Leogo: An Equal Opportunity User Interface for Programming

Andy Cockburn
Department of Computer Science
University of Canterbury
Christchurch, New Zealand
andy@cosc.canterbury.ac.nz

Andrew Bryant
Department of Psychological Medicine
Christchurch School of Medicine
Christchurch, New Zealand
abryant@chmeds.ac.nz

Contact author: Andy Cockburn

Abstract

Leogo is a novel programming environment supporting an "equal opportunity" user interface which allows users to express their programming tasks through any mixture of three concurrently active programming paradigms: by direct-manipulation using 'programming by demonstration'; by clicking buttons and dragging sliders in an iconic language; and by typing commands in a normal text-based language. Equal opportunity ensures that the effects of any interface action are simultaneously displayed across each of the three paradigms—input expressions in one paradigm cause output of equivalent expressions in the other two paradigms. Leogo is designed to promote programming skills in primary and junior schools, but the interface properties it demonstrates are applicable to a wide range of novel programming environments. Leogo's motivation, design, development, and preliminary usability study are described.

1 Introduction

Programming skills are becoming increasingly important at work and at home. Many standard software packages such as spreadsheets and word-processor mail-merge facilities require programming skills, as do household appliances such as VCRs and microwave cookers. Few would doubt that in the coming decades software systems and 'embedded machines' will become much more prevalent and user-tailorable. We believe it highly desirable that schoolchildren be introduced to programming techniques at an early age. We also believe that the user interface to systems for programming can critically influence the development of programming skills.

Many researchers are exploring new paradigms for programming. Kahn [13], for instance, notes the "incredible breadth of ways that programs can be expressed, ranging from sketches on paper to video-game animation to manipulation of physical objects," and systems demonstrating many alternative paradigms have been developed [22]. It is clear, however, that no single programming paradigm is ideal for all uses and users. One portion of a program may be best defined through animated programming by demonstration, another portion through static graphics (such as state-transition diagrams), and another portion by traditional text-based programming instructions. Furthermore, the preferred mechanisms for expressing and representing programs will differ between programmers.

In this paper we describe Leogo, a programming environment that supports user-interface equal opportunity between multiple paradigms for expressing programming tasks. Our work on Leogo, which is targeted at primary and junior school children, is motivated by pedagogical and user-interface research interests. In this paper we focus on the potential of equal-opportunity user interfaces for programming and on interface design issues.

Based on Logo [27, 28], Leogo allows users to articulate their programming tasks through any mixture of three alternative programming paradigms—a direct manipulation environment for programming by demonstration, an iconic language for graphical programming, and a text-based dialect of Logo. To reinforce the user's perception of equivalence between the three programming styles, Leogo supports full user-interface equal opportunity [36]—the input actions in one programming paradigm cause corresponding outputs in the other two programming paradigms.[1] To encourage exploratory learning, Leogo also supports full state-history allowing almost unlimited undo and redo.

The structure of the paper is as follows. Section 2 describes "equal opportunity" as a user interface principle, and reviews work on novel educational programming environments. Section 3 describes Leogo and discusses its interface design rationale. Leogo's preliminary evaluation, which was intended to illuminate problems in its user interface, is discussed in Section 4. Related work on multi-paradigm programming systems is described in Section 5, and directions for future work are presented in Section 6. Section 7 summarises the paper.

2 Background

In this section we describe "equal opportunity" in user interfaces, and we review work on novel programming paradigms. Closely related work on multi-view systems for programming is described in Section 5 after the description of Leogo.

[1] Compromises in this design goal will be discussed in the paper.

2.1 Equal Opportunity Interfaces

Equal opportunity user interfaces blur the distinction between input and output [36]. By doing so the flexibility, utility, and ease of use of software can be increased. Equal opportunity properties are seen in many modern user interfaces, and when successfully implemented they provide a seamless part of the dialogue between user and computer. The ambitions of equal opportunity, and the extents to which interfaces support it, can be broadly categorised into three levels, discussed below.

2.1.1 Activating output for subsequent input

In the least ambitious form of equal opportunity, computer outputs are activated so that they can be used for expressing subsequent input. For instance, many `Find File` facilities present, as output, several filenames that match a user's query. In an equal opportunity interface, the output file names are themselves active interface elements that can be used for expressing subsequent input—if the user clicks on the filename, then the corresponding file is opened. Without equal opportunity, the computer's output must be massaged by the user into a subsequent input expression. In a command-line environment, for instance, this will involve re-typing the file name, or copying and pasting it.

2.1.2 Exchanging the roles of input and output

Equal opportunity can be extended far beyond making output objects active. Interface mechanisms normally reserved for presenting output can, in certain domains, be adapted to allow the user to express the *desired* output. The equal opportunity interface then shows what inputs would be necessary in order to produce that output.

Spreadsheet applications partially achieve this level of equal opportunity because no cell has a specific role as either input or output: input to a cell may be a formula that determines the cell's output from values input to other cells. Interestingly, although much of the spreadsheet programmer's power is derived from this blurring of input and output roles, guidelines for spreadsheet design recommend that input and output regions should be clearly separated to clarify roles for the end-user [12].

The roles of input and output can, in certain domains, be entirely removed so that users are free to express causes and see the effects, or conversely to express effects and visualise the causes. Thimbleby [36] demonstrates such properties in an equal opportunity calculator which allows the user to enter either of the following expressions.

$16 + 6 * 4 = \diamondsuit$

$16 + \diamondsuit = 40$

The calculator's interface completes the expressions, replacing the diamond with the appropriate value.

2.1.3 Multiple representations of input and output

Most computer applications support only a single syntax for expressing input: for instance, a text-based command language, or a direct manipulation graphical user interface. Similarly, applications normally only support a single interface mechanism for presenting output. Exceptions which allow multiple representations of input and output are common, however, and

they add to the flexibility of the user interface. For instance, many commercial spreadsheets allow users to view data directly on the spreadsheet or through graphical representations.

With equal opportunity between multiple representations there is no restriction on the user's interaction with any of the representations for input and output. In the spreadsheet example, not only would users be able to view data values in a graphical format, but they would also be able to manipulate the values from the graph (perhaps allowing students to see the change in numerical values that occur as a graph-plot curve becomes a straight line).

Leogo, the system described in Section 3 supports extensive equal opportunity facilities at all three levels described above.

2.2 Novel Programming Paradigms

In his book "Mindstorms", Papert [27] describes the potential of computer programming languages to provide a platform for education on structured thinking processes. His language, Logo, allows children to issue programmed instructions to a computerised 'turtle' which leaves a trail on the floor (if mechanical) or on a computer screen. The graphical output produced by turtle motion is intended to act as a motivator for children, and it provides a concrete metaphor for their programming tasks.

This section reviews some systems that support novel approaches to computer programming, particularly those that are designed primarily for children. The systems are presented in three overlapping categories of programming paradigms, as follows: natural and tangible languages, visual programming, and implicit programming.

2.2.1 Natural and Tangible Languages

The cognitive problems associated with learning and manipulating formal computer languages has motivated the development of programming languages that are similar to the user's natural language. The programming languages BASIC (Beginner's Abstract Symbolic Instruction Code) and Cobol, and more recently HyperTalk [4], all attempt to increase the similarity between English and programming languages. However, natural languages such as English have weakly defined grammars, and are highly permissive in syntactic variance. The designers of 'natural' computer languages must therefore address a trade-off between the 'naturalness' of the language (the amount of syntactic variability it will parse) and the complexity of the language's parser. Consequently there is a mismatch between natural and programming languages, and although the languages initially seem natural and intuitive, they often become frustrating when more complex tasks are attempted [37].

Changing the grammar of programming languages is not the only way to increase the naturalism of programming. Many of the mechanisms for program representation (output) can be altered to better suit youthful programmers. Logo, for instance, uses graphical representations (turtle graphics) to provide more concrete representation of program activity. The use of graphical and audio output is extended in AlgoArena [16] which uses a Sumo-wrestling metaphor to depict program execution.

The mechanisms for program expression (input) can also be altered to match the needs of young programmers. Logo uses appropriate mnemonics for the language's lexems, for instance renaming Lisp's 'CAR' and 'CDR' functions 'FIRST' and 'BUTFIRST'. Furthermore, iterations on Logo, such as AlgoBlock [35], note the difficulty that children have in manipulating a keyboard, and observe the value of tangible artifacts in program expression. AlgoBlock

allows children to 'build' programs by connecting electronic building blocks, each of which has a discrete command associated with it, to form a sequence of program steps. AlgoBlock provides a novel input mechanism, but its lack of iteration and parameters limits its use to early primary children. sLogo [11] provides an iconic equivalent to AlgoBlock, but with a larger subset of Logo. sLogo commands (including iteration) are connected to form an iconic representation of the program through a drag-and-drop interface.

2.2.2 Visual Programming

Graphical constructs such as flowcharts are often used to help visualise the behaviour and performance of programs. Several visual programming environments have been developed to support an improved mapping from graphical program expression to resultant executable code: examples include Pygmalion [31], Pict [9], and Pictorial Janus [15]. Many Computer Aided Software Engineering tools also support graphical descriptions of system components [34].

Several researchers are investigating graphical means for expressing programs within children's programming environments. KidSim [33] (now renamed Cocoa [32]) is a radical departure from Logo-based educational programming paradigms. It supports animated simulations that are described using visually programmed graphical rewrite rules. Graphical rewrite rules show a *before-and-after* snapshot of a portion of the simulation. While the simulation runs, whenever the *before* image appears, it is replaced by the *after* image. Rewrite rules can have conditions (or "property tests") associated with them, and they can be generalised in a variety of ways. KidSim's graphical rewrite rules escape the need for text-based input, and for a strict grammar. In our experience with KidSim[2], predicting the behaviour of the simulation from the list of graphical rewrite rules can be difficult, and may encourage ad-hoc programming strategies, with additional rules being added for each of the "unexpected" graphical states generated in the simulation.

ToonTalk [14, 13] deviates even further from standard programming environments than KidSim. Like KidSim, the execution of ToonTalk programs is animated, but unlike KidSim, the act of expressing a program is based on a video-game metaphor in which, "for example, a computation is a city, an active object or agent is a house, birds carry messages between houses..." [14]. ToonTalk's evaluation is underway, and reports on the evaluation will be extremely interesting.

Myers [22] provides an extensive review of research on visual programming systems.

2.2.3 Implicit Programming

In normal text-based programming, and in the novel schemes described above, the user explicitly provides instructions to the computer on the actions to be performed. In 'implicit programming' the computer system observes user actions, and generates a model of the user or their actions that can be used to predict future actions. Included in this category are systems that support 'programming by demonstration' and 'programming by example' [23, 7, 22]. Although these techniques may include *some* explicit instruction from the user, it is normally limited to meta-instructions such as "Start observing now". The notion of generalising the user actions is central in many programming by demonstration systems. Through abstraction these systems can generalise from the demonstrated sequences to unseen, but similar,

[2]Thanks to Apple Computer Inc. for providing a prototype copy of the software.

sequences. Example systems include Eager [6], Metamouse [20], and much of the work on "Intelligent Agents" [19].

KidSim's graphical rewrite rules demonstrate rudimentary programming by example techniques, but KidSim does not support autonomous abstraction from sample cases to similar ones.

3 Leogo: Intentions and Description

This section describes three aspects of our experimental programming environment, Leogo. First, the motivation for equal opportunity interfaces within children's programming environments is reviewed. Leogo's interface is then described. Third, some complex interface design decisions are considered.

3.1 Potential of Equal Opportunity Interfaces for Programming

Section 2.2 reviewed several programming environments that provide novel mechanisms for expressing programs and for viewing their output or execution. In these systems there is a single consumer-level paradigm for program input and one for program output. Several environments support multiple representations of programs, but normally only one representation mechanism is intended for use by end-users. For instance, programming by demonstration systems normally convert the user's actions into text-based generalised and parameterised descriptions. These descriptions are used internally by the programming environment, but they are not manipulated directly by the user. Furthermore, most of the systems reviewed in Section 2.2 have substantial seams between the mechanisms for program expression and those for representing program execution. Exceptions include the systems for programming by demonstration in which program behaviour is expressed in terms of actions at the interface. KidSim and ToonTalk also reduce the seams between the mechanisms for expression and representation through visual programming, but they still require separate 'modes' for programming (rule creation) and execution (rule use).

Through Leogo, we are exploring the potential of equal opportunity user-interfaces in programming environments. Our aims are to support multiple concurrently active programming paradigms, and to remove all preconceived notions of input and output roles between them. We believe these facilities offer many benefits to users.

In an environment that supports more than one way of articulating programming tasks (with equal opportunity between them), the user is free to select whichever paradigm(s) that best suits their current needs. Additionally, by supporting multiple means for articulating programming tasks, an educational system can span a wide range of educational abilities and ages. In Leogo, for instance, young children can program using the direct manipulation mechanisms, and older students can use the more abstract but powerful text-based language.

We believe that equal opportunity is also a powerful tool for self-directed learning. Traditional programming environments require a novice programmer to work forwards, from the program towards the desired solution. This can be a frustrating and disheartening process. With equal opportunity the user can state (or demonstrate) the required result, and view a program that produces the same result. We contend that users may gain an accelerated and deeper understanding of the mechanisms of programming by seeing how the alternative programming paradigms achieve equivalent effects. Naturally, there are limits on the generality of

equal opportunity, such as the problems of inferring repeated sequences, but we are interested to see how far we can extend equal opportunity in a child's programming environment.

3.2 The Leogo System

[Figure 1 about here.]

Figure 1 shows Leogo's three programming paradigms: the iconic programming environment on the left; the direct-manipulation environment in the middle; and the text-based environment on the right. There is extensive equal-opportunity between the three environments, allowing the user to express programming tasks in any of the three environments, and to see the corresponding expression in the other two environments. Programming tasks initiated in one environment can be completed in either of the other environments. Each of the programming environments is described below.

3.2.1 Text-based programming

Leogo's text-based programming paradigm, shown on the right of Figure 1, is equivalent to standard Logo programming. A full dialect of Logo is supported, with the exception of list processing commands. Program lines are typed into the text-entry widget at the bottom of the screen, and the line is executed when the user clicks the "Do It" button or when they press the return key. The scrollable list-box shows the history of previously executed commands, which may have been expressed in any of the paradigms. Users can re-execute lines or sequences of lines by selecting them in the list-box and by clicking "Do It."

3.2.2 Iconic programming

The left-hand window of Figure 1 provides iconic representations of all of Leogo's language elements. The iconic mechanisms for Leogo constructs are, from the top of the window to the bottom, as follows: mechanisms for defining new procedures, parameters, and sample parameter values; constructs for creating repeat loops; constructs for generating conditional statements; icons for calling a series of Leogo commands that take no parameters (including a "Calculator" for generating expressions); mechanisms for calling turtle-motion procedures; and icons allowing the user to invoke user-defined procedures with parameter values. All iconic programming actions cause corresponding actions to be displayed in the text-based programming window and in the direct manipulation programming window.

Parameter values for any procedure invocation are set through the slider widgets alongside the procedure icons. User-defined procedures can display an arbitrary number of parameter sliders, but procedures defined within the iconic programming environment are limited to only two parameters (a design trade off intended to save screen real-estate).

Expressions, such as $: size < : height * 5$, are supported in the iconic environment through Leogo's calculator. Similar to KidSim's calculator [33], the calculator allows users to enter expressions which include any of the local parameters declared for procedures within the current scope level. When the user clicks in the conditional regions of the `IF` expression, the calculator pops up. Expressions can also be used for the bounds of `REPEAT` loops.

3.2.3 Direct manipulation programming

In standard Logo, the middle window of Figure 1 would be the output region displaying the turtle-motion described by a text-based program. In Leogo, however, this region is a direct manipulation programming environment. Users generate Logo commands by dragging different segments of the turtle with the mouse. Corresponding programming actions that produce identical turtle motion are simultaneously displayed in the iconic and text programming windows. Dragging the turtle's head causes it to rotate on the spot. Dragging its body causes straight line motion forwards or backwards, and clicking the turtle's tail toggles between Pen-up (no trail on motion) and Pen-down (leaving a trail).

The tape recording icons at the top of the window allow the user to record procedures, and to undo and redo previous actions. Recording procedures is described in Section 3.3.2. Undo and redo are particularly valuable in educational environments as they encourage students to explore without concern for the consequences of erroneous actions. Full undo and redo is supported in each of Leogo's programming environments.

3.3 Interface Design Issues

During the conceptual design of Leogo, and during its iterative development, we continually encountered interface conundrums. In this section we review some of these problems, and the partial solutions that we selected. Many of these problems stem from the tensions between our desire for user interface consistency, the need for equivalence between the equal opportunity programming paradigms, and the substantial difference between the programming mechanisms offered by these paradigms.

3.3.1 Abstracting the Object-Action Paradigm

In direct manipulation interfaces, control of objects is best achieved through an Object-Action paradigm [1]. In Logo, however, the turtle is the implicit object of all interface actions. Logo's user commands are of the form `Action-Amount`, where the amount may be zero or more values (parameters).

Leogo's text-based programming environment is consistent with Logo (for example, `FORWARD 90`). In the direct manipulation environment, however, the implicit object `Action-Amount` grammar is problematical. We wished to support a highly tangible relationship between object (the turtle) and action (the command sequence). To achieve this we chose to use portions of the turtle's body to activate specific Logo commands (its body for `FORWARD` and `BACK`, its head for rotation, and its tail for pen control). Our preliminary usability study indicates that this solution is readily accepted by young children (Section 4).

In the initial design of Leogo's iconic programming environment we accidentally deviated from the `Action-Amount` ordering, and required the `Amount` to be specified before the `Action`. The severity of this error was immediately obvious in preliminary user-trails. Users would continually click the `Action` first (for example `FORWARD`), and then be surprised that they could not subsequently set the associated value. This error of consistency was easily remedied. In the current version, parameter sliders are disabled until the user clicks the associated action which causes each of the parameter sliders to become active, and highlighted in green. As the user sets each parameter it becomes inactive once again (returning to the default colour). Once all parameter values are set, the command is executed. A side-effect of this solution is that as soon as the user releases a particular parameter's slider control, the value is committed

and cannot be reset. In our preliminary trials, this interface feature resulted in several "off by one" errors, where the slider value nudged off the desired value as the mouse-button was released. However, users seemed more willing to accept this problem (which could be remedied by undo) than the previous problem of the misunderstanding raised by the `Amount-Action` paradigm (further solutions to this problem are discussed in Section 6).

3.3.2 Procedures and parameters

Designing direct-manipulation and iconic equivalents to text-based procedure definition was a major challenge. In Logo, procedure definition is analogous to teaching the turtle how to carry out a generic set of actions, but users cannot see the effects of their procedure until it is completed and called (a problem also noted by Lieberman [18]). We believe that continual visual feedback during procedure definition is useful, particularly for younger users. Consequently, all three environments the user the effects of their procedures *as* they are defined, reinforcing the notion of programming by demonstration.

To define a procedure in the direct-manipulation environment, the user clicks the 'Record' button at the top of the middle window of Figure 1. When they do so, the 'Stop' button becomes active, the tape-reel icon in top-right of the icon-programming window becomes animated to show that the user's actions are being recorded, and the associated procedure declaration code is shown in the text window. All user actions while the tape is running are contained in a new procedure that the user names when the stop button is pressed. The new procedure can be called from the icon programming environment or from the text programming environment, but in the current implementation procedures cannot be called directly from the direct-manipulation environment. Another limitation in the direct-manipulation environment is that users cannot define parameters for their procedures.

The icon-programming environment is more powerful than the direct-manipulation environment, allowing the full range of Logo facilities. Like the direct-manipulation environment, it provides continuous feedback showing users the results of their procedures as they are defined. To define a procedure in the iconic programming environment the user types the name of the procedure and its parameters. They then use the sliders to supply sample values for each of the parameters. The sample values are used to provide a demonstration of the procedure's effect as it is defined. In the iconic programming environment, parameter values are accessed through the calculator.

The increased power of procedure declaration in the iconic programming environment comes at the cost of increased interface complexity. We believe, however, that these mechanisms can act as a bridge between the simplistic direct-manipulation methods, and the powerful but abstract text-based mechanisms.

4 Usability Study

Throughout its development, Leogo's interface has been continually assessed by computer science students. Some of the design alterations brought about by these initial user-trials have been discussed earlier in the paper (for instance, the `Action-Amount` interaction paradigm described in Section 3.3.1). Although useful in shaping portions of the interface, the expertise of computer science students is very different to that of Leogo's target user-base. Once we believed Leogo to be robust, we invited primary school children to experiment with it. We felt that the equal-opportunity programming facilities were natural and intuitive, but there

is substantial evidence of gross gaps between designer's expectations and user-experiences in HCI literature [10]. Prior to committing extensive time and effort (for both the researchers and the school involved) on a full-scale evaluation, we ran a usability study of the system with Leogo's target user base. The aims of the usability study were not to determine the effectiveness of the system as a learning tool, rather we were looking for major interface flaws and omissions, and for initial feedback on the effectiveness of the equal-opportunity paradigm.

The usability study focused on the novel parts of the interface—the direct-manipulation programming environment, the iconic programming environment, and the equal opportunity between them. Ten standard four primary school children (fifth grade USA), aged ten and eleven, were invited to test the system in one-hour sessions at the University. Holding the study at the University allowed us an interruption-free environment that would not have been available at the school. As yet we are not analysing the effectiveness of the system in a situated learning environment, but this is planned for our further work.

The evaluation was based on constructive interaction [25], where pairs of users share control of the system. Constructive interaction was used for two reasons. First, class-room pressure on machines means that primary students normally share computers, so the partnership of constructive interaction would be representative of classroom use. Second, constructive interaction results in natural conversation between the subjects, which reduces the need for the disruptive prompts for users to 'think-aloud'. Consequently, constructive interaction encourages the students to verbalise their problem solving strategies, and allows evaluators access to the users' understanding (and misunderstanding) of the system. All the subjects had previous computer experience (most citing games), and all were familiar with the mouse. Two of the subjects had prior experience with Logo.

In a brief pre-test the subjects were reassured that they were testing the system, and not themselves being tested. Through the session, portions of Leogo's interface were introduced, and the subjects were given small tasks to carry out on the introduced features. Once all the features had been introduced, the subjects were asked to solve larger programming tasks. The portions of the system were introduced as follows.

1. Basic direct-manipulation facilities. The subjects were asked to move the turtle forward and back, to turn it left and right, to put the pen up and down between movements, and finally to write their initials using the turtle's trail.

2. Basic iconic-programming facilities. The subjects were asked to move the turtle forward and backward by specific amounts (as set by the parameter sliders), to rotate it left and right, to undo and redo actions, and to reset the screen. They were then asked to draw their initials using only the iconic programming facilities.

3. Equivalence of expressions in the direct-manipulation and iconic programming environments. Subjects were shown that equivalent commands were being produced in the text-based environment, and they were asked to turn the turtle to the left in each of the environments. The text-based environment was not described further. The subjects were asked to confirm the equivalence of expressions by drawing two similar shapes, first using the direct manipulation environment, then using the iconic environment. They were told that they could copy the values for the sliders from the text-display of past commands.

4. Recording procedures in the direct manipulation environment, and calling procedures in the iconic environment. Subjects were shown how to record a series of actions using the

tape-recorder buttons in the direct-manipulation window, and they were asked to record a procedure to draw a triangle, and to name it "tri". Care was taken to ensure that the subjects noticed that a new button appeared in the iconic environment as soon as the new procedure was named. They were then asked to call the procedure several times by clicking the new button. In all cases, the multiple calls to the procedure produced interesting spiral patterns due to small errors in subjects' drawing of the triangles. This had a very positive effect on the subjects' motivation and interest.

5. Repeat. The repeat facility in the iconic programming environment was described. Subjects were then asked to create repeat constructs that grouped a small motion forwards with a call to their "tri" procedure, generating a variety of spiral patterns.

6. Procedures with parameters in the iconic programming environment. Subjects were shown how to create a procedure to draw a square of any size using parameters in the iconic programming environment. This included showing the subjects how to refer to parameters using the iconic calculator. Where time permitted (three of the pairs), the subjects were asked to create a procedure that could draw spirals of various sizes, as determined by a parameter.

7. At least 5 minutes was reserved for free-form exploration with the system. Subjects were encouraged to draw polygons, circles, and spirals, and to express their programming solutions in whatever way they preferred.

Each session included a five-minute post-task interview in which the subjects were invited to comment on the system.

4.1 Observations

All of the subjects clearly enjoyed using the system, and all of them made reasonable progress during the single one-hour session. Several low-level user interface bugs were repeatedly encountered, particularly the problem of setting precise values with the 'fiddly' slider controls. In general, all subjects reported that they understood the equal opportunity between the three programming paradigms, and we made no observations that would contradict their claim. None of the students entered commands into the text-based programming environment, except when requested to do so.

Some of the more surprising and interesting observations are reported below.

4.1.1 Forward versus Backward Error Recovery

Although all of the subjects were informed that Undo was continually available, forward error recovery was used much more often than backward. For instance, when trying to complete one side of a rectangle, a subject marched the turtle forward little by little, filling in the remaining distance rather than undoing the previous action and trying a larger single step.

We suspect that Undo and Redo will be more useful for more sophisticated programming tasks, particularly those involving procedural abstraction, where greater precision is required.

4.1.2 Arbitrary Selection of Programming Paradigms

We had hoped that during the subjects' free-form exploration of the system we would be able to observe how and why particular programming paradigms were selected. Although we were

not surprised that our youthful subjects showed little interest in the text-based paradigm, we were surprised that there appeared to be little task-specific preference for one paradigm over any other. We had expected that direct manipulation would be used for "doodling", for short abstract tasks, and for demonstrating non-precise motions inside procedures and repeat loops. We could not, however, detect any major bias between the mechanisms during particular tasks. All activities, from doodling to procedure definition, were carried out in both the iconic and direct-manipulation environments, and users often switched between the two environments without any evident rationale for doing so. When asked in the post-test interview which environment they preferred, there were no clear preferences. Several users stated that motion forwards and backwards was easier in the direct-manipulation environment because they could make the turtle move whatever distance they liked, and that turning the turtle was easier in the iconic environment because it was tricky to grab the turtle's small head in the direct-manipulation environment.

Interestingly, tasks started using one environment were normally completed within the same environment—probably because of the close proximity of other controls. A common exception, however, that demonstrated appropriate switching between environments was revealed when two of the the subjects pairs drew all but one side of a polygon in the iconic environment, and then completed the final turn and edge using the direct-manipulation environment. This action is probably due to their poor understanding of degrees of rotation, the arbitrary distance measures (described in Section 4.1.3), and the lack of dynamic feedback in the iconic environment (discussed in Section 6.3)..

We suspect that, with greater familiarity with Leogo's programming mechanisms, patterns of use will emerge, and we will investigate these in our upcoming evaluations. None of the subjects reported confusion in the relationship between the three programming environments.

4.1.3 Confusion of distance and angle measures

The numerical values associated with distances and angles were troublesome for the users. The parameters for `FORWARD` and `BACK` correspond to pixel distances, and the parameters for `LEFT` and `RIGHT` correspond to degrees of rotation. Neither of these values were intuitive to the youthful user group. Consequently, some of the pairs used the direct-manipulation environment for activities where closure of shapes was required.

A potential solution to this limitation in the iconic-programming environment is described in Section 6.

4.1.4 Action-Repeat Sequences

When given an iterative task using repeat loops and when defining procedures, several subjects attempted to specify the actions *before* initiating the loop or starting the procedure recorder. Similar problems have been noted by Lieberman [18].

Potential solutions to these problems are described in Section 6.

4.1.5 Premature Closure

All subjects encountered at least one premature closure [36] problem when defining procedures or when creating repeat loops. These problems arose when the subjects forgot to finish a repeat loop or procedure after the desired actions had been completed. In the case of repeat loops, user actions would continue to be encapsulated with the repeat loop, and the subject

would not understand why the actions were not being repeated. Failure to finish procedures meant that the user's click on the procedure icon resulted in recursive calls to the procedure rather than a direct invocation. In most cases the experimenter had to intervene to reveal the problem.

Clearly the cues for the "recording" modes were inadequate. These were the animated tape icon (for procedure definition) and the activated `Finish` button (for repeat loops). Leogo has since been modified to generate an audible tape-winding sound during recording modes. The tape stop button and the `Finish` button are also now coloured bright red while recording.

5 Related work

This section reviews related work on programming environments that support multiple views of programs. Background material on general novel programming paradigms was reviewed in Section 2.2.

Many systems have been developed which support multiple views of programs, such as graphical *and* textual views. Myers [22] reviews dozens of systems that support some form of graphical programming. Many of these systems support a one-way interaction between the alternative program views—the user can only express programming tasks in one view, but the other views are autonomously updated to ensure consistency between the views. Myers uses the term "visual programming" to describe systems that allow program expression through visual means, and the term "program visualisation" to describe systems which provide a graphical representation of programs which are expressed through conventional text-based languages. Our interests lie in systems which support dynamic consistency between multiple views of programs, and which allow the user to express their programming tasks through *any* of the views.

PECAN was an early example of work towards equal opportunity in interfaces for programming [29]. PECAN supported multiple views of the program, and the views were dynamically updated to ensure consistency with the underlying text-based program. PECAN allowed the program to be expressed through several mechanisms, including conventional text-based input, menu selections, and keyboard short-cuts. Graphical editing of the program through Nassi-Shneiderman diagrams [24] was planned for further work.

FPL [5] combined textual and graphical means for program expression. In FPL, graphical mechanisms were used to express the control-flow and logic of the program solution, and text was used to provide specific values. Users could not choose between alternative means of expression. Completed FPL programs were autonomously translated to Pascal for execution, but there was a one-way interaction from FPL to Pascal. Consequently any changes the user made to the Pascal program would not be reflected in the FPL program.

PegaSys [21] supported multiple views of different types of information about programs. It connected a "picture hierarchy" of formal documentation with Ada programming units. PegaSys was not intended to provide pictorial mechanisms for program expression, but it demonstrated some of the ambitions of equal-opportunity through its interactive connections between program elements and documentation. It was intended that facilities for dynamically animating the connections would be implemented in further work.

Balsa[3] [2] supported multiple views of program execution, but not of the program itself. Animated graphics revealed the execution of algorithms, and multiple algorithms could be

[3]Brown University Algorithm Simulator and Animator.

run concurrently to reveal their comparative performance. Balsa was an educational system, designed to assist comprehension of existing algorithms. It was not intended to assist programming, and it did not support equal-opportunity.

More recently, LCSI's[4] system "Micro Worlds 2" [17] provides a polished and comprehensive commercial Logo programming environment. Similar to Leogo, the user can express programming tasks through three interface mechanisms: iconic palettes, a text-based dialect of Logo, and by direct manipulation. Equal opportunity, however, is not provided by Micro Worlds. User inputs in one paradigm do not cause corresponding outputs in the other paradigms, and there is limited equal opportunity "redundancy" across the paradigms: that is, most programming actions are only possible using the iconic palettes. With its impressive range of multi-media facilities, including musical keyboards, QuickTime movies, and photo banks, it is likely that Micro Worlds will become many children's first programming environment.

6 Further work

Leogo's usability study, reported in Section 4, was intended to shape Leogo's user interface and to fuel our further work. Feedback from the evaluation is encouraging, and it has shown that Leogo is not unusable. Claims of usability, utility, and pedagogical value, however, would be imprudent without extensive further evaluation. Further evaluation, then, is the major focus of our work to be carried out in the coming year.

Several systemic developments are also planned, described below.

6.1 Collaborative programming

We are interested in making Leogo group-aware, so that it explicitly supports and reinforces the benefits of peer-learning and of collaborative problem solving [8]. We have extensive experience with collaborative software developed in GroupKit [30], including Turbo-Turtle, a collaborative micro-world for exploring Newtonian physics [3]. We do not foresee any major obstacles in making Leogo fully collaboration-aware. Given the value of peer learning, it is beneficial for almost any piece of educational software to be made collaboration aware. Equal-opportunity interfaces, however, seem particularly appropriate due to the multiple representation and expression mechanisms which cater to personal preference and varied ability ranges.

6.2 Equal opportunity history mechanisms

Currently, only the text-based programming environment provides an interactive history mechanism. We will extend the history mechanisms to the other two environments and maintain equal opportunity between them.

The turtle's path in the direct manipulation environment essentially provides a history of past commands. We will make the turtle's trail interactive, allowing the user to select "unit" portions of the path. Each "unit" will correspond to a single user action (a procedure call, for instance). Consequently, if the user clicks on an edge that was generated by a call to the procedure `square` then the entire square will be selected. Users will be able to increase the range of contiguous history commands by clicking the forward and back buttons.

[4]Logo Computer Systems, Inc.

In the iconic programming environment we will add a "stamp sheet" that shows iconic representations of the users past actions. Contiguous portions of the stamp-sheet will be selectable.

The portion of history selected in one environment will be reflected in all environments. The user will be able to encapsulate selected histories within procedures, repeat loops, conditionals, or execute them directly.

6.3 Increased dynamic equal opportunity

Section 4.1.3 noted the problems that users had in predicting the effects of iconic programming actions due to the "confusing" numerical values. A solution to this problem would be to dynamically show the effect that the action *would* have once completed. For instance, while dragging the parameter slider for `Forward` to larger values, the user would dynamically see the turtle's resultant position and trail (if the pen was down). Similarly the turtle's heading could be dynamically adjusted to show the effect of releasing the slider with the current parameter setting.

This dynamic style of equal opportunity already exists when the user expresses actions in the direct manipulation environment. The user sees the amounts of motion and rotations dynamically displayed in the iconic environment's parameter sliders and in the text-environment. We did not, however, implement dynamic updates in the iconic environment because we were concerned about interface consistency. The Leogo "primitive" procedures `Forward`, `Back`, `Left`, and `Right` would operate dynamically, while procedures that the user defined would not. Dynamic updates cannot be provided for user-defined procedures because executing the procedures may take too long.

At future trials of the system we will support dynamic equal opportunity for the Leogo primitive commands at the expense of interface consistency, and see whether this causes user problems.

6.3.1 Inferencing programs by example

Myers [23, 22] distinguishes systems that support "programming-by-example" from systems that allow "programming-with-example". Systems for programming-by-example autonomously *infer* generalisations in the the user's actions. Programming-with-example systems, however, require explicit instruction from the user at all stages. Leogo therefore, only supports programming-with-example.

Several times we have been technologically tempted to incorporate programming-by-example features within Leogo. Despite our temptations, we have not done so for two reasons. First, we strongly believe that it would be pedagogically invalid to incorporate inferencing mechanisms within a teaching and learning environment of this kind. The effect would be to replace some of the student's learning (from the process of specifying precise actions) with machine learning through autonomous inferencing. Second, part of the research motivation for Leogo is to forcefully demonstrate equal opportunity interfaces within programming environments. Although it is likely that equal opportunity interfaces could be used to reveal the results of "internal" program expressions (the system's inferences), the value of such interfaces, and the mechanisms used to reveal them are beyond the scope of this work—Maes [19] describes somewhat related work on interfaces to the inferences made by autonomous agents.

7 Summary

Leogo allows children to experience a variety of programming paradigms within a single system. The equivalence and equal opportunity between these paradigms is intended to reinforce understanding of alternative styles of programming. Our observations of young students interacting with Leogo are encouraging, and we intend to continue development and evaluation. We believe that equal opportunity interfaces are useful in broad ranging programming environments, and we hope to motivate further work in this area.

Availability

Leogo is written in Tcl/Tk [26], and is available on request from the first author.

Acknowledgements

Thanks to Alan Cypher, Saul Greenberg, and Steve Jones for their comments on Leogo. Thanks also to the anonymous paper reviewers for their helpful and constructive comments.

References

[1] Apple Computer Inc. *Human Interface Guidelines: The Apple Desktop Interface.* Addison-Wesley, 1988.

[2] MH Brown and R Sedgewick. Techniques for algorithm animation. *IEEE Computer*, 2(1):28–39, 1985.

[3] A Cockburn and S Greenberg. Turbo-turtle: A collaborative microworld for exploring newtonian physics. In *ACM Conference on Computer Supported Cooperative Learning (CSCL '95). Bloomington, Indiana. October 17–20, 1995*, pages 62–66. Lawrence Erlbaum Associates, Inc, October 1995.

[4] G Coulouris and H Thimbleby. *HyperProgramming.* Addison-Wesley, 1992.

[5] N Cunniff, RP Taylor, and JB Black. Does programming language affect the type of conceptual bugs in beginners' programs? a comparison of fpl and pascal. In *Human Factors in Computing Systems III. Proceedings of the CHI'86 conference.*, pages 175–182. Amsterdam; North Holland/ACM, 1986.

[6] A Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of CHI'91 Conference on Human Factors in Computing Systems* New Orleans, May, pages 33–39, 1991.

[7] A Cypher, editor. *Watch what I do: programming by demonstration.* MIT Press, 1993.

[8] DC Edelson, RD Pea, and LM Gomez. The collaboratory notebook. *Communications of the ACM*, 39(4):33–34, 1996.

[9] EP Glinert and L Tanimoto. Pict: An interactive graphical programming environment. *IEEE Computer*, 17:7–25, 1984.

[10] JD Gould and C Lewis. Designing for usability: Key principles and what designers think. *Communications of the ACM*, 28(3):300–309, 1985.

[11] M Hardie. Logo meets KidPix, 1994. M.Sc. Thesis. Department of Computer Science. University of Canterbury, Christchurch, New Zealand.

[12] T Hogan. The perfect spreadsheet. *MacUser*, pages 225–258, August 1991.

[13] K Kahn. Drawing on napkins, video game animation, and other ways to program computers. *Communications of the ACM*, 39(8):49–59, 1996.

[14] K Kahn. ToonTalk—an animated programming environment for children. *Journal of Visual Languages and Computing*, 7(2):197–217, 1996.

[15] K Kahn and VA Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the IEEE Visual Language Workshop. Skokie, Illinois.*, pages 7–15, 1990.

[16] H Kato and A Ide. Using a game for social setting in a learning environment: *AlgoArena — a tool for learning software design. In ACM Conference on Computer Supported Cooperative Learning (CSCL '95). Bloomington, Indiana. October 17–20*, pages 195–199. Lawrence Erlbaum Associates, Inc, October 1995.

[17] Logo computer systems inc. `http://www.lcsi.ca/`, 1997.

[18] H Lieberman. Tinker: A programming by demonstrations system for beginning programmers. In A Cypher, editor, *Watch What I Do: Programming By Demonstration*, pages 46–64. MIT Press, 1993.

[19] P Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, 1994.

[20] DL Maulsby and IH Witten. Inducing programs in a direct manipulation environment. In *Proceedings of CHI'89 Conference on Human Factors in Computing Systems* Austin Texas, April, 1989.

[21] M Moriconi and DF Hare. Pegasys: A system for graphical explanation of program designs. *SIGPLAN Notices: Proceedings of ACM SIGPLAN Symposium on Langauge Issues in Programming Environments*, 20(7):148–160, 1985.

[22] B Myers. Taxonomies of visual programming and program visualizations. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[23] BA Myers. Demonstrational interfaces: A step beyond direct manipulation. In D Diaper and N Hammond, editors, *People and computers VI. Proceedings of the HCI '91 Conference. 20–23 August. Edinburgh.*, pages 11–30, 1991.

[24] I Nassi and B Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8):12–26, 1973.

[25] C O'Malley, S Draper, and M Riley. Constructive interaction: A method for studying human-computer-human interaction. In *Interact '84. The First International Conference on Human-Computer Interaction.* London, UK. 4–7 September., pages 269–274, 1984.

[26] JK Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1993.

[27] S Papert. *Mindstorms — Children, Computers, and Powerful Ideas.* Harvester Press, Brighton, 1980.

[28] S Papert. *The Children's Machine: Rethinking School in the Age of the Computer.* Basic Books, 1993.

[29] S Reiss. Graphical program development with pecan program development systems. *SIGPLAN Notices: Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium of Practical Software Development Environments*, 19(5):30–41, 1984.

[30] M Roseman and S Greenberg. Building real time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, 1996.

[31] D Smith. Pygmalion: A creative programming environment. Technical Report STAN-CS-75-499, Department of Computer Science, Stanford University, 1975.

[32] DC Smith, A Cypher, and K Schmucker. Making programming easier for children. *Interactions*, 3(5):58–67, 1996.

[33] DC Smith, A Cypher, and J Spohrer. Kidsim: Programming agents without a programming language. *Communications of the ACM*, 37(7):55–67, 1994.

[34] I Sommerville. *Software Engineering*. Addison-Wesley, fourth edition, 1992.

[35] H Suzuki and H Kato. Interaction-level support for collaborative learning: *AlgoBlock* — an open programming language. In *ACM Conference on Computer Supported Cooperative Learning (CSCL '95). Bloomington, Indiana. October 17–20*, pages 349–355. Lawrence Erlbaum Associates, Inc, October 1995.

[36] H Thimbleby. *User Interface Design*. ACM Press, Addison-Wesley, 1990.

[37] H Thimbleby, A Cockburn, and S Jones. Hypercard: An object-oriented disappointment. In P Gray and R Took, editors, *Building interactive systems: architectures and tools*, pages 35–55. Springer-Verlag, 1992.

Figures

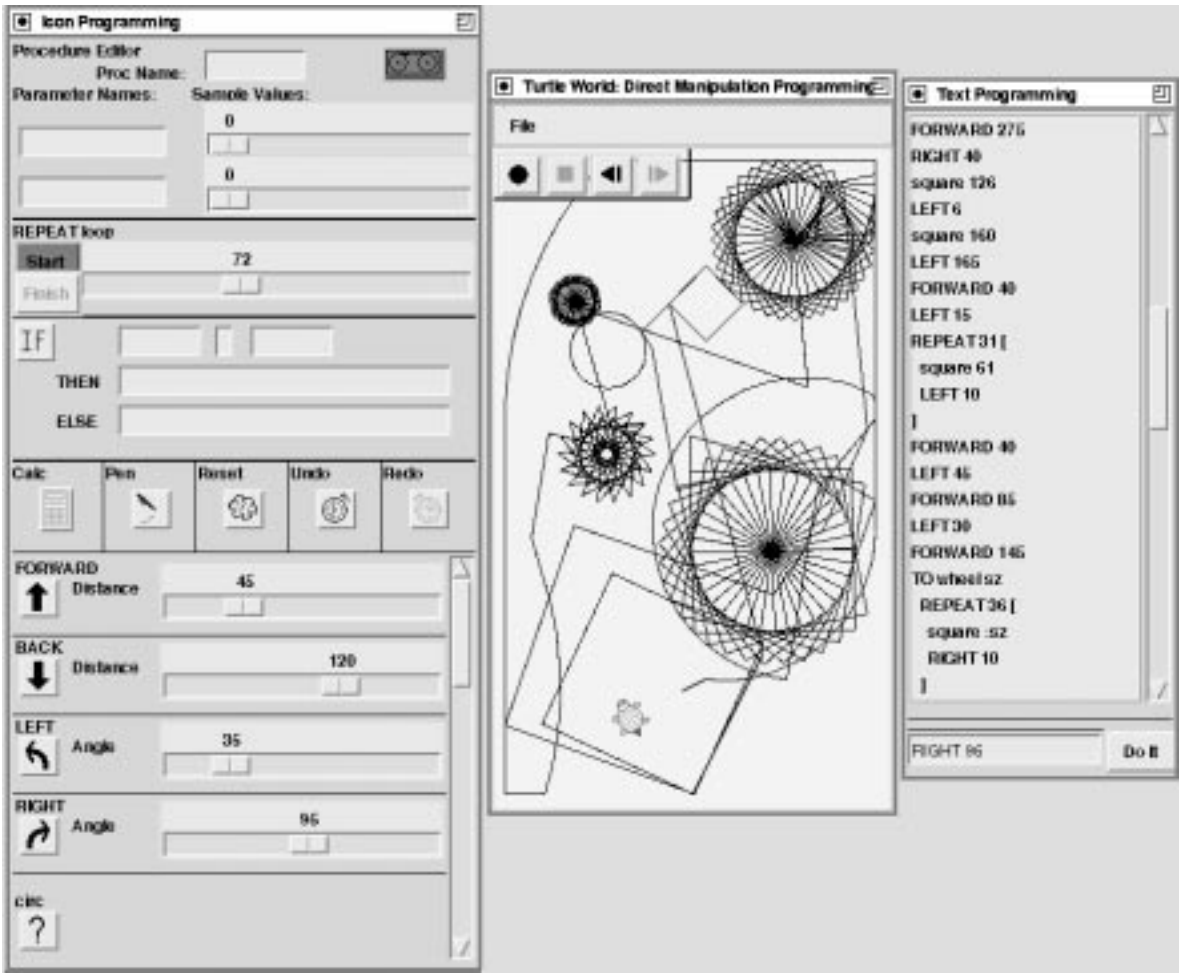

Figure 1: Leogo's three programming paradigms: iconic, direct manipulation, and textual.