LEARNING GRAPHICS PROGRAMMING BY DIRECT COMMUNICATION

Martin Tuori
Tim Pointing
Defence and Civil Institute of Environmental Medicine
PO Box 2000
Downsview, Ontario, M3M 3B9

ABSTRACT

The process of learning the graphics functions of a computer graphics workstation environment is both assisted and hampered by the presence of an intermediary programming language. Assistance comes in the form of programming language functions for preprocessing, storage declaration, expression evaluation, control flow, and system libraries. Working against the student, compilation of programmed examples is slow, and errors may arise both from the syntax and semantics of the graphics functions, and from those of the programming language.

We propose an approach to learning the graphics functions that temporarily separates the graphics component from other aspects of the overall programming environment; in a sense, we are proposing training wheels for the graphics subsystem.

This approach was used in creating, for the IRIS series of workstations, [1] a graphics interpretter that allows a student to test out graphics concepts, without the need to write programs. Subroutine calls typed to the interpreter are carried out immediately, allowing a quick, trial-and-error approach. We argue that this approach is a useful addition to conventional learning techniques, and that its success can be attributed to bringing the student programmer into more direct communication with the graphical components of the programming environment.

1. IRIS is a trademark of Silicon Graphics Inc.

INTRODUCTION

A student learning the details of a new computer graphics programming environment may employ many different techniques. He may begin by reading the manufacturer's documentation, which, if well written, conveys basic concepts, syntax, semantics and suggestions for efficient use of the computer graphics system. While this is an important stage in the student's training, it is not enough to give him fluency as a graphics programmer. Writing small test programs is good way to proceed, and is made much easier if a sample skeleton program, or stub, is provided. As Duff says, ''Whenever possible, steal code.'' [Duff 1985]. The student can extend the stub to exercise individual features of the graphics environment, or combinations of features, thereby gaining familiarity with the concepts and behaviour of the system.

A high-level programming language provides a variety of features that can help the student in his exploration of a system and its functions. Macro preprocessing serves two useful functions — common constants and expressions are provided in system files, for inclusion in new programs, and the student can define macros to suit his own needs. Storage declaration provides for complex object definition, and for loading them from external sources. Expression evaluation allows results from one operation to be used as input to another; for example, reading pixel values from a raster display, modifying and redisplaying them. Features for control flow allow conditional, iterative and recursive action. Finally, various support libraries for mathematical, input/output, networking and other functions offer specific functionality, as needed. Although the student can defer the use of some language features, such as specialized subroutine libraries, he cannot avoid the basic syntax and semantics of the programming language itself.

Many of the basic language features are of little interest, initially, to the student studying a graphics subsystem; rather, he needs to concentrate on the graphics subroutine calls. The *programming approach* is error-prone, tedious and time-consuming. The student's efforts at writing even small, correct programs are invariably delayed by errors in the syntax or semantics of the programming language; these must be corrected by repetitive editing, compilation and testing.

These problems arise because the programming approach is *indirect*. The student needs to test his skills in using the graphics functions, but is forced to do so through an intermediary, albeit high-level, programming language (Figure 1). If the programming language is interpretive (some implementations of Basic, Lisp, APL, etc.), test runs can proceed quickly; in many cases, however, the programming language is compiled (most implementations of C, Pascal, Fortran, etc.), and considerable time is spent waiting for compilation to take place. Since the student's efforts are highly exploratory, characterized by tens or hundreds of trial and error steps, considerable time and system resources may be wasted.

A DIRECT INTERPRETIVE APPROACH

Recent literature on Human-Computer Interaction (HCI) has promoted the use of *direct manipulation* [Shneiderman 1983], [Kay 1984], [Hutchins, Hollan and Norman 1986], [Witten and Greenberg 1985]. Shneiderman characterizes direct manipulation by: the visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest.

The situation here is different, in that there is no easily defined *visible object of interest*. The student is studying the process, or language of graphics programming. Perhaps the conventional approach, in which we use a complex command language (the programming language interface) should ultimately be replaced by more more visual, manipulative, or demonstrative programming methods. We are somewhat constrained, however, by the present state of programming support on graphics workstations; the student must learn to control a graphics system through a highly linguistic interface. Our objective here is not to introduce direct manipulation, but to offer *direct communication* between the programmer and the graphics library.

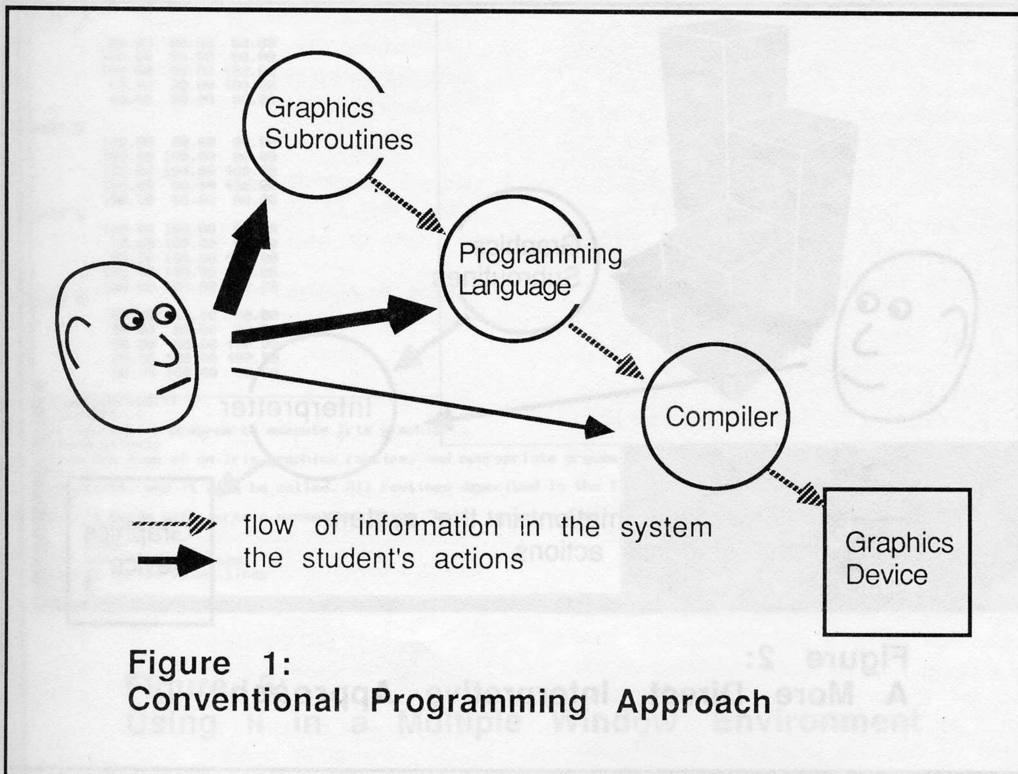

Figure 1:
Conventional Programming Approach

A student's initial, exploratory efforts are better supported by a fast, interpretive interface (Figure 2). The student should be able to compose requests for graphic subroutine calls, and have them carried out directly, with immediate visual results. The approach is not new, but can be dated back at least to Turtle Geometry [Byte 1982], [Papert 1980]. The work presented here is not intended to teach children to use computer graphics, or to teach them problem-solving skills; it is intended to teach the programming details of two- and three-dimensional shaded graphics in environments like the IRIS workstation [Silicon Graphics Inc. 1984].

A graphics interpretter should act as an additional tool in the student's kit. As he progresses, he will need to try writing real programs; this transition is easier if the language of the interpretter corresponds, as closely as possible, with the style of programming that will ultimately be demanded of the student. Although there is a temptation to provide additional functionality in the form of high-level primitives for drawing, menu-driven interfaces, etc., this must be resisted, unless those primitives are part of the toolkit the student will later use. The objective here is not to create yet another language for graphical expression, but to mimic, as closely as possible, the existing graphical component of the high-level programming language.

We have constructed an interpretter, for the IRIS workstation, called *irisinterp*, or *ii*. In *ii* the following sequence of commands produces a perspective view of a coloured box with a white top:

```
perspective(600,1,1,2000)
makeobj(1)
/* a tall red box */
color(1)
polf(5,  0, 0, 0,  10, 0, 0,  10, 0,40,
         0, 0,40,  0, 0, 0)
polf(5,  10, 0, 0,  10,10, 0,  10,10,40,
         10, 0,40,  10, 0, 0)
polf(5,  10,10, 0,  0,10, 0,  0,10,40,
         10,10,40,  10,10, 0)
polf(5,  0,10, 0,  0, 0, 0,  0, 0,40,
         0,10,40,  0,10, 0)
color(7)
polf(5,  0, 0,40,  10, 0,40,  10,10,40,
         0,10,40,  0, 0,40)
closeobj
color(0)
clear
lookat(45,45,50,0,0,15,1150)
callobj(1)
```

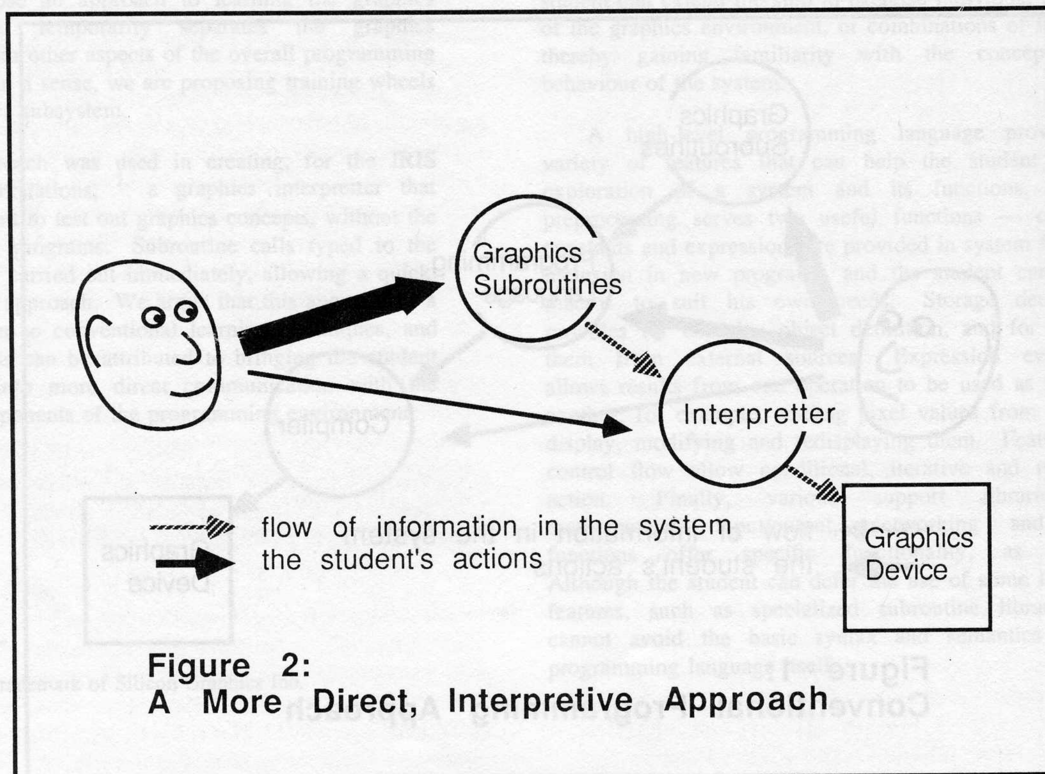

Figure 2:
A More Direct, Interpretive Approach

Punctuation, and other syntactic details are relaxed in *ii*; trailing semicolons (signifying the end of a statement in C), parentheses for subroutine arguments, and commas are treated as white space. Readers familiar with the IRIS programming environment will recognize that, with a few changes in punctuation, this sequence could be turned into a C program to carry out the same function. In fact, it is part of a longer sequence to draw the coloured boxes example provided in the manufacturer's documentation. With this sequence, the student can more easily follow the documented description of three-dimensional viewing controls, use of the z-buffer, etc.

Our early experience with *ii* led us to extend the basic concept, somewhat, to include the following features. A script inclusion feature has been added, by which a file containing *ii* instructions can be called, for insertion into a sequence; for example, the following sequence performs simple animation by calling the *boxes* script, and then rotating it by 5 degrees about the z-axis:

```
script(boxes)
color(0)
clear
rotate(50z)
callobj(1)
color(0)
clear
rotate(50z)
callobj(1)
color(0)
clear
rotate(50z)
callobj(1)
...
```

This allows longer sequences to be prepared with a text editor, tested and refined. It also allows the development of a set of tutorial examples. Scripts may be nested to a pre-determined limit; but recursion is ineffective, due to the lack of a method for expressing conditional termination. Standard defined constants are provided, for boolean values, colours, and screen limits:

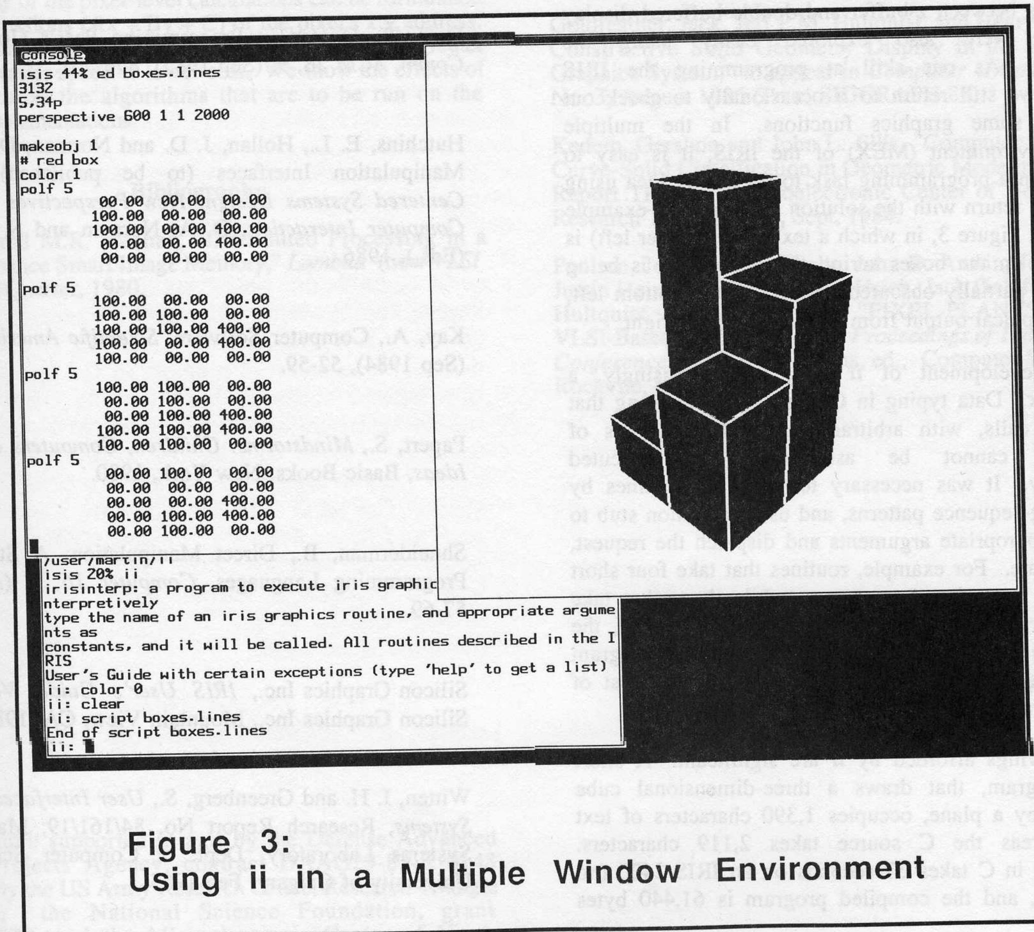

Figure 3: Using ii in a Multiple Window Environment

```
color(magenta)
move(0,0,0)
draw(xmaxscreen,ymaxscreen,0)
```

We have resisted the temptation to add declarative and iterative capabilities, partly because of the implementation effort they would require, and because the student who is ready to use those features is ready to move on to programming in C.

A few subroutine calls from the IRIS GL-2 graphics library are not supported in *ii*, because they are inappropriate in this context. For example, the subroutine *callfunc()* requires the address of a C subroutine, to be called from within a graphical object; the student using *ii* has no way of determining such an address. Similarly, the subroutine *defrasterfont()* requires an array containing the bitmap definition of a raster font; the student cannot be expected to type in such an extensive data structure.

The *ii* program has been useful in our efforts to explore and understand the IRIS programming environment. For example, details of the window-to-viewport coordinate transformation were initially confusing; trial and error with *ii* helped considerably. Interactions between z-buffer and double-buffered display techniques were also explored easily using the interpretter. As our skill at programming the IRIS increases, we still return to *ii* occasionally to check out details of some graphics functions. In the multiple window environment (MEX) of the IRIS, it is easy to digress from a programming task to try out an idea using *ii*, and then return with the solution in hand. An example is shown in Figure 3, in which a text editor (upper left) is being used on the boxes script, the *ii* program is being run from a partially obscured window in the bottom left, and the graphical output from *ii* is at the upper right.

The development of *ii* was, not surprisingly, a tedious task. Data typing in C is sufficiently strong that subroutine calls, with arbitrary numbers and types of arguments, cannot be assembled and executed dynamically. It was necessary to group subroutines by their calling-sequence patterns, and use a common stub to assemble appropriate arguments and dispatch the request, as appropriate. For example, routines that take four short integers as input form one group, while those that take four pointers to short integers form another. In all, the source code for *ii* takes 30 pages; the compiled program is quite large, at 164 k-bytes, since it includes most of the GL-2 graphics library.

The savings afforded by *ii* are significant. A short sample program, that draws a three-dimensional cube intersected by a plane, occupies 1,390 characters of text in *ii*, whereas the C source takes 2,119 characters. Compilation in C takes 37 seconds on an IRIS-2400 (no other users), and the compiled program is 61,440 bytes long.

CONCLUSION

In this paper, we have described a learning situation (graphics programming) in which the student's efforts are hampered by the insertion of an intermediary, high-level programming language and its support environment. A direct, interpretive approach improves the speed of learning, by bringing the student into closer contact, or communication, with his objective — the syntax and semantics of the graphics subroutine library he is trying to learn. *Direct communication* is an adaptation of the concept of *direct manipulation*, for situations where the user's objective is not a visible entity, but a linguistic process.

References

Byte, *Special Language Issue on LOGO*, McGraw-Hill, Aug 1982.

Duff, T., Quoted in Programming Pearls (Jon Bentley), *Comm. ACM 28*, 9, (Sep 1985), 896-901.

Hutchins, E. L., Hollan, J. D. and Norman, D. A., Direct Manipulation Interfaces (to be published), in *User Centered Systems Design: New Perspectives in Human-Computer Interaction*, D. A. Norman and S. W. Draper (Eds.), 1986.

Kay, A., Computer Software, *Scientific American 251*, 3, (Sep 1984), 52-59.

Papert, S., *Mindstorms: Children, Computers & Powerful Ideas*, Basic Books, New York, 1980.

Shneiderman, B., Direct Manipulation: A Step Beyond Programming Languages, *Computer 16*, 8, (Aug 1983), 57-69.

Silicon Graphics Inc., *IRIS User's Guide, Version 2.0*, Silicon Graphics Inc., Mountain View, CA, 1984.

Witten, I. H. and Greenberg, S., *User Interfaces for Office Systems*, Research Report No. 84/161/19, Man-Machine Systems Laboratory, Dept. of Computer Science, The University of Calgary, Feb 1985.