A Programmer's Interface:
A Visually Enhanced and Animated Programming Environment

Ronald Baecker
Dynamic Graphics Project, Computer Systems Research Institute, University of Toronto
Toronto, Ontario, M5S 1A4, Canada
baecker@dgp.toronto.edu, (416) 978-6853, FAX: (416) 978-4765

John Buchanan
Department of Computer Science, University of British Columbia
Vancouver, British Columbia, Canada

Abstract
This paper describes LOGOmotion, a programming environment that is visually enhanced, animated, unobtrusive, extensible, and monomorphic. This means that the programmer can obtain with ease visual and animated presentations of the execution of programs written in the LOGO language. She can also define new methods of program presentation by writing visualization procedures in LOGO but without altering the original program code. Methods may include classical debugging tools such as traces as well as more modern visualization aids such as the animation of program behavior. The design goals and fundamental principles underlying the system are discussed. Several illustrations of its use are presented, as is a brief description of the implementation. The paper concludes with an evaluation of the system as a programmer’s interface that is easy to learn, easy to use, flexible, and powerful.

Keywords
User interface design, human factors of computing systems, programmer’s interface, programming environments, programmer productivity, software engineering, debugging aids, program visualization, program animation, algorithm animation, computer graphics, computer animation, graphics applications.

Introduction
Programs are complex processes which exhibit behavior over time. Programmers employ diagrams and pictures as representations of program behavior, to help themselves visualize and master this complexity. They also hand simulate the execution of their programs through time, in effect constructing an animation of what the program is doing.

Unfortunately, most programming environments transact primarily in text and numbers, rarely in diagrams and pictures, and primarily in static displays, rarely in animation. They therefore fail to provide optimum support to their users. This paper describes LOGOmotion, a new environment for the LOGO programming language that is designed to enable programmers to produce useful pictures and animations of their programs’ behavior with a minimum of distraction and a minimum of effort.

Survey of Relevant Work
Program visualization systems have existed since the earliest days of computer science. The earliest systems were interactive assembly language control programs (see Crossey, 1977). The advent of higher-level languages led to a variety of higher-level debugging tools (Evans and Darley, 1966; Crossey, 1977; Sun, 1986) and automatic flowchart generators (Abrams, 1968). Beginning in the 60’s, a few individuals began working on the animation and illustration of algorithms and programs (Knowlton, 1966; Baecker, 1973; Hopgood, 1974). Several projects were carried out at the University of Toronto (Chan, 1974; de Boer, 1974; Baecker, 1975; Yarwood, 1977), eventually resulting in the dramatic and effective teaching film Sorting out Sorting (Baecker, 1981). More recently, the introduction of powerful graphical workstations has sparked the development of real-time, interactive, program visualization systems, a few of which we now describe.

Animus (London and Duisberg, 1985; Duisberg, 1986a, 1986b) developed at Tektronix, is a system designed to study the applicability of constraints to the development of process animations. Visualizations are developed by specifying a set of constraints between program execution events and some display procedures. These constraints are defined by selecting a message in the program’s code, and an object which will receive the message. In cases where there are no suitable messages, the programmer can insert code which will generate them. The satisfaction of these constraints causes the generation and display of the animation. The display procedures are specified either by using some existing display code, or by writing new code in the Smalltalk programming language.

Balsa-II (Brown and Sedgewick, 1984; Brown, 1988a, 1988b), developed at Brown University, is a system that provides the programmer with a set of tools with which interactive program movies can be produced. These movies illustrate the program’s execution using a variety of dynamic graphic views. The views are generated by altering the program code to call the display procedures. The execution of the program generates a detailed script which is then used to generate the program-movie. The system allows the user to change both the input to the program and the manner in which it is displayed during the replay of the movie.

Movie and Stills (Bentley and Kernighan, 1987), developed at AT&T Bell Labs, is a system which allows the programmer to develop simple animations of arbitrary programs. Visualizations are constructed by inserting, in the source code, state-
ments that print animation commands to a file. The resulting script file is then processed by Movie and Stills to produce the animation. The simplicity and versatility of this system ensures that it can be used by programmers in all stages of program development. One of its weaknesses is the requirement that the source code of the program be altered to produce the visualization.

The Transparent Prolog Machine (TPM) (Eisenstadt and Brayshaw, 1987), developed at the Open University, is a powerful Prolog debugger which was developed to provide "an augmented AND/OR trees representation of logic programs". The system offers the user a variety of visual forms which aid in debugging. Execution of a Prolog program can be characterized by the manner in which the interpreter attempts to satisfy the program clauses. TPM uses nodes to represent the clauses of the program, each of which contains graphical information indicating its status. There are symbols to indicate that the clause has been satisfied, that the clause is being examined, and that resolution of the clause has been attempted previously. Each node also has a graphical abbreviation that is used when the execution space of the program is large.

Reviewing these systems and a number of others (Herot et al., 1982; Baecker and Marcus, 1983, 1986, 1990; Lieberman, 1984; Reiss, 1984, 1985, 1987; Brown et al., 1985; Raeder, 1985; Dewar and Clew, 1986; Myers, 1986; Chang, 1987; Isoda, Shimomura and Ono, 1987; Baecker, 1988) has allowed us to draw a number of conclusions (Baecker, Price, Small, and Buchanan, in preparation):

- Few systems provide the programmer with the ability to animate arbitrary programs; most are restricted to programs in a particular language or a particular application area.
- These restrictions do have one advantage, for it becomes possible to incorporate special purpose knowledge to take advantage of the specific nature of the visualizations desirable for that host language or application area.
- In some of these systems, visualizations include animated as well as static displays.
- A few systems allow one to program arbitrary visualizations; in most systems, the set of display methods is fixed or only tailorable in minor ways.
- Sometimes one can use the same language to request or define the visualizations as that being used for the program's development (the host language), although sometimes the user must learn a completely different language.
- Finally, the programmable systems typically require alteration of the source code to produce these displays.

We summarize these insights in Table 1. Included are the four systems reviewed above, as well as LOGOmotion.

### Table 1: Comparison of program visualization systems

<table>
<thead>
<tr>
<th>Domain of use</th>
<th>Animus</th>
<th>Balsa II</th>
<th>Movie &amp; Stills</th>
<th>TPM</th>
<th>LOGOmotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special-purpose knowledge and built-in visual enhancements</td>
<td>some</td>
<td>very</td>
<td>none</td>
<td>many</td>
<td>some</td>
</tr>
<tr>
<td>Animation possible</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Extensible: Visualizations can be programmed</td>
<td>yes</td>
<td>yes</td>
<td>user</td>
<td>tailorable</td>
<td>yes</td>
</tr>
<tr>
<td>Monomorphic: Host language used to define visualizations</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Unobtrusive: Visualizations requested and defined without altering source code</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Design Goals**

The implementation of LOGOmotion was motivated by five goals. The system should include a rich set of defaults so that the programmer can obtain useful visualizations of her code with essentially no effort. These visualizations should include animation of the program behaviour. It should also provide her with the ability to produce specifically crafted visualizations by writing programs. This programming should be in the host language of the system, not in some other language. No alterations of the source code should be required.

A visually enhanced environment with a rich set of default visualizations

The system should provide the programmer with default views of events which are not immediately visible, such as procedure events and variable-update events. A quick view of a program can be obtained, which can later be refined if so desired.

**An animation capability**

Built-in views and programmable views (see below) should include both static and dynamic (animated) representations of program behaviour.

**Extensible: fully programmable views**

Many of the systems cited above allow the programmer to enhance the execution of the program with some restricted class of views that can be requested declaratively. If the pro-
grammer is provided with a fully programmable visualization language, the class of views which can be generated will be greatly enriched.

**Monomorphic: View programming in host language**

Even though some existing systems provide programmable control of the visualization, many require that this be done using a different language. The programmer should be able to use the host language for this purpose.

**Unobtrusive visualization**

A visualization system which requires that the programmer alter the source code of the program will restrict the manner in which it is used. Imagine that the programmer wants to view some events or occurrences in the program before it is fully developed. Since the programmer must alter the source code, she will save a version of the unaltered code and then proceed to make the required visualization changes. After viewing the visualization, she may realize that there is some part of the code which requires fixing. Now she can either alter the current version and then later alter the original code, or can alter the original code and re-introduce the code which produces the visualization. Either decision requires that work be needlessly repeated.

An obtrusive system allows multiple users of a program to develop their own individual visualizations of the code. Visualizations and animations can be constructed by animators who are not programmers or who are not familiar with the original source code.

**Why LOGO?**

Our conjecture was that satisfying these five goals would allow us to produce a system that was easy for programmers to learn, easy for them to use, flexible, and powerful. To test this conjecture, we needed a host language that has a simple syntax, is extensible, and is widely used. A suitable syntax would simplify the implementation. Extensibility of the language would allow us to add controlling primitives for the visualizing components of the system. Popularity of the language would provide us with a large body of test programs. LOGO (Harvey, 1986a, 1986b, 1987; Clayson, 1988; Goldenberg and Feurzeig, 1987) is a language that meets all these requirements.

**LOGOMotion Concepts**

LOGOMotion consists of a standard implementation of LOGO enriched by a set of commands for monitoring events and mapping them into views which appear in windows.

**Events and views in LOGOMotion**

The execution of a program is characterized by a set of events such as procedure invocations, statement executions, and token parsing. If visual representations of these events are constructed, the programmer may be able to abstract information about the execution of the program from the generated displays. LOGOMotion provides tools for the display of four different kinds of events. These are variable-updates, procedure-invocations and procedure-returns, input and output operations, and turtle actions. (The turtle is a mechanism which is used to provide graphical output in LOGO.) Each event class has a set of properties which characterize individual events in the set.

These classes of events were chosen because they provide a rich set of hooks onto which animations and visualizations can be attached. LOGO is a functional language, thus the ability to view procedure calls and returns is vital. Of almost equal importance are changes to values of the program variables. Finally, although input-output including turtle movements can be seen during program execution, construction of meaningful visualizations also requires hooks on the I/O events.

A variable-update event occurs when a new value is assigned to a variable. The properties that characterize such an event are the name of the variable, the value it contained prior to the assignment, and the value assigned to it.

The procedure-invocation event occurs when all parameters have been assigned their values, and the procedure code is about to be executed. The properties that characterize such an event are the name of the procedure and the values assigned to its parameters.

The procedure-return event occurs when the procedure has finished execution and control is about to return to the calling procedure. The properties that characterize such an event are the name of the procedure and the value that it returns, if any.

The input event occurs when a LOGO object is received as input from the I/O window. The property that characterizes such an event is the LOGO object that was input.

The output event occurs when a LOGO object is output to the I/O window. The property that characterizes such an event is the LOGO object that was output.

The turtle event occurs when a turtle executes a command. The property which characterizes such an event is the command that was executed.

**Windows in LOGOMotion**

Standard LOGO implementations provide the programmer with an I/O window and a turtle window. All interaction between the programmer and the interpreter occurs in the I/O window, and the turtle window is used for the turtle actions. LOGOMotion allows the programmer to create and use additional turtle windows. When any of these windows is selected as the active turtle window, the standard turtle commands will manipulate the window's turtle. The availability of several turtle windows allows the programmer to keep the execution of a program separate from its visualization.

**Adding and controlling views**

The implementation of LOGOMotion has provided the programmer with the ability to enhance events with visualizations. We call each of these enhancements a view.

The programmer can control the manner in which views are attached to events by using extended LOGOMotion commands. We now present an overview of these commands.

For each event class three primitive procedures were added to the language. The first allows the programmer to add views to selected events and has the format:

```
[VAR|PROC|TURTLE]VIEW <Windows> <Viewer> <List of events>
```
The second removes views from the specified events and has the format:

\[ \text{NVARPROCTURTLEVIEW } \langle \text{Viewer} \rangle \langle \text{List of events} \rangle. \]

The third procedure allows the programmer to request the status of the viewers of an event class. This has the format:

\[ \text{NVARPROCTURTLEVIEWED} \]

and returns a LOGO list containing the required information.

Examples and Illustrations

The four examples that we present here are intended to illustrate the manner in which programs executions can be enhanced by attaching views to different events. The first example is a LOGOMotion-provided stack view of procedure invocations and returns in a solution to the Towers of Hanoi problem. Next we explain how the user can define her own trace view which integrates a list of procedure calls with the input-output from the Towers of Hanoi. The third example introduces a user-defined animation of the Towers of Hanoi solution. Finally, we present a series of illustrations of the recursive computation of a fractal curve.

A stack view of procedure calls

We will use a standard solution to the Towers of Hanoi problem (Figures 1,2) to illustrate the LOGOMotion-provided stack view (Figure 3). The LOGO code appears in Figure 1, the output in Figure 2. The code is presented using the principles of enhanced typographic visualization of source code discussed in Baecker and Marcus (1990).

```logo
TO Hanoi :NUMBER FROM :TO OTHER
\% A standard solution to the Towers of Hanoi problem
\% IF NOT EQUAL :NUMBER 0 \{ STOP \}
MoveDisc :NUMBER FROM :TO
Hanoi DIFFERENCE :NUMBER \{ OTHER TO \} FROM
END
```

```logo
TO MoveDisc :NUMBER FROM :TO
\% Move the disc from the FROM post to the TO post
\% TYPE \# Move Disc
\% TYPE \# NUMBER
\% TYPE \# FROM
\% TYPE \# TO
PRINT TO END
```

Figure 1: A solution to the Towers of Hanoi problem

```
Hanoi 3 "A" "B" "C"
Move Disc 1 from A to B
Move Disc 2 from A to C
Move Disc 1 from B to C
Move Disc 1 from A to B
Move Disc 1 from C to A
Move Disc 2 from C to B
Move Disc 1 from A to B
```

Figure 2: The output from the solution to the Towers of Hanoi problem

Figure 3: The stack view after partial execution of the Hanoi solution
A user-defined trace of procedure calls

Traditional LOGO implementations have provided the user with a tool that allows them to trace procedure invocations and exits. In LOGOmotion the default trace facility is not provided as a primitive, but instead may be built using the system. In this example we present the LOGO code that implements this TRACE tool (Figure 4) and some example output produced by it (Figure 5). The trace in Figure 5 corresponds to the same point in the program's execution as does the stack in Figure 3.

This example illustrates how the LOGOmotion programmer can quickly enhance the visualization of a program by attaching views to events. The trace is generated by writing a simple LOGO program but without altering the source code of the two original procedures.

```
TO Trace :NAME :PARAMETERS :CALLED
  IF :CALLED
    [PrintEntry :NAME :PARAMETERS]
  END

TO PrintEntry :NAME :PARAMETERS
  TYPE :NAME
  NPE 1
  LOCAL :TMP
  MAKE :TMP :PARAMETERS
  REPEAT :COUNT :PARAMETERS
    [TYPE LAST FIRST :TMP]
    [TYPE "]
  MAKE :TMP IF :TMP
END
```

**Figure 4:** The LOGO code for a trace utility

```
>PROCVIEW "TO Trace [Hanoi MoveDisc]
>  Hanoi 3 A B C
  Hanoi 2 A C B
  Hanoi 1 A B C
  MoveDisc 1 A B
  MoveDisc 1 from A to B
  Hanoi 0 C B A
  MoveDisc 2 A C
  MoveDisc 2 from A to C
  Hanoi 1 B C A
  Hanoi 0 B A C
  MoveDisc 1 B C
  etc.
```

**Figure 5:** A partial trace view of the Hanoi and MoveDisc procedures

**Animation of the Towers of Hanoi solution**

An obvious animation of the Towers of Hanoi problem will show the three posts and the discs on them. As the program runs, the discs will move from post to post, thus dynamically illustrating program execution. Figure 6 shows some stills from an animation generated in this manner.

In the example shown in Figures 4 and 5, a trace view is created by performing a set of actions each time the procedures Hanoi and MoveDisc are called. The animation in Figure 6 is generated by attaching separate views to Hanoi and MoveDisc. The view attached to Hanoi is responsible for drawing the posts and the initial stack of discs and for initializing the variables which are used by the animation procedures. The view attached to MoveDisc is responsible for erasing and redrawing the discs each time the MoveDisc procedure is called. The top-level procedures are shown in Figure 7.

**Annotations of a Snowflake curve computation**

An elegant use of the LOGO programming language is a recursive implementation of a fractal Snowflake curve. We introduce a program that computes one third of the Snowflake to various recursively defined levels (Figure 8). We then illustrate how LOGOmotion can be used to help elucidate the algorithm's recursive nature with a series of views that annotate the curve. Each of Figures 9 through 14 shows a different level of the recursive computation with a slightly different view.
Figure 6: A graphical view of Hanoi and MoveDisc

HanoiView

TO HanoiView NAME PARAMETERS CALL
  % Procedure to view the invocation of the HANOI procedure
  LOCAL "FROM LOCAL "DISCS
  MAKE "FROM LAST FIRST BUTFIRST PARAMETERS
  MAKE "DISCS LAST FIRST PARAMETERS
  MAKE "POSTTOTAL 0
  MAKE "POSTTOTB 0
  MAKE "POSTTOTA 0
  MAKE "POSTS 100-75
  MAKE "POSTS 0-75
  MAKE "POSTS 100-0
  MAKE "POSTS [0, 255]
  MAKE "COLOUR [0, 255]
  MAKE "POSTS [0, 255]
  MAKE "DISCS [0, 255]
  DRAWPOSTS FROM DISCS
  NOPROCVIEW 'HanoiView [Hanoi]
END

MoveDiscView

TO MoveDiscView NAME PARAMETERS CALL
  % The first parameter to MoveDisc is the number of the disc to be moved
  % The second parameter to MoveDisc is the post from which the disc is to be moved
  % The last parameter to MoveDisc is the post on which the disc is to be placed.
  % IF NOT CALL [STOP]
  % Take no action on return
  LOCAL "DiscNumber
  MAKE "DiscNumber LAST FIRST PARAMETERS
  LOCAL "FromPost
  MAKE "FromPost LAST FIRST PARAMETERS
  LOCAL "ToPost
  MAKE "ToPost LAST LAST PARAMETERS
  RemoveDisc DiscNumber FromPost
  AddDisc DiscNumber ToPost
END

Support routines including DrawPosts, RemoveDisc, and AddDisc omitted here.

>PROCVIEW 'HanoiView [Hanoi]
>PROCVIEW 'MoveDiscView [MoveDisc]
> Hanoi 3 'A 'B 'C

Figure 7: How Figure 6 was produced using LOGOmotion
The LOGOMotion code for Flakeside is:

```
TO Flakeside Level Length
IF等于 Level 0 [[FORWARD :Length] [STOP]]
Flakeside DIFFERENCE Level 1 QUOTIENT Length 3
RIGHT 60
Flakeside DIFFERENCE Level 1 QUOTIENT Length 3
LEFT 120
Flakeside DIFFERENCE Level 1 QUOTIENT Length 3
RIGHT 60
Flakeside DIFFERENCE Level 1 QUOTIENT Length 3
END
```

Figure 8: LOGOMotion code for Flakeside

In the first view, the Flakeside procedure is called with the parameters 0 and 640. Since this is the simplest possible call, a single line is drawn and the program finishes. The line is annotated with the name of the procedure and the two arguments with which it is called.

The view of “Flakeside 2 640” in Figure 11 omits the length parameter of the procedure invocations, since the viewer should be able to see that the length is divided by 3 at each invocation.

Figure 9: Zero order view of “Flakeside 0 640”

Figure 10 illustrates a different view of “Flakeside 1 640”. In this view only the arguments to the calls are shown in relationship to the position of the turtle; the procedure names are omitted.

Figure 11: Second order view of “Flakeside 2 640”

The view of “Flakeside 3 640” in Figure 12 is further simplified, since we observe that, when a procedure is invoked with a level value of n, there are n procedure calls corresponding to the level values n-1,n-2,...,0. Thus we annotate only with the highest level at each point.

Figure 10: First order view of “Flakeside 1 640”

Figure 12: Third order view of “Flakeside 3 640”
As the complexity of the curve increases, illustrating the level numbers with digits results in an increasingly cluttered picture. The view of "Flakeside 4 640" in Figure 13 represents the level at each point with a circle whose radius indicates its value.

The final view in Figure 14 uses the identical "fourth order" representation to illustrate the execution of a call to "Flakeside 5 640".

The implementation

LOGOmotion comprises three main modules, the parser, the interpreter and the animator. The parser parses the input from the user and produces a structured internal representation of the input. The interpreter interprets this internal representation. The animator manages the display of the selected events. The implementation of the parser and the interpreter follow standard compiler/interpreter techniques, and will not be discussed here.

The information required by the animator for each event include the event's viewers, and the windows in which each viewer generates its display. This information is kept in a LISP property list. The animator also keeps track of the events being recorded. When the animator is required to generate a view, it saves the run-time environment and invokes the view generating procedures. After these finish, the environment is restored and control is returned to the interpreter. The interpreter and the animator communicate by means of a simple set of messages.

The interpreter sends two classes of messages to the animator. The first class informs the animator that an event has occurred in the execution of the program, while the second set controls the manner in which the program is displayed.

Messages informing the animator that an event has occurred are directly linked to the four event classes introduced above. Upon receiving a particular kind of message, the animator checks to see if the event has any viewers, in which case the animator ensures that each viewer produces its display.

Messages requesting that the animator alter the program display are sent when the interpreter encounters one of the view-controlling extensions. The animator changes its data structures so that they reflect this change. If the interpreter has requested some information from the animator, this information is pushed onto the interpreter's execution stack. Control is then returned to the interpreter.

Experience with the implementation

The message system between the interpreter and the animator, and the simple data structures which the animator keeps, enable the two modules to function relatively independently. Adding a new LOGO primitive only requires 10-20 minutes programming time. Testing of new ideas in the animator is a simple matter. Unfortunately, though, the system runs very slowly, and cannot easily be ported to more modern machines, so a redesign and reimplementation is underway.

One use of program visualization is for teaching. Program animation can convey aspects of a program's execution which static representations cannot. Our new implementation of LOGOmotion will enable us to produce instructional program movies. This system will assist in enhancing the replay of program executions with a variety of cinematographic effects. This will include adding high-quality digital typography and colour, combining multiple views in a single scene, controlling timing, adding narration and sound effects, and adding special effects such as pans, zooms, wipes, fades, and dissolves.
Future research problems

Now that we have attacked the problem on LOGO, there remains the question of how to apply this program visualization methodology to other languages. Although implementation is easier for an interpretive language, modern compilers maintain sufficient information to allow the implementation of a similar system.

In developing an event driven, program visualization system, the selection of events will determine the success or failure of the system. Further use of LOGOmotion will determine whether our choice of events is necessary and sufficient, and will hopefully guide the design of suitable event classes for other procedural languages as well as those that are non-procedural.

Contributions, Summary, and Conclusions

We have presented LOGOmotion, a novel conversational programming environment which allows the graceful construction of a wide variety of powerful debugging and visualization aids. These may be achieved within the host programming language but without altering the host program. LOGOmotion represents a unique and important point in the design space of program visualization systems (Baecker, Price, Small, and Buchanan, in preparation), a point which previously had not been explored.

This point in the design space has some valuable properties for the programmer. LOGOmotion's unobtrusive nature and built-in visual enhancements make it easy to use. Because the system is monomorphic, it is easy for a LOGO programmer to learn. Finally, because LOGOmotion is visually enhanced, animated, and extensible, it is flexible and powerful.

In summary, LOGOmotion provides a programming environment with a congenial and powerful programmer's interface.

Acknowledgements

This research was partly funded by the Natural Sciences and Engineering Research Council of Canada and partly by the Information Technology Research Centre of the Province of Ontario. We are also grateful to Xerox Canada and Apple Computer for their generous support, and to Alan J Rosenthal, Jan Venus, and Avi Naiman for their assistance.

References


