N88-17225

THE DESKTOP INTERFACE IN INTELLIGENT TUTORING SYSTEMS

Stephen Baudendistel Grace Hua

COMPUTER SCIENCES CORPORATION
Applied Technology Division
16511 Space Center Blvd.
Houston, TX 77058
(713) 280-2430

Abstract

The interface between an Intelligent Tutoring System (ITS) and the person being tutored is critical to the success of the learning process. If the interface to the ITS is confusing or non-supportive of the tutored domain, the effectiveness of the instruction will be diminished or lost entirely. Consequently, the interface to an ITS should be highly integrated with the domain to provide a robust and semantically rich learning environment. In building an ITS for ZetaLISP on a LISP Machine, a Desktop Interface was designed to support a programming learning environment. Using the bitmapped display, windows, and mouse, three desktops were designed to support self-study and tutoring of ZetaLISP. Through organization, well-defined boundaries, and domain support facilities, the desktops provide substantial flexibility and power for the student and facilitate learning ZetaLISP programming while screening the student from the complex LISP Machine environment. The student can concentrate on learning ZetaLISP programming and not on how to operate the interface or a LISP Machine.

Introduction

Artificial Intelligence techniques are now beginning to be applied to the area of education, in particular to the development of Intelligent Computer Assisted Instruction (ICAI). Frequently, the ICAI is in the form of Intelligent Tutoring Systems. Figure 1 depicts a typical ICAI architecture [9]. The area of the ITS most frequently addressed to date has been the student model. By contrast, the interface has been minimally addressed. Yet the interface is the student's contact with every component of the tutor. If the student cannot get past the interface, the quality of the student model or of any other component of the ITS will not matter. Consequently, the interface must be a high priority in the development of any ICAI [17].

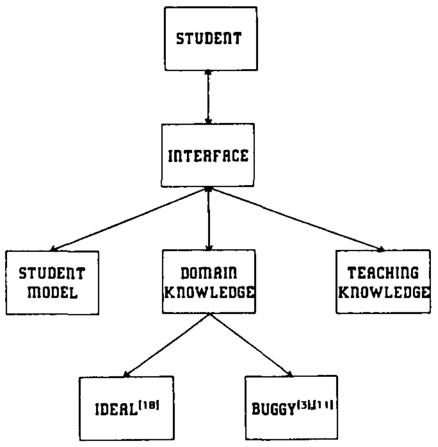


Figure 1. A typical Intelligent Computer Assisted Instruction (ICAI) Architecture.

This paper will describe the implementation of an ICAI interface, referred to as the Desktop Interface, for a ZetaLISP Intelligent Tutoring Assistant (ZITA). To date the ZITA student model has been only minimally implemented, while the emphasis has been on developing an interface which would support and encourage learning to program in ZetaLISP on a LISP Machine. In fact, the Desktop Interface is intended to provide much more than a typical user interface; it is to provide a Programming Learning Environment (PLE) [19]. Moreover, the Desktop Interface is presented as an authoring vehicle for developing programming language tutors for languages in addition to ZetaLISP.

In promoting the ITS interface, we are not advocating a position of ignoring components of ICAI other than the interface or of producing a glittering interface with no underlying substance. Ideally, all the components would be highly integrated. However, up to this point, more attention has been devoted to the more glamorous components: the Student Model and the Domain Knowledge. We do not want the gains made in these latter components diminished or lost because the learning environment does not foster and facilitate learning. Unpleasant experiences with frustrating, difficult interfaces will not advance ICAI, but rather retard it. Our ideal tutoring environment is one which seems invisible to the student but which supports the intuitive operational expectations of the student relative to the domain being tutored.

Background

In the past five years important advances in graphical presentation capability have made possible a new, powerful method of communication. Bitmapped, graphical windows and the mouse have resulted in proven techniques for reliable, high-bandwidth information exchange between people and computers [21] which more closely model human cognitive processes, especially with the use of metaphor and frames [5]. With these capabilities we can move far beyond the limitations imposed by static CRT screens with 25 lines of 80 characters. Previously such capabilities have required expensive, multi-MIPS computers. But the decreasing cost and increasing power of microcomputers now make such capabilities readily available for ICAI. Indeed, we should demand windows and mice, and refuse to consider systems limited to complicated keystroke patterns and displaying a few lines of text.

Criteria for Developing Tutoring Environments

While the tutoring environment must be designed with the specific domain in mind, some general criteria for developing tutoring environments have begun to emerge [24]. Environments should be intuitive, obvious and fun. The use of metaphor, icons, and the mouse should take advantage of student intelligence, experience and resourcefulness. Environments should provide high-bandwidth communication between the student and the tutor. Designers should be motivated by teaching and cognitive knowledge about how experts perform tasks in the subject domain. Environments should isolate key tools for attaining expertise in the domain. Environments should maintain fidelity with the real world (in learning programming, the student should be able to run both examples and problem solutions). Environments should be responsive, permissive, and consistent based on skills students already have rather than forcing them to learn new skills. Finally, all tools should be based on similar interface devices such as menus, mouse clicks, etc.

A ZetaLISP Tutor

We currently have a task with the Artificial Intelligence Section of the Mission Planning and Analysis Division (MPAD) of NASA's Johnson Space Center (JSC) to provide training in AI topics (Common LISP, ZetaLISP, LISP Machines, CLIPS, ART). The ZetaLISP tutor has been developed on an as-time-permits basis to complement our ZetaLISP class. In designing the ZetaLISP tutor, two goals were established. First, we wanted an effective environment for tutoring ZetaLISP on a LISP Machine. Secondly, we wanted to develop a general programming learning environment for computer applications languages. In particular, we wanted a PLE which could be duplicated on workstations and the upcoming, more powerful personal computers.

One must make a number of assumptions when implementing a tutor. Ours were as follows: the student would be a technical professional employed by NASA or its contractors; the student would have the equivalent of 40 hours of Common LISP training and 8 hours of hands-on training in the use of a LISP Machine; the tutor would supplement our classroom ZetaLISP training; the tutor could evolve to be used by persons who had completed the ZetaLISP training (about 45 hours) and were interested in obtaining more experience or were seeking examples to help in their current tasks.

The coaching system of ZITA evaluates the student's performance through a differential modeling technique, comparing the student's progress to an ideal solution step-by-step, intervening immediately when it perceives the student has made a mistake [4], [18]. At this stage of development, the immediate intervention issued by the tutor primarily points out syntactic errors and noise level errors made by the student presumably due to negligence and fatigue. Based on the previous assumption of the student's background, these errors are not considered to have resulted from misconceptions in learning.

Learning to Program and the PLE

How could an appropriately structured environment facilitate the acquisition of programming skills [16]? In order to answer this question, we first investigated some of the aspects of learning to program. Three aspects of learning to program were to be supported by our PLE [1]. First, the PLE was to help the student organize and compile problem-solving operators for programming. Learning to program involves recognizing appropriate goals and decomposing the goals into subgoals until goals are reached which correspond to code. Secondly, the PLE was to represent the relevant knowledge, both declarative and procedural, in ways which correspond to the cognitive representations of programmers, because one's representation of a problem has strong impact on one's problem-solving ability. Thirdly, the PLE was to act as an external memory device for programmers to reduce the impact of human memory limitations. Approximately 50 percent of LISP novices' time is spent recovering from errors of memory [1]. By reducing student working memory load, the PLE will minimize student errors due to memory limitations.

Good programmers are made, not born [23]. B.S. Bloom found that 98 percent of the students with private tutors performed better than the average classroom student. He also found that the greatest learning gains were for the poorest students [2]. The average college graduate is not prepared to perform professional programming tasks without additional training when he or she first arrives on the job in industry. Large sums of money are spent training and retraining programmers with widely varying results. We can improve this process greatly by developing intelligent tutors for learning programming which will provide consistent, cognitively modeled [12] tutoring when and where needed, and at significant cost savings.

The PLE of our ZetaLISP tutor addresses the three aspects of learning programming described above in four ways:

a) Learning by example [20], [10], [4];
b) Facilitating knowledge representation;
c) Reducing student working memory requirements;
d) Unleashing the power of the computer on the ICAI interface.

The PLE is based on learning by example. Examples are critical to learning and to the structure of knowledge and memory. Learning by example provides the student with early, positive experiences and lays down a solid foundation on which to build. Examples help the student organize and compile the use of appropriate operators for programming. Examples illustrate goals and subgoals appropriate to a particular language but which may not transfer to or from other languages. Techniques recalled from examples help reduce the number of steps to produce a solution in similar problems. Novices use examples to generalize solutions, set limits to those generalizations, make recipes for standard tasks, and as a basis for retrieval and modification approach to generating other examples.

Adult students only acquire effective use of problem-solving knowledge by practicing with a series of examples and problems [19]. Adults prefer learning by doing rather than watching because it makes the subject immediately useful and meaningful [22]. Studies by the Xerox Corporation confirm that learning occurs 50 percent faster with active, hands-on training than when the learning is passive [13]. Adult learners seek a focused, applicable treatment of the subject so they can transfer the concept to their work. Generalities are acceptable only when they lead to specific information and ideas. Adults are highly motivated to apply their learning to their work and are willing to assume responsibility for learning. Adult learning uses experience as a resource. Adults feel rewarded when the learning enriches their experience. Material that provides options is more appealing to adults than material that locks in one approach. Examples reinforce and strengthen the link between the concept and application transfer, rewarding the learning experience and disposing the student toward further knowledge.

The PLE facilitates programming knowledge representation as used by the expert. Not only is syntactic knowledge represented, but more importantly, much implicit semantic knowledge, acquired over many years of experience, is presented to the student. Techniques illustrating when, what, and how to extend specific knowledge in the examples to solve new problems (extrapolate) [15] must be taught. Human learning occurs as a search in a problem space [12] and the desktop interface of the PLE helps constrain and focus the search. Each learning state and operators are well defined for each desktop in the PLE. Chunking is well suited to learning because it is a recorder of goal-based experience; it caches the processing of a subgoal in such a way that a chunk can substitute for the normal, possibly complex, processing of the subgoal the next time the same or a similar subgoal is generated [11]. Each exercise is a chunking process of storing both knowledge and links to appropriate, related knowledge.

Memory load is minimized by the PLE. Each desktop of the PLE organizes information by chunking into easily recognized areas, minimizing student memory requirements. Each desktop is self-contained; the information necessary to perform required actions on the desktop is present in a window. Transitions from one desktop to another are accomplished with a simple mouse click on a clearly marked box. By using direct manipulation techniques with the mouse and menus, options are clearly delineated and selected in obvious, foolproof ways. Examples and problems help clearly separate details from general principles and establish limitations when extending operators. Finally, each student can use as much or as little of the instructions and explanations as desired, thus both avoiding information overload and frustration from too little information.

Students fail to learn from ICAI only when there are negative forces set up against learning [23] such as unfriendly, difficult interfaces. By unleashing the power of the computer in creating a seemingly invisible desktop tutorial interface, we provide an ideal programming learning environment. The format of the PLE defines boundaries unobtrusively while leaving the horizons of the domain open for the student to acquire the desired knowledge. Bitmapped windows, the mouse, and high-powered (MIPS, memory, windowing operating systems), low-cost, microprocessor-based computers have made possible high-bandwidth, self-evident ICAI interfaces.

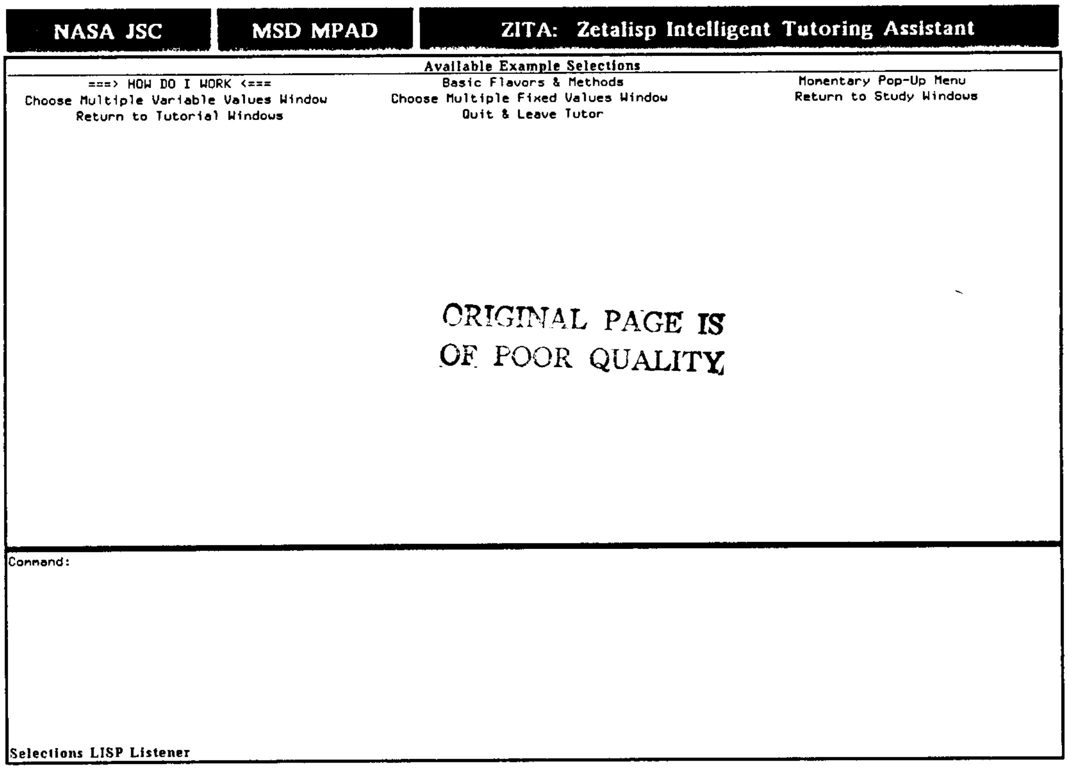
The Desktop Interface Implementation of the PLE

The Desktop Interface implemented for the ZetaLISP PLE resembles a desk with relevant documents spread out neatly on it; because there are several discrete stages in the PLE, there is a separate desktop for each stage. Each desktop is divided into four or five parts (windows) with each part representing one document; if a document cannot be seen completely in its window, the window scrolls (using the mouse) to permit unseen sections to be read. People can deal with from four to seven chunks of data at one time [8]. The division of the desktop into less than seven chunks is designed to fit this cognitive model and thereby to limit the student working memory load. Desktops and windows are consistent in format and function. Each desktop must be self-contained so that the student can concentrate on learning the desired knowledge of the domain and not on operating the interface or searching books for additional information. All options are selected with the mouse. Code for examples and problem solutions can be executed by clicking the mouse on an appropriate menu item. The student can hardcopy the window contents for easier reading, making notes, or for future reference [23].

Four desktops comprise the Desktop Interface for this PLE. The first three have been implemented; the fourth has not been designed. The first desktop is the Selections Desktop (Figure 2). In the Selections Desktop the student selects, with the tutor's assistance (based on past performance), the topic of study by selecting an example topic with the mouse. This desktop also contains a LISP listener where the student can enter and execute LISP code if desired for any reason. When an example topic is agreed on between the tutor and the student, the student is taken to the second desktop, the Study Desktop (Figure 3).

In the Study Desktop, the student is presented with instructions for the desktop, the code for the selected example topic, explanations for the topic, and a LISP listener. Because so much information about programming is conveyed only by executing programs, the student can execute the code for the example being studied by selecting a box with the mouse at the bottom of the LISP listener (Figure 4). When the student has finished studying the example, he or she can work problems posed by the tutor which are variations of the code of the example studied by selecting a box with the mouse at the bottom of the LISP listener. In this case, the student is taken to the third desktop, the Tutorial Desktop. As before, there are instructions for this desktop and the code of the example from the Study Desktop.

In the Tutorial Desktop, the student clicks the mouse on the menu item "Show Variation Choices" and is then presented with a list of available problems. Once the student selects a problem to work, the code of the problem, which is a variation of the example studied, is loaded in a window (code which the student is to supply is missing, from a few lines to whole functions). Guidelines for working the problems appear in reverse video and a reverse video window appears over the example code window for the student to enter the missing code according to the guidelines (Figure 5). The student enters ZetaLISP code and the tutor


NASA JSC | MSD MPAD | ZITA: Zetalisp Intelligent Tutoring Assistant

Available Example Selections

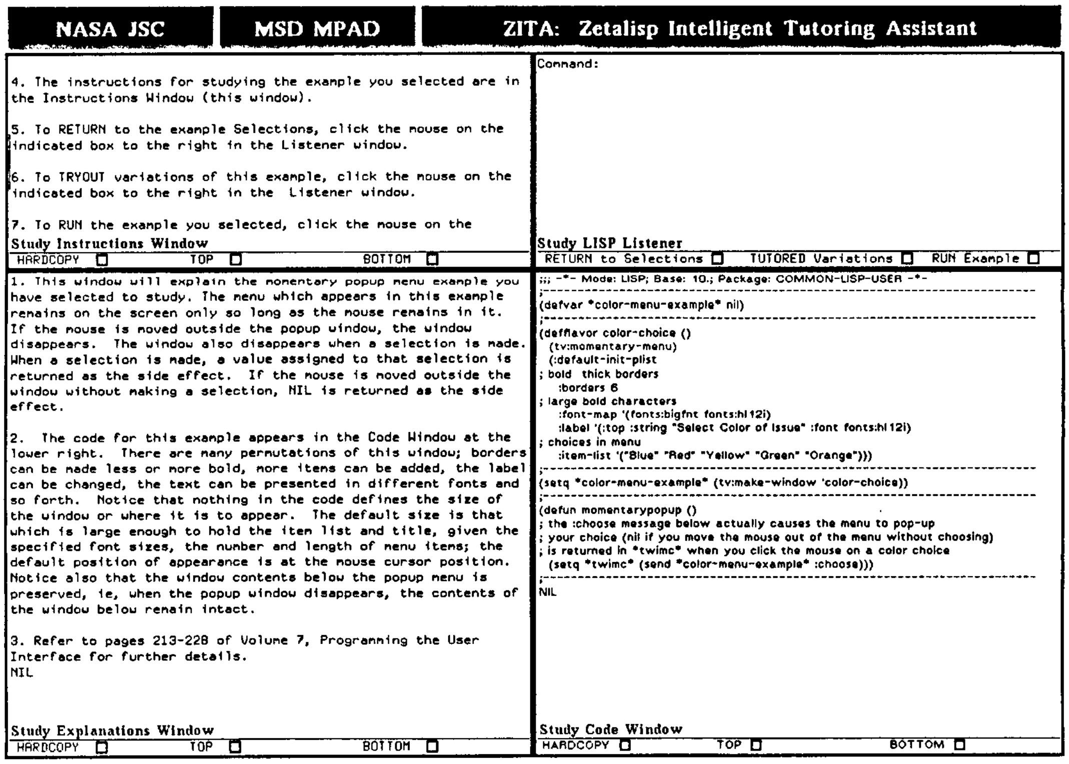
===> HOW DO I WORK <===	Basic Flavors & Methods	Momentary Pop-Up Menu
Choose Multiple Variable Values Window	Choose Multiple Fixed Values Window	Return to Study Windows
Return to Tutorial Windows	Quit & Leave Tutor	

ORIGINAL PAGE IS
OF POOR QUALITY

Command:

Selections LISP Listener

Figure 2. The Selections Desktop of the Desktop Interface.


NASA JSC | MSD MPAD | ZITA: Zetalisp Intelligent Tutoring Assistant

4. The instructions for studying the example you selected are in the Instructions Window (this window).

5. To RETURN to the example Selections, click the mouse on the indicated box to the right in the Listener window.

6. To TRYOUT variations of this example, click the mouse on the indicated box to the right in the Listener window.

7. To RUN the example you selected, click the mouse on the

Study Instructions Window
HARDCOPY ☐ TOP ☐ BOTTOM ☐

Command:

Study LISP Listener
RETURN to Selections ☐ TUTORED Variations ☐ RUN Example ☐

1. This window will explain the momentary popup menu example you have selected to study. The menu which appears in this example remains on the screen only so long as the mouse remains in it. If the mouse is moved outside the popup window, the window disappears. The window also disappears when a selection is made. When a selection is made, a value assigned to that selection is returned as the side effect. If the mouse is moved outside the window without making a selection, NIL is returned as the side effect.

2. The code for this example appears in the Code Window at the lower right. There are many permutations of this window; borders can be made less or more bold, more items can be added, the label can be changed, the text can be presented in different fonts and so forth. Notice that nothing in the code defines the size of the window or where it is to appear. The default size is that which is large enough to hold the item list and title, given the specified font sizes, the number and length of menu items; the default position of appearance is at the mouse cursor position. Notice also that the window contents below the popup menu is preserved, ie, when the popup window disappears, the contents of the window below remain intact.

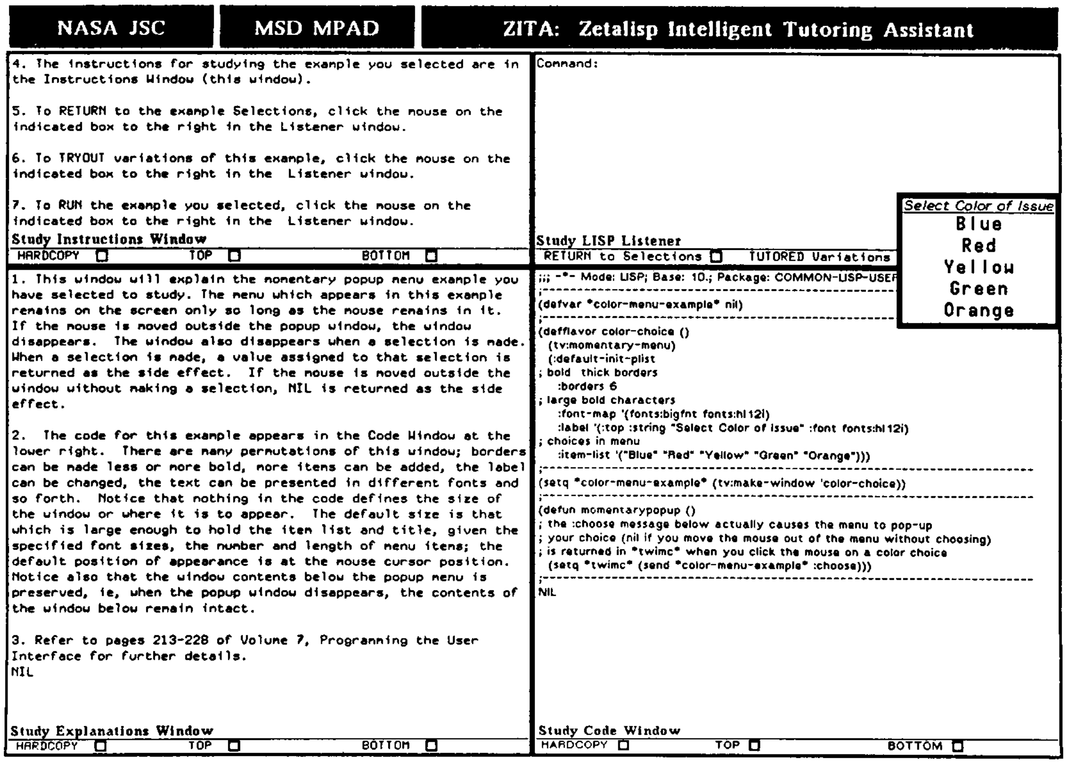
3. Refer to pages 213-228 of Volume 7, Programming the User Interface for further details.
NIL

Study Explanations Window
HARDCOPY ☐ TOP ☐ BOTTOM ☐

```
;;; -*- Mode: LISP; Base: 10.; Package: COMMON-LISP-USER -*-
;------------------------------------------------------------------
(defvar *color-menu-example* nil)
;------------------------------------------------------------------
(defflavor color-choice ()
  (tv:momentary-menu)
  (:default-init-plist
; bold  thick borders
    :borders 6
; large bold characters
    :font-map '(fonts:bigfnt fonts:hl12i)
    :label '(:top :string "Select Color of Issue" :font fonts:hl12i)
; choices in menu
    :item-list '("Blue" "Red" "Yellow" "Green" "Orange")))
;------------------------------------------------------------------
(setq *color-menu-example* (tv:make-window 'color-choice))
;------------------------------------------------------------------
(defun momentarypopup ()
; the :choose message below actually causes the menu to pop-up
; your choice (nil if you move the mouse out of the menu without choosing)
; is returned in *twimc* when you click the mouse on a color choice
  (setq *twimc* (send *color-menu-example* :choose)))
;------------------------------------------------------------------
NIL
```

Study Code Window
HARDCOPY ☐ TOP ☐ BOTTOM ☐

Figure 3. The Study Desktop of the Desktop Interface.



```
4. The instructions for studying the example you selected are in
the Instructions Window (this window).

5. To RETURN to the example Selections, click the mouse on the
indicated box to the right in the Listener window.

6. To TRYOUT variations of this example, click the mouse on the
indicated box to the right in the  Listener window.

7. To RUN the example you selected, click the mouse on the
indicated box to the right in the  Listener window.
```

```
1. This window will explain the momentary popup menu example you
have selected to study. The menu which appears in this example
remains on the screen only so long as the mouse remains in it.
If the mouse is moved outside the popup window, the window
disappears.  The window also disappears when a selection is made.
When a selection is made, a value assigned to that selection is
returned as the side effect.  If the mouse is moved outside the
window without making a selection, NIL is returned as the side
effect.

2.  The code for this example appears in the Code Window at the
lower right.  There are many permutations of this window; borders
can be made less or more bold, more items can be added, the label
can be changed, the text can be presented in different fonts and
so forth.  Notice that nothing in the code defines the size of
the window or where it is to appear.  The default size is that
which is large enough to hold the item list and title, given the
specified font sizes, the number and length of menu items; the
default position of appearance is at the mouse cursor position.
Notice also that the window contents below the popup menu is
preserved, ie, when the popup window disappears, the contents of
the window below remain intact.

3. Refer to pages 213-228 of Volume 7, Programming the User
Interface for further details.
NIL
```

```
;;; -*- Mode: LISP; Base: 10.; Package: COMMON-LISP-USER
;-------------------------------------------------
(defvar *color-menu-example* nil)
;-------------------------------------------------
(defflavor color-choice ()
  (tv:momentary-menu)
  (:default-init-plist
; bold  thick borders
    :borders 6
; large bold characters
    :font-map '(fonts:bigfnt fonts:hl12i)
    :label '(:top :string "Select Color of Issue" :font fonts:hl12i)
; choices in menu
    :item-list '("Blue" "Red" "Yellow" "Green" "Orange")))
;----------------------------------------------------------------------
(setq *color-menu-example* (tv:make-window 'color-choice))
;----------------------------------------------------------------------
(defun momentarypopup ()
; the :choose message below actually causes the menu to pop-up
; your choice (nil if you move the mouse out of the menu without choosing)
; is returned in *twimc* when you click the mouse on a color choice
  (setq *twimc* (send *color-menu-example* :choose)))
;----------------------------------------------------------------------
NIL
```

Figure 4. Student executing code for the example being studied on the Study Desktop.

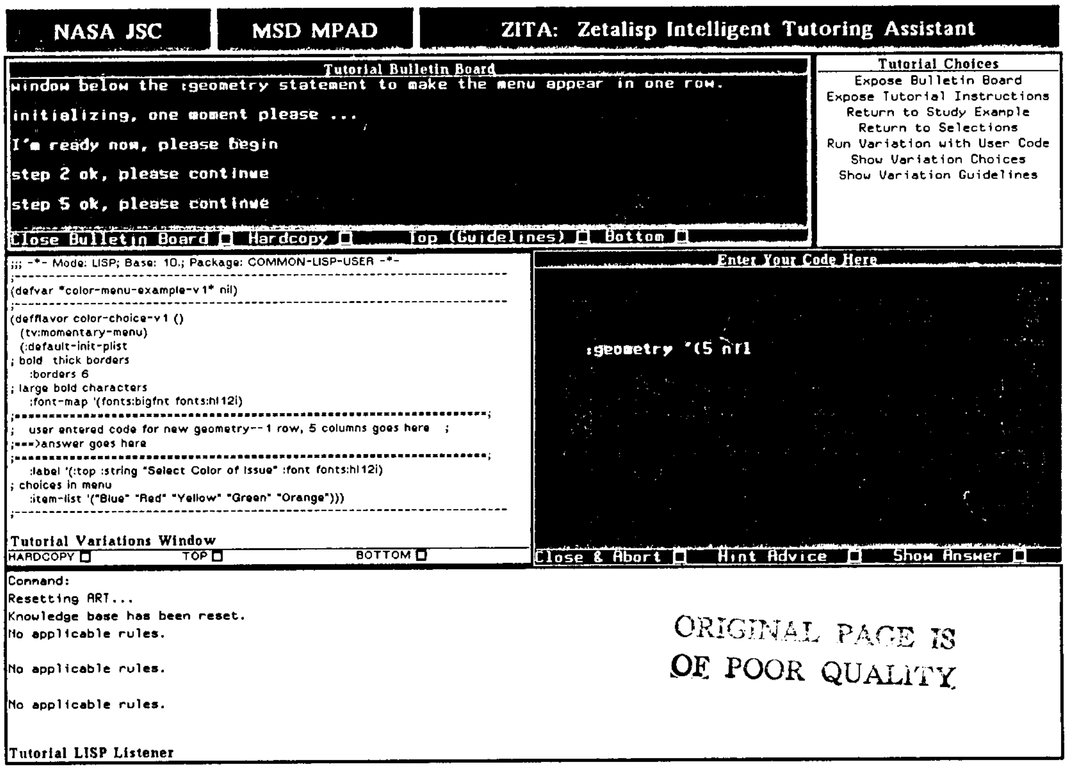
attempts to diagnose bugs and offer corrective dialogue. When the student successfully completes the problem, the tutor inserts the code into the variation code window and the student can execute the problem solution (Figure 6) by clicking the mouse on the menu item "Run Variation with User Code". The student may then select another problem on the current topic or return to the Selections Desktop to choose another topic.

The fourth desktop, the Planning and Goals Help Desktop, has not been implemented yet. Because successful programming requires knowledge of how to both recognize recurring operations and make goals and plans to perform those operations, unsuccessful programmers will exhibit a lack of such abilities. Consequently, the tutor will have to help not only with syntax but also with establishing programming goals and plans. Overcoming this inability is critical if the student is to learn programming [18], [14], [6], [7]. Thus, when the student demonstrates an inability to form correct programming goals and plans, he or she will be transferred to this desktop and will be assisted by the tutor in devising successful goals and plans for the selected problem before being returned to the Tutorial Desktop. Once back in the Tutorial Desktop, the tutor will assist the student in writing code based on the goals and plans developed in the Planning and Goals Help Desktop.

Expectations for an ICAI PLE

We expect the PLE to satisfy a number of sound cognitive principles. The actual layout of the PLE is not important so long as the underlying structure makes the semantics of the domain evident, that is, makes it easy to carry out actions in the domain, and to see and understand the results and implications of those actions. It must support students as they acquire an understanding of the complex semantic domain of programming, minimizing the gap between expectations and actions supported. Certainly it is specialized, highly integrated with the domain and semantically rich with high-bandwidth information transfer between interface and student. It avoids low-bandwidth, semantically weak interfaces which greatly complicate the diagnosis problem. By offering a good match to goals and plans of the student as they

ORIGINAL PAGE IS
OF POOR QUALITY



```
window below the :geometry statement to make the menu appear in one row.

initializing, one moment please ...

I'm ready now, please begin

step 2 ok, please continue

step 5 ok, please continue
```

```
;;; -*- Mode: LISP; Base: 10.; Package: COMMON-LISP-USER -*-
;-----------------------------------------------------------------------
(defvar *color-menu-example-v1* nil)
;-----------------------------------------------------------------------
(defflavor color-choice-v1 ()
  (tv:momentary-menu)
  (:default-init-plist
; bold  thick borders
   :borders 6
; large bold characters
   :font-map '(fonts:bigfnt fonts:hl12i)
;=====================================================================;
;  user entered code for new geometry--1 row, 5 columns goes here   ;
;===>answer goes here
;=====================================================================;
   :label '(:top :string "Select Color of Issue" :font fonts:hl12i)
; choices in menu
   :item-list '("Blue" "Red" "Yellow" "Green" "Orange")))
;-----------------------------------------------------------------------
```

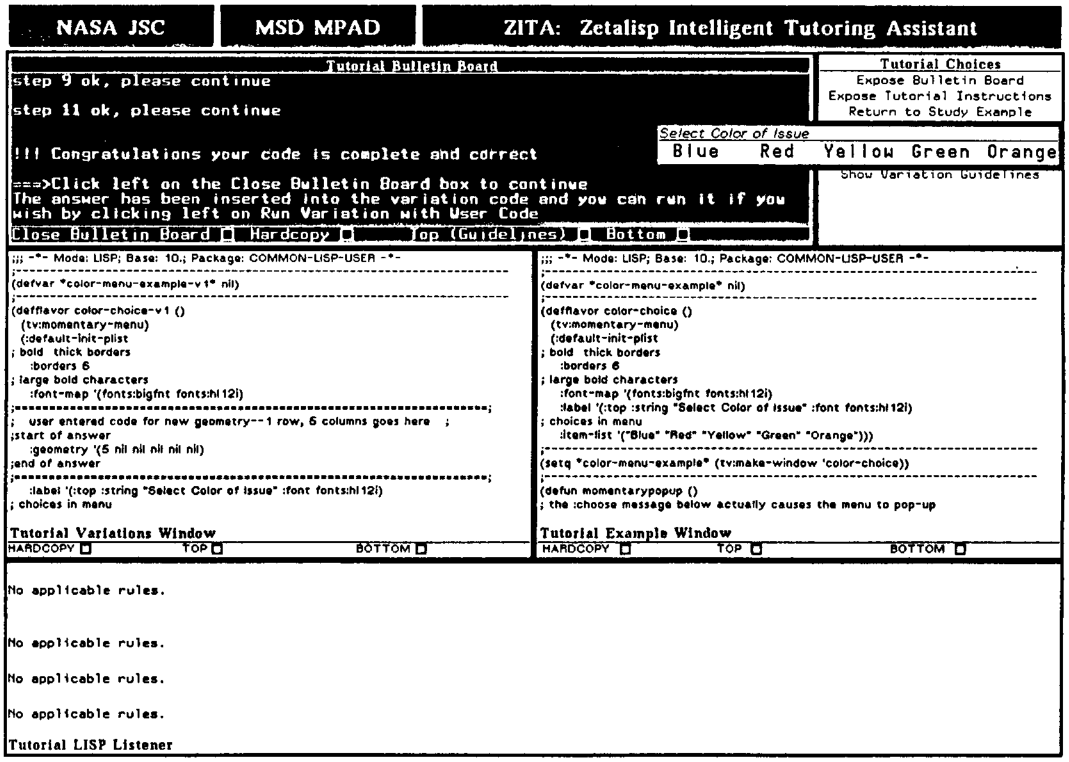
```
:geometry '(5 nil
```

```
Command:
Resetting ART...
Knowledge base has been reset.
No applicable rules.

No applicable rules.

No applicable rules.
```

Figure 5. Student entering code to solve the problem posed by the ZetaLISP tutor on the Tutorial Desktop.



```
step 9 ok, please continue

step 11 ok, please continue

!!! Congratulations your code is complete and correct

===>Click left on the Close Bulletin Board box to continue
The answer has been inserted into the variation code and you can run it if you
wish by clicking left on Run Variation with User Code
```

```
;;; -*- Mode: LISP; Base: 10.; Package: COMMON-LISP-USER -*-
;-----------------------------------------------------------------------
(defvar *color-menu-example-v1* nil)
;-----------------------------------------------------------------------
(defflavor color-choice-v1 ()
  (tv:momentary-menu)
  (:default-init-plist
; bold  thick borders
   :borders 6
; large bold characters
   :font-map '(fonts:bigfnt fonts:hl12i)
;=====================================================================;
;  user entered code for new geometry--1 row, 5 columns goes here   ;
;start of answer
   :geometry '(5 nil nil nil nil nil)
;end of answer
;=====================================================================;
   :label '(:top :string "Select Color of Issue" :font fonts:hl12i)
; choices in menu
```

```
;;; -*- Mode: LISP; Base: 10.; Package: COMMON-LISP-USER -*-
;-----------------------------------------------------------------------
(defvar *color-menu-example* nil)
;-----------------------------------------------------------------------
(defflavor color-choice ()
  (tv:momentary-menu)
  (:default-init-plist
; bold  thick borders
   :borders 6
; large bold characters
   :font-map '(fonts:bigfnt fonts:hl12i)
   :label '(:top :string "Select Color of Issue" :font fonts:hl12i)
; choices in menu
   :item-list '("Blue" "Red" "Yellow" "Green" "Orange")))
;-----------------------------------------------------------------------
(setq *color-menu-example* (tv:make-window 'color-choice))
;-----------------------------------------------------------------------
(defun momentarypopup ()
; the :choose message below actually causes the menu to pop-up
```

```
No applicable rules.


No applicable rules.

No applicable rules.

No applicable rules.
```

Figure 6. Student executing code for their solution to the problem posed by the ZetaLISP tutor.

learn to program, it accommodates stages of student conceptualization of the domain and how movement from one stage to another takes place. It reflects the task of learning programming, the information that must be presented, and ways in which students may interact with the information, that is, how good programmers organize knowledge and use operators. Serving as an external memory system, the PLE uses the desktop metaphor to organize, standardize, define boundaries, reduce memory requirements, obviate actions/results, and convey a feeling of control.

Conclusions

We now need, and will continue to need, many well-trained programmers. The current method of training programmers is expensive, haphazard, and not founded on an understanding of how to learn programming. Over the past five years we have obtained much knowledge of how to learn programming and, at the same time, computers and software have advanced dramatically in capability while their cost has declined substantially. At this point we have the knowledge and tools available to develop an ICAI Programming Learning Environment and deliver uniform, semantically rich, and cognitively based tutors to train the necessary programmers. The Desktop Interface is a candidate authoring vehicle for such an ICAI PLE. We are continuing, as time permits, to develop and test the Desktop Interface and the Student Model in the ZetaLISP tutor.

References

1. Anderson, J., "Learning to Program," *Proceedings Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, August 8-12, 1983, pp. 57-62.

2. Anderson, J., Boyle, F., Yost, G., "The Geometry Tutor," *Proceedings Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA., Aug 17-23, 1985, pp. 1-7.

3. Burton, R., "Diagnosing bugs in a simple procedural skill," in Intelligent Tutoring Systems, D. Sleeman and J.S. Brown, eds., Academic Press, New York, NY, 1982.

4. Burton, R., Brown, J., "An investigation of computer coaching for informal learning activities" in Intelligent Tutoring Systems, D. Sleeman and J.S. Brown, eds., Academic Press, New York, NY, 1982.

5. Dear, B., "AI and the Authoring Process," *IEEE Expert*, Summer 1987, pp. 17-24.

6. Farrell, R., Anderson, J., Reiser, B., "An Interactive Computer-based Tutor for LISP," *Proceedings Third National Conference on Artificial Intelligence*, Austin, Tx, Aug 6-10, 1984, pp. 106-109.

7. Genesereth, M., "The role of plans in intelligent teaching systems," in Intelligent Tutoring Systems, D. Sleeman and J.S. Brown, eds., Academic Press, New York, NY, 1982.

8. Harmon, P., King, D., *Expert Systems, Artificial Intelligence in Business*, John Wiley & Sons, New York, NY, 1985.

9. Kearsley, G., ed., *Artificial Intelligence & Instruction*, Addison-Wesley, Reading, Mass., 1987.

10. Kolodner, J., Simpson, R. Jr., Sycara-Cyranski, K., "A Process Model of Cased-Based Reasoning in Problem Solving," *Proceedings Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA., Aug 17-23, 1985, pp. 284-290.

11. Laird, J., Rosenbloom, P., Newell, A., "Towards Chunking as a General Learning Mechanism," *Proceedings Third National Conference on Artificial Intelligence*, Austin, Tx, Aug 6-10, 1984, pp. 188-192.

12. Langley, P., Ohlsson, S., "Automated Cognitive Modeling," *Proceedings Third National Conference on Artificial Intelligence*, Austin, Tx, Aug 6-10, 1984, pp. 193-197.

13. Lichtman, D., Watt, P., "Bottom Line Training - Getting Results, Not Classes," *Manage*, Third Quarter 1986, pp. 22-23, 35.

14. Littman, D., Pinto, J., Soloway, E., "An Analysis of Tutorial Reasoning About Programming Bugs," *Proceedings Fifth National Conference on Artificial Intelligence*, Philadelphia, Pa., August 11-15, 1986, pp. 320-326.

15. Matz, M., "Towards a process model for high school algebra errors," in Intelligent Tutoring Systems, D. Sleeman and J.S. Brown, eds., Academic Press, New York, NY, 1982.

16. Miller, M., "A Structured Planning and Debugging Environment for Elementary Programming," in Intelligent Tutoring Systems, D. Sleeman and J.S. Brown, eds., Academic Press, New York, NY, 1982.

17. Miller, J., "Human-computer Interaction and Intelligent tutoring systems," MCC Technical Report Number HI-294-86, April 1987.

18. Orlikowski, W., Vasant, D., "Imposing Structure on Linear Programming Problems: An Empirical Analysis of Expert and Novice Models," in *Proceedings Fifth National Conference on Artificial Intelligence*, Philadelphia, Pa., August 11-15, 1986, pp. 308-312.

19. Reiser, B., Anderson, J., Farrell, R., "Dynamic Student Modelling in an Intelligent Tutor for LISP Programming," *Proceedings Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA., Aug 17-23, 1985, pp. 8-14.

20. Rissland, E., Valcarce, E., Ashley, K., "Explaining and Arguing with Examples," *Proceedings Third National Conference on Artificial Intelligence*, Austin, Tx, Aug 6-10, 1984, pp.288-294.

21. Shneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading, Mass., 1987.

22. Von der Embse, T., "Course Leadership," *Manage*, Volume 39, Number 2, July 1987, pp. 7-8, 33.

23. Weinberg, G., *The Psychology of Computer Programming*, Van Nostrand Reinhold Company, New York, NY, 1971.

24. Woolf, B. and Cunningham, P., "Multiple Knowledge Sources in Intelligent Teaching Systems," *IEEE Expert*, Summer 1987, pp. 41-54.