Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?

E. M. Wilcox*†, J. W. Atwood*, M. M. Burnett*, J. J. Cadiz*, C. R. Cook*

* Dept. of Computer Science
Oregon State University
Corvallis, OR 97331, USA
{atwoodj, burnett, cadiz, cook}@research.cs.orst.edu

† Timberline Software
9600 S.W. Nimbus Ave.
Beaverton, OR 97008, USA
ericw@timberline.com

ABSTRACT
Continuous visual feedback is becoming a common feature in direct-manipulation programming systems of all kinds—from demonstrational macro builders to spreadsheet packages to visual programming languages featuring direct manipulation. But does continuous visual feedback actually help in the domain of programming? There has been little investigation of this question, and what evidence there is from related domains points in conflicting directions. To advance what is known about this issue, we conducted an empirical study to determine whether the inclusion of continuous visual feedback into a direct-manipulation programming system helps with one particular task: debugging. Our results were that although continuous visual feedback did not significantly help with debugging in general, it did significantly help with debugging in some circumstances. Our results also indicate three factors that may help determine those circumstances.

Keywords
Direct manipulation, debugging, end-user programming, spreadsheets, visual programming languages, liveness, empirical study

INTRODUCTION
Shneiderman describes three principles of direct manipulation systems [16] (italics added for emphasis):

1. *continuous representation* of the objects of interest;
2. physical actions or presses of labeled buttons instead of complex syntax; and
3. rapid incremental reversible operations *whose effect on the object of interest is immediately visible.*

Many systems today use visual or demonstrational mechanisms with continuous visual feedback to support forms of programming based upon these principles. Examples include spreadsheets, "watch what I do" macro recording systems, and visual programming languages aimed at audiences ranging from experienced programmers to novice programmers to end users. In devising such systems, many researchers and developers have expended great effort to support the features shown in italics above. However, a rather fundamental question has yet to be answered: do these features actually help in the domain of programming?

To shed some light on this question, we chose to study the effects of continuous visual feedback in direct-manipulation programming systems on one task: debugging. In this paper, we briefly describe our study and present the key results. A more complete description is given in [20].

We chose the task of debugging for two reasons. First, although there has been a fair amount of study of debugging in traditional programming languages, it has been largely overlooked in empirical studies of direct-manipulation programming systems. Second, debugging is a problem-solving activity, and there is evidence both for and against the use of continuous visual feedback in problem-solving activities.

But does debugging ever arise in direct-manipulation programming systems? Some designers of these systems believe it does: for example, several recent papers about demonstrational languages discuss how after-the-fact debugging is done in such languages [4]. In fact, in the subgroup of these languages that infer programs from users' actions, there is added potential for bugs, because a user may not recognize when an inference is close but not entirely correct. In the way of empirical evidence about the importance of debugging in direct-manipulation programming systems, studies of the most widely-used type of such programming systems—spreadsheets—have found that users devote considerable time to debugging [13] and that after the users believe they are finished, bugs often remain [2]. (This work is particularly relevant to the research questions addressed in this paper, because spreadsheets' default behavior is to provide continuous visual feedback via the automatic recalculation feature.) In an analysis of debugging in HyperCard, Eisenstadt reports debugging in HyperCard with HyperTalk scripts to be "disproportionately difficult" compared with the ease he attributes to programming in that environment [5]. Thus it is clear that debugging does indeed arise in direct-manipulation programming, and it may be just as recalcitrant in those systems as it has historically been in the realm of traditional programming.

BACKGROUND AND RELATED WORK

Liveness Levels
The term "continuous visual feedback" is somewhat ambiguous when applied to the domain of direct-manipulation

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee
CHI 97, Atlanta GA USA
Copyright 1997 ACM 0-89791-802-9/97/03 ..$3.50

programming. To address this ambiguity, Tanimoto introduced the term "liveness" to focus on the immediacy of *semantic* feedback that is automatically provided [18].

Tanimoto described four levels of liveness; level 3 is the one of interest here. At level 1 no semantic feedback is provided to the user, and at level 2 the user can obtain semantic feedback, but it is not provided automatically. At level 3, incremental semantic feedback is automatically provided whenever the user performs an incremental program edit, and all affected on-screen values are automatically redisplayed. For example, the automatic recalculation feature of spreadsheets supports level 3 liveness. At level 4, the system responds to program edits as in level 3, and to other events as well such as system clock ticks and incoming data from sensors. Level 3 (and 4, which includes the features of 3) reflects the way continuous visual feedback has been applied to the semantics of programming in direct-manipulation programming systems, and is the kind of continuous visual feedback our study investigated. In this paper, we will use the term "live" to describe systems at level 3 or 4.

Related Work

We have been unable to locate any studies of liveness's effects on either direct-manipulation programming systems or on debugging. However, there are results from related domains that seem to make assorted—and often conflicting—predictions about what can be expected about the effects of liveness on debugging in such programming systems.

The viewpoint that liveness should help in debugging seems to be supported by work such as Green's and Petre's research into cognitive aspects of programming languages [8] and Gugerty's and Olson's studies of debugging behavior [9]. They report that novices and experts alike tend to rely strongly on what Green and Petre term "progressive evaluation" (ability to execute bits and pieces of programs immediately, even if program development is not yet complete). According to this body of work, novices require this feature to find bugs, and experts, who can debug without it if necessary, rely on it even more heavily than novices when it is available.

Empirical studies into debugging in traditional languages have considered the effects of a number of factors on debugging performance (e.g., [6, 9, 12, 15]), but liveness has not been one of these factors. This is also true of studies on end-user programming: the few studies of debugging in end-user programming systems (described in the introduction to this paper) have not investigated liveness as a factor. Most studies of visual programming languages for programmers have concentrated on whether graphics are better than text. These studies are surveyed in [19]. (The upshot of these studies is that for some programming situations graphics are better than text, and for other programming situations text is better than graphics.) None of these studies provide any information about the effects of liveness.

However, in studying the graphics versus text question in the context of algorithm animation systems, Lawrence, Badre, and Stasko gleaned information that is relevant to the liveness question. They learned that, whereas passively observing animations did not significantly increase students' understanding of algorithms, active involvement in the creation of input values to the animations did significantly increase understanding [10]. In light of the several studies showing that programmers in traditional languages perform significantly more interactions when there are shorter response times [16], these findings about algorithm animation may predict that liveness will help in debugging because of the active involvement quick feedback promotes in programmers.

On the other hand, there are studies of problem-solving behavior that point to just the opposite conclusion. One example is the work of Martin and Corl, who contributed to a long line of response time studies (see [16] for a survey of that literature). While they, like their predecessors, found that faster response time improved productivity for simple data entry tasks, contrary to their predecessors they found that the effect declined for more complex data entry tasks and disappeared entirely for problem-solving tasks [11]. Consistent with this are studies by Gilmore and by Svendsen, which showed that too much feedback and responsiveness can interfere with people's problem-solving ability in solving puzzles [7, 17], a task with much in common with debugging. O'Hara's findings are similar; in three experiments that varied the (time) cost of performing certain operations, he found that high costs encouraged people to engage in more advance planning, resulting in more efficient solutions in terms of number of operations [14]. These works would seem to predict that liveness will actually hinder debugging.

THE EXPERIMENT

In our study, we investigated the effects of liveness on three aspects of debugging: debugging accuracy, debugging behavior, and debugging speed. We conducted the experiment in a lab with students seated one per workstation, using the visual programming language Forms/3 [1, 3]. First, the lecturer led a tutorial of Forms/3 in which each student actively participated by working with the example programs on individual workstations. Following the tutorial, the students were given two Forms/3 programs to debug, each with a 15-minute time limit: one that displays a graphical LED image of a numeric digit, and one that verifies a password mathematically in order to determine whether to unlock access to some hypothetical resource. All subjects debugged one of the programs using the normal Forms/3 system, which is live, and the other using the same system but with immediate feedback removed. Before carrying out the main experiment, we conducted a pilot study with four participants to test the experiment's design.

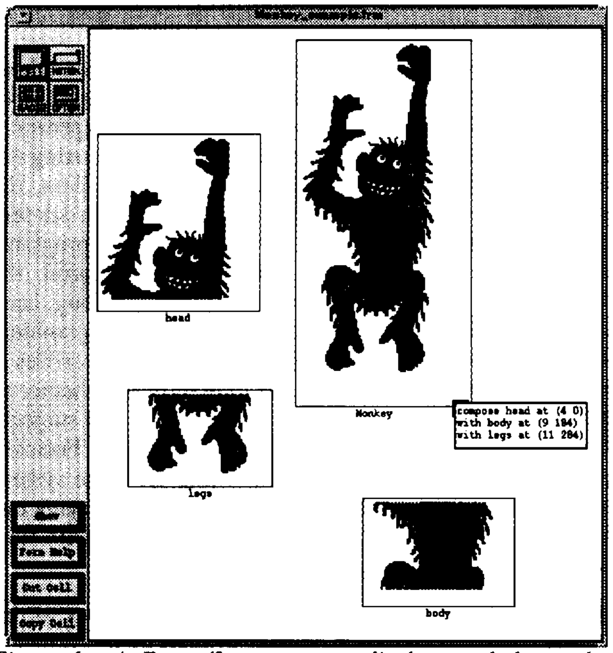
The experiment was counterbalanced with respect to the liveness factor. Thus, each subject worked both problems, but half the subjects debugged the LED problem live and the Lock problem non-live, while the other half did the opposite. The assignments into these two groups were made randomly. The LED problem was always first, giving the Lock problem a learning advantage. (Since there was no assumption that the two problems were of equal difficulty, this did not affect the validity of the results.) The data collected during the experiment were post-problem and post-test questionnaires answered by the subjects, their on-line activities collected by an electronic transcripting mechanism, and their final debugged programs.

The Programming Environment

Specifying a program in Forms/3 is similar to specifying a program in a spreadsheet: The user creates cells and gives each cell a formula, and as soon as the user enters a formula, it is immediately evaluated and the result displayed. See Figure 1 for a sample Forms/3 program.

Forms/3 supports direct manipulation in several ways, among which are the ability to point at cells instead of typing a textual reference, and the ability to specify some kinds of formulas by demonstration. However, the most important attribute of Forms/3 with respect to this study is its automatic and immediate feedback about the effects of changes, i.e. its support of liveness. In languages supporting liveness, there is no compile phase, and no need to query variables in order to see values. Thus, in Forms/3, changing a cell's formula causes its value, as well as the values of any cells dependent on the change, to be recalculated and displayed immediately and automatically.

Since Forms/3 is a live system, it was necessary to produce a second version for this experiment (which we will call the non-live version), in which the automatic computation facilities were replaced by a button labeled *compile*, which would be used to submit the program to the system for execution. Each such submission took 90 seconds,



```
compose head at (4 0)
with body at (9 184)
with legs at (11 284)
```

Figure 1: A Forms/3 program to display a whole monkey, given a collection of clip-art body parts. This program was used in the experiment's tutorial to demonstrate "compose" formulas like the one for cell Monkey. *This formula could be either typed or demonstrated by arranging the* head, body, *and* legs *cells as desired and rubberbanding the arrangement.*

simulating the time not only to compile, but also to set up and run a test as would be required in a non-live system. Upon completion, all the values would be displayed and would remain on display throughout the subject's explorations until the subject actually made another formula change, at which point all the values were erased (since their values could no longer be determined without re-execution).

The Subjects

The subjects were students enrolled in a senior level operating systems course. All subjects had experience programming in C or C++, 86% had used spreadsheets, and 55% had programmed in LISP. One student had seen Forms/3 before, but had not actually programmed in it. As Table 1 shows, there was very little background difference between the two groups. Although 12 subjects claimed professional experience, these were simply internships of one year or less for all but two subjects, one in each group. These 12 subjects' small amount of professional experience did not provide a performance advantage; there was actually a small negative correlation (-0.11) between the presence of professional experience and debugging accuracy.

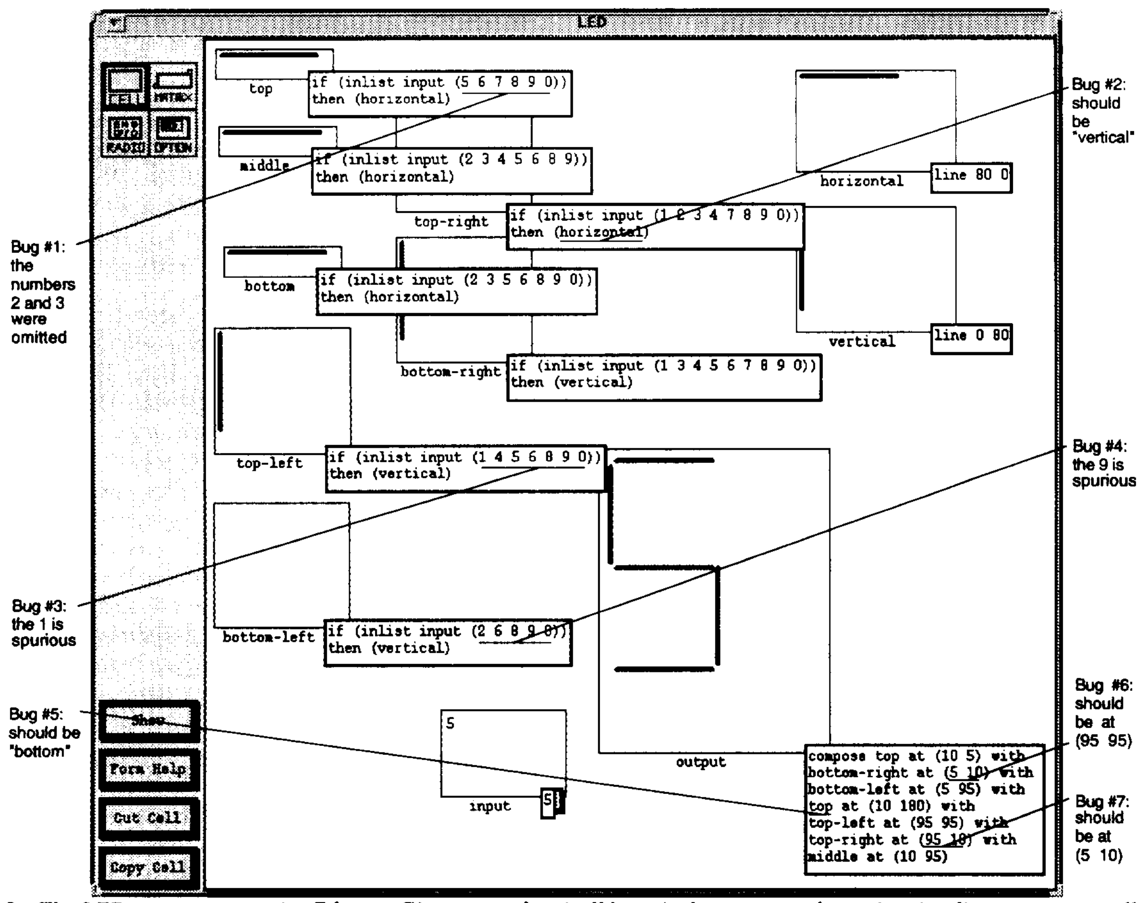
The Programs

To avoid limiting our study to exclusively graphically-oriented programs or exclusively mathematically-oriented programs, we chose one of each kind for the subjects to debug. The LED program was the first program the subjects debugged. This program produces graphical output similar to LED (light-emitting diode) displays on digital clocks. See Figure 2. Cell *output* is a composition of the lines needed to draw the digit in cell *input*. A useful aspect of this kind of program is that many subjects' prior experience with digital clocks helps them recognize the right answer when they see it.

The Lock program shown in Figure 3 was designed to emphasize mathematics instead of graphics. The basic premise was that the lock could only be unlocked with a two-step security mechanism. To gain access to such a lock, a person might insert an ID card containing his/her social security number and then key in two combinations. If the combinations were valid for that social security

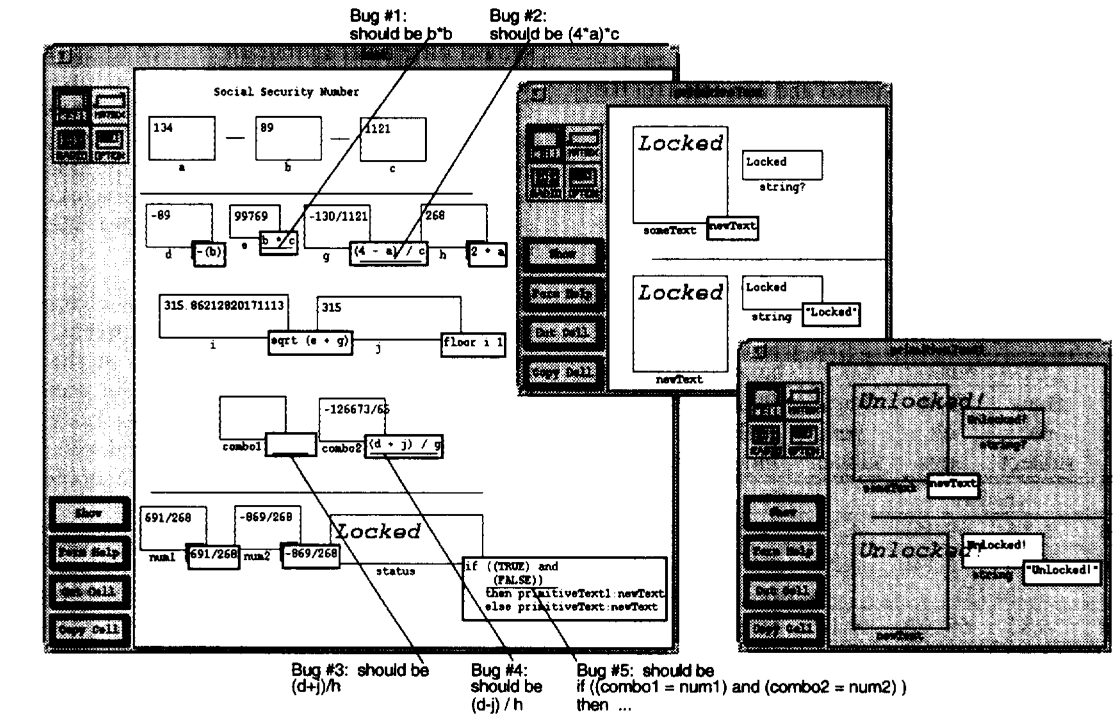
	Cum. GPA	CS 411 grade	Number of CS courses taken	Number of prog. langs. known	Subjects with prof. experience
	mean	counts	mean	mean	count
Group 1 (n=14) (Live first)	3.18	6 As 7 Bs 1 C	9.85	5.00	4
Group 2 (n=15) (Live second)	3.30	5 As 10 Bs	9.36	4.40	8

Table 1: Summary of the 29 subjects' backgrounds.


LED
top
middle
top-right
bottom
bottom-right
top-left
bottom-left
horizontal
vertical
output
input
5
Show
Form Help
Cut Cell
Copy Cell
Bug #1: the numbers 2 and 3 were omitted
Bug #2: should be "vertical"
Bug #3: the 1 is spurious
Bug #4: the 9 is spurious
Bug #5: should be "bottom"
Bug #6: should be at (95 95)
Bug #7: should be at (5 10)

```
if (inlist input (5 6 7 8 9 0))
then (horizontal)

if (inlist input (2 3 4 5 6 8 9))
then (horizontal)

if (inlist input (1 2 3 4 7 8 9 0))
then (horizontal)

if (inlist input (2 3 5 6 8 9 0))
then (horizontal)

if (inlist input (1 3 4 5 6 7 8 9 0))
then (vertical)

if (inlist input (1 4 5 6 8 9 0))
then (vertical)

if (inlist input (2 6 8 9 0))
then (vertical)

line 80 0

line 0 80

compose top at (10 5) with
bottom-right at (5 10) with
bottom-left at (5 95) with
top at (10 180) with
top-left at (95 95) with
top-right at (95 10) with
middle at (10 95)
```

Figure 2: The LED program contains 7 bugs. Given a number (cell input*), the program draws it using line segments (cell* output*).*



```
134 — 89 — 1121
-89        -(b)
99769      b * c
-130/1121  (4 - a) / c
268        2 * a
315.86212820171113   sqrt (e + g)
315                  floor i 1
-126673/65           (d + j) / g
691/268    691/268
-869/268   -869/268
if ((TRUE) and
   (FALSE))
then primitiveText1:newText
else primitiveText:newText
Locked     newText
Locked     "Locked"
Unlocked!  newText
Unlocked!  "Unlocked!"
```

Figure 3: The Lock program contains 5 bugs. The subjects were told that the inputs num1 *and* num2 *were correct for the given social security number. When the bugs are removed, the values of cells* num1 *and* num2 *need to match cells* combo1 *and* combo2 *respectively to unlock the lock.*

number, the lock would unlock. The Lock program used a formula similar to the quadratic formula to compute the two valid combinations from the social security number (in cells *a*, *b*, and *c*), and match them with the keyed-in input (cells *num1* and *num2*). The formula used was (-b±floor(sqrt(b^2+4ac)))/2a. Its differences from the true quadratic formula prevented complex roots and simplified the output formats. These points were explained to the subjects before the problem began.

If we had put the entire formula into a single cell, some subjects might have simply rewritten the formula rather than trying to find the bugs. (In fact, such behavior was observed during the pilot with a programming problem that we decided to replace for that reason.) To avoid this problem, we broke the formula's computation into parts using several intermediate cells, each with its own small formula.

RESULTS

We present the results of each of our three research questions separately.

Results: Does liveness help debugging accuracy?

To investigate accuracy, we examined the subjects' final programs to determine their success at identifying and correcting bugs. In this paper, we use the term "identified bug" to mean one of the 12 planted bugs (7 in the LED problem and 5 in the Lock problem) for which the subject has changed the formula (but not necessarily correctly) and the term "corrected bug" to mean an identified bug that the subject succeeded at fixing. Recall the 15-minute limit for each problem; thus the accuracy figures show debugging accuracy the subjects could obtain within 15 minutes. This time limit was adequate for bringing out accuracy differences, since 3 subjects identified and corrected all the bugs in both problems, and all but 2 subjects were able to correct at least one bug in at least one problem.

The rightmost column of Table 2 summarizes the overall accuracy differences. The subjects identified slightly more bugs under the live version and corrected almost the same number of bugs in both systems. But of the bugs they actually identified, the subjects corrected slightly fewer in

	LED (7 bugs)		Lock (5 bugs)		Total
	Bugs per subject	% of all bugs	Bugs per subject	% of all bugs	% of all bugs
Live	(n=14)		(n=15)		(n=29)
Identified	4.86	69.4%	4.73	94.7%	80.4%
Corrected	3.86	55.1%	3.53	70.7%	61.3%
Non-live	(n=15)		(n=14)		(n=29)
Identified	4.87	69.5%	4.14	82.9%	74.9%
Corrected	4.27	60.9%	2.93	58.6%	60.0%

Table 2: Bugs identified and corrected by subjects working live versus non-live. Variable n identifies the number of subjects. Since the LED and Lock problems had differing numbers of seeded bugs, LED versus Lock performances were compared only via the percentage columns.

the live version: 77% live versus 80% non-live (not shown in this table). None of these slight differences in accuracy were statistically significant.

However, as Table 2 shows, the overall totals were comprised of opposite effects of liveness in the LED problem as in the Lock problem. Subjects corrected slightly fewer bugs in the live version of the LED problem, but significantly more in the live version of the Lock problem (Wilcoxon Rank Sum, $p<0.05$).

Looking to within-subject differences, a natural question to ask is whether individual subjects who had difficulties in one version did better in the other. For this and other summary comparisons, we categorized our subjects' accuracy performance as "high performing" (correcting more than half the bugs in a problem) versus "low performing". Table 3 summarizes how the subjects' performances varied between the live and non-live versions. A disproportionate number of subjects (15 of the 29) were high performers in both versions. However, for the 14 subjects who performed poorly in at least 1 version, the live and non-live versions elicited significant performance differences: only 2 of the 14 (14.3%) were able to perform higher in the non-live version than in the live version (χ^2=7.14, df=1, $p<0.01$).

Results: Does liveness change debugging behavior?

There were many differences between the subjects' behavior making changes live versus non-live. A change was defined to be each time a subject opened a cell for editing (other than an input cell such as LED's cell *input*) and accepted his/her changes, and the time spent on the change started when the cell was opened for editing and ended when the subject hit the *accept* button. (Since there are easier ways to view formulas than via edit windows in Forms/3, opening such a window is a bona fide expression of intent to edit in that system.) As Table 4's "Number of changes" columns show, on each problem and in total, the subjects made significantly more changes when working live (LED: t=3.669, df=27, $p<0.001$; Lock: t=4.195, df=27, $p<0.0005$; Total: t=6.557, df=28, $p<0.0005$).

Table 4's next column ("Seconds per change") shows an interesting contrast. In the LED problem, subjects took significantly less time per change in the live version (t=2.872, df=28, $p<0.008$) than in the non-live version. However, the time per change for the Lock problem was nearly the same for both the live and non-live versions. Due to the large differences in the LED problem, the overall

	Number of subjects
Low live, low non-live	6
Low live, high non-live	2
High live, low non-live	6
High live, high non-live	15
Total	29

Table 3: Summary of within-subject performance level comparisons live versus non-live.

differences were also significant: in the live version, subjects took significantly less time per change overall (t=2.341, df=56, p<0.023).

Since there were more changes live, but (overall) each change was done faster, another question arises: did the subjects spend more time in total making changes in the live version or in the non-live version? As Table 4's "% time" column shows, subjects did indeed allocate significantly more of their time to making changes in the live version than in the non-live version (t=-2.502, df=56, p<0.016). Given the previous two columns, it is not surprising that this difference came mainly from the Lock problem, in which subjects spent significantly more time making changes in the live version than in the non-live version (t=4.289, df=28, p<0.0005). Interestingly, for the LED problem, in the live version the higher number of changes balanced out against the lower time per change, resulting in total time spent making changes being nearly the same in the live version as in the non-live version.

Another aspect of debugging behavior is how uniformly subjects' change activity is distributed over time. To assess this aspect, a sequence of two or more changes was defined to be a "burst" if the time between successive changes in the sequence was 9 seconds or less apart. (We chose 9 seconds by looking at the data and finding the smallest threshold at which there would not be hairline differences between what was classified as part of a burst and what was not). The means are shown in Table 4's "Number of bursts" column. The non-live version for Lock problem was the only case in which fewer than half the subjects (only 1/3) had bursts. In the other three cases more than 55% had at least one burst of activity. The average number of bursts for the live version was significantly different from that for the non-live version (t=3.449, df=28, p<0.002).

Finally, we considered the time of the first change. Subjects made their first change significantly sooner in the live version than in the non-live version (t=-2.303, df=28, p<0.029).

These behavior differences seem consistent with the many studies cited by Shneiderman [16] showing that programmers perform more interactions when the system responds faster. Still, in looking for possible confounding factors, it would be reasonable to suspect that some of these behavioral differences could be attributed to differences in subjects' high/low performances rather than working live versus non-live. However, t-tests showed no significant differences in any of the categories represented by Table 4's columns when compared according to high versus low performances instead of the live versus non-live comparison shown.

	Number of changes		Time (sec.) per change	% time spent chan-ging	Num-ber of bursts	Time (min.) of first change
	count	mean	mean	mean	mean	mean
Live						
LED (n=14)	252	18.00	16.13	29.7%	2.71	3.08
Lock (n=15)	249	16.60	18.68	37.5%	1.40	2.30
Total (n=29)	501	17.28	17.45	33.8%	2.03	2.68
Non-live						
LED (n=15)	127	8.47	35.96	31.0%	0.33	3.84
Lock (n=14)	117	8.36	18.79	17.2%	0.86	3.82
Total (n=29)	244	8.41	27.67	24.4%	0.59	3.83

Table 4: Change data. Note that the LED and Lock problems are not comparable against each other in this table, because they required different numbers of changes.

Results: Does liveness affect debugging speed?

We chose to investigate the question of debugging speed separately for the high and low performance levels, because the implications of a faster rate could be very different in those two situations. Whenever the high-performing subjects finished up quickly, this reflected an efficient rate of achieving their successful outcomes. On the other hand, when low-performing subjects finished up more quickly, they stopped too soon, since (by definition) they did not correct very many of the bugs.

We investigated debugging speed over time by considering what percent of a subject's total changes had been made by 5, 7.5, and 10 minutes into the debugging session. Table 5 collects this information into all the high performances of all subjects versus all the low performances. In the high performances, subjects were more efficient in the live version at the 5-minute point, but less efficient by the 10-minute point. However, in the low performances, subjects in the live version remained actively involved in making changes longer, which is appropriate for low performing subjects. These differences are highlighted in Figure 4.

	By 5 min.	By 7.5 min.	By 10 min.
High performances (m=38)			
Live (m=21)	32.8%	58.8%	73.1%
Non-live (m=17)	23.9%	55.2%	76.3%
Difference	8.9	3.6	-3.2
Low performances (m=20)			
Live (m=8)	19.8%	35.9%	61.0%
Non-live (m=12)	27.7%	52.7%	68.7%
Difference	-7.9	-16.8	-7.7

Table 5: Percents of all the subjects' changes that had been made by 5, 7.5, and 10 minutes into the experiment, separated into their high performances and low performances. Variable m identifies the number of performances.

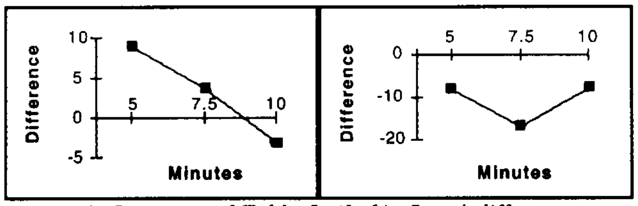


Figure 4: Summary of Table 5. (Left): Speed differences over time between the live and non-live versions for high performances. The live version had a greater percent completed at the 5-minute mark, but was overtaken by the non-live version by the 10-minute mark. (Right): Speed differences over time between the live and non-live versions for low performances. In these performances, subjects remained actively involved in making changes longer in the live version.

Table 5 is intriguing, but it combines independent data with paired data, thereby preventing straightforward statistical analysis of it. However, we were able to analyze a segment of it by extracting a within-subject subgroup: namely, the subjects who had consistently high performances. See Table 6. (The opposite within-subject subgroup, those who had consistently low performances, had only 6 subjects, which were too few for meaningful analysis.) The columns count how many subjects had made more than 0% of their own total changes by the 5-minute point, 25% of their changes by the 7.5-minute point, and 50% of their changes by the 10-minute point. For the consistently high performers, working in the live version was associated with completing a greater portion of their changes at the 5-minute point (χ^2=6.0, df=1, p<0.05), but not at the 7.5- or 10-minute points.

DISCUSSION

Many developers of direct-manipulation programming systems (including several of the authors of this paper) have believed that liveness would make a significant difference in the accuracy or speed with which bugs can be identified and removed from a program. From the post-test questionnaires, we also know that our own subjects believed liveness helped with accuracy: 75% of them expressed more confidence in the problems they worked in the live version. Our results show that these beliefs are not necessarily true: there were no significant overall accuracy differences in our study, and the speed differences that could be tied with efficiency seemed to favor the non-live version.

However, this lack of overall difference was actually a mixture of opposing results. In fact, liveness sometimes significantly improved accuracy and sometimes slightly impaired it. We believe that this finding, especially when added to the collection of mixed predictions from related work, shows that the questions of whether liveness affects debugging accuracy, behavior, or speed are complex—they seem to combine many factors.

	>0% by 5 min		>25% by 7.5 min		>50% by 10 min	
	Live	Non-live	Live	Non-live	Live	Non-live
High live, High non-live	15	10	14	11	13	13

Table 6: Counts of subjects who had reached the thresholds shown by 5, 7.5, and 10 minutes into the experiment. These are within-subject comparisons of the 15 consistently high-performing subjects.

Thus the next steps must be further investigations into just what these factors are. Our study's results suggest three: type of problem, type of user, and type of bug. Regarding the first, type of problem, in our study liveness significantly improved accuracy on the Lock problem, but it slightly impaired accuracy on the other. Was the difference due to the different problem domains (mathematical versus graphical)? Investigating this question could help developers of programming systems for particular problem domains choose whether to make their systems live.

There was also a significant effect on accuracy for one kind of subject—those who were low performers in at least one version. There were also indications of improvement in active involvement for subjects who were consistently low performers, although there were too few such performers for meaningful statistical analysis of this aspect. These findings indicate that a repeat study is warranted, using subjects with little debugging experience, to further investigate whether and how liveness affects this kind of subject in particular. Positive findings about the effects of liveness on subjects with little debugging expertise would bode particularly well for the use of liveness in end-user programming systems.

It is interesting to consider the high performers' changing differences in speed over time. The strong behavior differences in the live version versus the non-live version suggest that they used different debugging strategies in the two versions, which in light of the speed patterns seemed superior in the live version with the bugs they worked on first, but seemed to hinder debugging with the bugs they worked on later. An important next step for us will be an investigation into what these speed patterns can show about the effects of liveness on fixing different kinds of bugs.

An as-yet open question is whether liveness helps keep the bugs out during initial program construction. Our study did not address this important question, but it is one for which many developers have strong intuitions that need to be either debunked or verified.

CONCLUSION

From this study, we obtained four key results. First, liveness was not the debugging panacea that developers of direct-manipulation programming systems might like to believe it is, but it did help significantly with accuracy in some situations. Second, there were significant differences in debugging behavior in the live versus non-live versions. Third, there were differences in debugging speed over time in the live versus non-live versions; for high performers these differences were significantly in favor of the live version at the 5-minute point but were ultimately overtaken by the non-live version.

The fourth result was the identification of three factors that, when varied, seemed to produce opposing results: the type of problem, the type of user, and the type of bug. Although the importance of these factors in debugging is documented in classical debugging literature, the new information from our study suggests that they may in fact be critical in determining whether *liveness* will help debugging or hinder it. If future research into these factors can provide an understanding of the particular situations in which liveness does help, then developers of direct-manipulation programming systems will have more of the knowledge they need to incorporate liveness effectively.

ACKNOWLEDGMENTS
We are grateful to Thomas Green and Sherry Yang for their helpful suggestions and feedback. Thanks are also due to Margaret Burnett's Fall '95 CS 411 students for their enthusiastic participation in the study, and to Rebecca Walpole and her Summer '95 CS 381 students for their help during the pilot. This work has been supported in part by the National Science Foundation under grant CCR-9308649 and an NSF Young Investigator Award.

REFERENCES
1. Atwood, J., Burnett, M., Walpole, R., Wilcox, E. and Yang, S. Steering Programs Via Time Travel, *1996 IEEE Symp. Visual Languages*, Boulder, CO, Sept. 3-6, 1996, 4-11.
2. Brown, P. and Gould, J. Experimental Study of People Creating Spreadsheets. *ACM Transactions on Office Information Systems* 5, 1987, 258-272.
3. Burnett, M. and Ambler, A. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. *Journal of Visual Languages and Computing* 5(1), Mar. 1994, 29-60.
4. Cypher, A. (ed.) *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA, 1993.
5. Eisenstadt, M. Why HyperTalk Debugging Is More Painful Than It Ought To Be. *HCI'93 Conference*, pub. as *People and Computers VIII*, Cambridge University Press, Cambridge, UK, 1993, 443-462.
6. Eisenstadt, M. Tales of Debugging from the Front Lines. *Proc. Empirical Studies of Programmers*, Ablex Publishing, Norwood, NJ, 1993, 86-112.
7. Gilmore, D. Interface Design: Have We Got It Wrong? *INTERACT '95*, (K. Nordby, D. Gilmore, and S. Arnesen, eds.), Chapman & Hall, London, England, 1995.
8. Green, T. and Petre, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing* 7(2), Jun. 1996, 131-174.
9. Gugerty, L. and Olson, G. Comprehension Differences in Debugging by Skilled and Novice Programmers. *Proc. Empirical Studies of Programmers*, (E. Soloway and S. Iyengar, eds.), Ablex Publishing: Norwood, NJ, 1986, 13-27.
10. Lawrence, A., Badre, A. and Stasko, J. Empirically Evaluating the Use of Animations to Teach Algorithms. *1994 IEEE Symp. Visual Languages*, St. Louis, MO, Oct. 4-7, 1994, 48-54.
11. Martin, G. and Corl, K. System Response Time Effects on User Productivity. *Behaviour and Information Technology* 5(1), 1986, 3-13.
12. Nanja, M. and Cook, C. An Analysis of the On-line Debugging Process. *Proc. Empirical Studies of Programmers*, (G. M. Olson, S. Sheppard, and E. Soloway, eds.), Ablex Publishing, Norwood, NJ, 1987, 172-184.
13. Nardi, B. and Miller, J. Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development. *International Journal of Man-Machine Studies* 34, 1991, 161-194.
14. O'Hara, K. Cost of Operations Affects Planfulness of Problem-Solving Behaviour. *CHI '94 Conf. Companion*, Boston, MA, Apr. 24-28, 1994, 105-106.
15. Oman, P., Cook, C. and Nanja, M. Effects of Programming Experience in Debugging Semantic Errors. *Journal of Systems and Software* 9, 1989, 197-207.
16. Shneiderman, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading, MA, 1992.
17. Svendsen, G. The Influence of Interface Style on Problem-Solving. *International Journal of Man-Machine Studies* 35, 1991, 379-397.
18. Tanimoto, S. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing* 2(2), Jun. 1990, 127-139.
19. Whitley, K. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing*, 1997 (to appear).
20. Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., and Cook, C. Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems? An Empirical Study. TR 96-60-9, Oregon State University, Dept. of Computer Science, (303 Dearborn, Corvallis, OR 97331 USA), Aug. 1996.