concepts required when creating interactive games and simulations. In addition to seeing how the subjects discuss programming in general, this study investigates how they express these particular concepts. Some examples are: variables, assignment of values, and initialization; comparison of values and boolean logic; incrementing and decrementing counters and other arithmetic; iteration and looping; conditionals and other flow control; searching and sorting; animation; parallelism (multiple things happening at the same time); collisions and interactions among objects; and responding to input from the user.

In addition to investigating how children think about the fundamental concepts, I am also interested in the vocabulary they use for describing the actions. For example, is "when" or "if" more popular as a conditional? How do people express conjunction and disjunction (since "and" and "or" are commonly used incorrectly [du Boulay 1989])? The sessions were audiotaped to make sure I can go back and look at the specific words used as necessary.

A real risk in designing this study was that the researcher would bias the students by the language used in asking the questions. For example, the researcher could not just ask: "How would you tell the monsters to turn blue when the PacMan eats a power pill?" because I expect this would lead the students to simply parrot the question back. Therefore, the students were shown small excerpts from working games and other familiar applications running on a computer, and then asked: "How would you make the computer do this?" We used a collection of pictures and QuickTime movie clips showing various situations that arise in PacMan which illustrate the various concepts. This study was conducted with 13 fifth graders at East Hills Elementary School in Pittsburgh during May 1997, and the results are forthcoming.

Related Work

In addition to the studies of programmers mentioned above, there have been a number of systems which have tried to be easy-to-learn and embody appropriate principles. The most successful and widely used language for children has been Logo [Papert 1980]. Logo uses the metaphor of a turtle executing a text-based program to move around and draw in a graphics world. Rehearsal World is a programming environment based on a theater metaphor, which was designed for adult curriculum designers with no programming experience [Gould 1984]. The system provides a concrete manifestation of the underlying Smalltalk programming system.

Agentsheets and its descendants are a family of programming environments for end-users, based on autonomous communicating agents in a two-dimensional grid-based world [Repenning 1995]. These systems combine graphical rewrite rules and programming by example to simplify programming. One descendent of Agentsheets is a modern programming environment for children named Cocoa [Smith 1996]. ToonTalk is a concurrent constraint programming language based on the metaphor of a video game [Kahn 1996]. A cartoon world is provided that provides concrete realizations of all of the concepts required in concurrent constraint programming.

Acknowledgments

This paper is derived from a draft of my thesis proposal. I would like to acknowledge the valuable contributions to this research from my advisors, Brad Myers and David Garlan, and from John Chang.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037 Arpa Order No. B326, and partially by NSF under grant number IRI 9319969. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, NCCOSC or the U.S. Government.

References

Anderson, J.R. and R. Jeffries (1985). "Novice LISP Errors: Undetected Losses of Information from Working Memory." Human-Computer Interaction 1: 107-131.

Baecker, R. (1986). Design Principles for the Enhanced Presentation of Computer Program Source Text. Proceedings of CHI'86 Conference on Human Factors in Computing Systems. M. Mantei and P. Orbeton. Boston, ACM: 51-58.

Bonar, J.G. and R. Cunningham (1988). Bridge: Tutoring the Programming Process. Intelligent Tutoring Systems: Lessons Learned. J. Psotka, L. D. Massey and S. A. Mutter. Hillsdale, NJ, Lawrence Erlbaum Associates: 409-434.

Carriero, N. and D. Gelernter (1989). "Linda in Context." Communications of the ACM 32(4): 444-458.

Cordy, J.R. (1992). Hints on the Design of User Interface Language Features – Lessons from the Design of Turing. Languages for Developing User Interfaces. B. A. Myers. Boston, Jones and Bartlett Publishers: 329-340.

Cunniff, N., R.P. Taylor and J.B. Black (1989). Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 419-429.

du Boulay, B. (1989). Some Difficulties of Learning to Program. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 283-299.

Fitter, M.J. and T.R.G. Green (1979). "When Do Diagrams Make Good Computer Languages?" International Journal of Man-Machine Studies 11: 235-261.

Goodman, D. (1987). The Complete HyperCard Handbook. New York, Bantam Books.

Gould, L. and W. Finzer (1984). Programming by Rehearsal. Palo Alto, CA, Xerox Corporation**:** 133.

Gray, W. and J.R. Anderson (1987). Change-Episodes in Coding: When and How Do Programmers Change Their Code. Empirical Studies of Programmers: Second Workshop. G. M.

Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex: 185-197.

Green, T.R.G. and M. Petre (1992). When Visual Programs are Harder to Read than Textual Programs. Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). G. C. van der Veer, M. J. Tauber, S. Bagnarola and M. Antavolits. Rome, CUD.

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

Grice, H.P. (1975). Logic and Conversation. Syntax and Semantics III: Speech Acts. P. Cole and J. Morgan. New York, Academic Press.

Gugerty, L. and G.M. Olson (1986). Comprehension Differences in Debugging by Skilled and Novice Programmers. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 13-27.

Halasz, F. and T.P. Moran (1982). Analogy Considered Harmful. Proceedings of Human Factors in Computer Systems: 383-386.

Halvorson, M. (1996). Learn Visual Basic Now: Everything You Need to Teach Yourself the Newest Version of Microsoft Visual Basic. Redmond, Washington, Microsoft Press.

Hill, R.D. (1986). "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction -- The Sassafras UIMS." ACM Transactions on Graphics 5(3): 179-210.

Hoc, J.-M. and A. Nguyen-Xuan (1990). Language Semantics, Mental Models and Analogy. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 139-156.

Hübscher, R., Ed. (1996). Problem-Centered Visual Language Benchmarks, http://www.cc.gatech.edu/people/home/roland/VL-Benchmarks.html.

Kahn, K. (1996). "Drawings on Napkins, Video-Game Animation, and Other Ways to Program Computers." Communications of the ACM 39(8): 49-59.

Lewis, C. and G.M. Olson (1987). Can Principles of Cognition Lower the Barriers to Programming? Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex: 248-263.

Mayer, R.E. (1989). The Psychology of How Novices Learn Computer Programming. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 129-159.

Mendelsohn, P., T.R.G. Green and P. Brna (1990). Programming Languages in Education: The Search for an Easy Start. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 175-200.

Modugno, F., T.R.G. Green and B.A. Myers (1994). Visual Programming in a Visual Domain: A Case Study of Cognitive Dimensions. People and Computers IX: Proceedings of the HCI'94 Conference. G. Cockton, S. W. Draper and G. R. S. Weir. Glasgow, Cambridge University Press: 91-108.

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

Myers, B.A., R.C. Miller, R. McDaniel and A. Ferrency (1996). Easily Adding Animations to Interfaces Using Constraints. Proceedings of ACM SIGGRAPH Symposium on User Interface Software and Technology. Seattle, WA: 119-128.

Nardi, B.A. (1993). A Small Matter of Programming: Perspectives on End User Computing. Cambridge, MA, The MIT Press.

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

Nielsen, J. (1993). Usability Engineering. Chestnut Hill, MA, AP Professional.

Onorato, L.A. and R.W. Schvaneveldt (1986). Programmer/Nonprogrammer Differences in Specifying Procedures to People and Computers. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 128-137.

Pane, J.F. and B.A. Myers (1996). Usability Issues in the Design of Novice Programming Systems. Pittsburgh, PA, Carnegie Mellon University**:** 85.

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

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

Repenning, A. and T. Sumner (1995). "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages." Computer: 17-25.

Sime, M.E., T.R.G. Green and D.J. Guest (1977). "Scope Marking in Computer Conditionals: A Psychological Evaluation." International Journal of Man-Machine Studies 9: 107-118.

Smith, D.C., A. Cypher and K. Schmucker (1996). "Making Programming Easier for Children." interactions 3(5): 59-67.

Soloway, E., S. Jackson, J. Klein, C. Quintana, J. Reed, J. Spitulnik, S. Stratford, S. Studer, J. Eng and N. Scala (1996). Learning Theory in Practice: Case Studies of Learner-Centered Design. Proceedings CHI'96 Conference on Human Factors in Computing Systems. Vancouver, BC, Canada: 189-196.