GRASS3, a language for interactive graphics

by NOLA DONATO
Wizard Software
Chicago, Illinois

ABSTRACT

With the advance of technology, graphics devices are becoming more powerful and less expensive, making interactive graphics increasingly popular as a method of man-machine communication. Often nonprogrammers play a principal role in the design and implementation of graphics applications. Interactive graphics requires a high level of feedback both with the user and with the hardware. For these reasons, conventional programming languages are not well suited for such applications.

This paper describes GRASS3, an interpretive language designed as a base for interactive graphics systems. The work derives from the author's thesis at the University of Illinois at Chicago Circle (UICC)[1] and similar work done by the author for the Bally Manufacturing Corp.[2] Design rationale for the language is given, followed by an overview including examples and a description of a specific real-time graphics system based on GRASS3.

DESIGN PHILOSOPHY

The GRASS3 language (GRASS3 stands for GRAphics Symbiosis System version 3) was designed as a base language for development of interactive graphics systems. Although GRASS3 bears very little resemblance to its predecessor, GRASS2,[3] much of the interactivity and simplicity which made old GRASS so powerful have been preserved in GRASS3. The language also borrows heavily from C and SNOBOL[4,5] for language design and internal structure.

One of the most serious drawbacks of conventional programming languages in the graphics environment is the difficulty of tailoring them to a particular device. In most of the higher level languages, subroutines are the only feasible way to add new features. Consequently, it is almost impossible to achieve the communication between hardware and software needed to support a real-time application. Even the recent efforts at graphics standardization such as the Core System are aimed primarily at static devices such as plotters.

So far, the standard way of solving this problem has been to design a special purpose language revolving around a particular device or hardware system. This approach was taken by the designers of SMALLTALK (which depends heavily on the Interim Dynabook[6]). GRASS2, a language used by artists at UICC,[3] revolves around the Vector General refresh CRT.

GRASS3 is designed to interface easily with specialized hardware and software. Depending on what devices it must talk to, GRASS3 may require a set of special commands, device-dependent variables or even new datatypes. A refresh Cathode Ray Tube, for example, needs a "picture" datatype. Creation and manipulation of display lists require a special set of functions. Device variables are also needed if the system has dials or joysticks.

The GRASS3 language is designed to make such internal rearrangements simple and straightforward. To a large extent the language is table driven. Because of this, it is not difficult to add commands, datatypes, or even new operators. One can define new conversions rules or redefine old ones. Many existing features (such as floating point support, interactive debugging, etc.) can be eliminated simply by recompiling the source. Almost all of the system commands can be made to dynamically swap in and out when needed. (One can also do this with user functions). Thus, GRASS3 can be easily configured to meet user specifications.

Another important feature of GRASS3 is that it is interactive. Almost anything allowed in a program may also be typed directly on the terminal. A user may "try out" a statement, display a picture or inspect her variables all without having to write a program.

This kind of interaction is necessary for interactive graphics. The feedback provided by such a system speeds up program development and the evolution of a graphics application. One should not have to go through the whole cycle of updating a source file, compiling, loading, initializing and then setting up the proper environment in order to determine the implications of a trivial change.

Interactivity is also essential when a human must be in the loop to supply decisions about how the animation is to proceed. Much of computer graphics is visual—the machine cannot predict whether one will be excited or bored by a particular effect and it is not capable of making artistic judgements. Conversational graphics systems are structured to permit just this sort of thing. Thomas Standish comes to the same conclusion in his paper on computer animation.[7]

Recent trends in home computing show that interactive systems are better for beginning programmers. Most commercially available home computers use some derivative of BA-

SIC.[8] Even experimental home computers, such as the Interim Dynabook, rely heavily on interactive feedback.[6]

Interactivity can help overcome the qualities of computer languages that are unnatural to the novice user. The immediate response available in an interactive system can surmount barriers that make a system difficult to program. For instance, both LISP and APL[9,10] are cryptic languages, yet they are very popular in interactive environments. LOGO, a derivative of LISP designed especially for naive programmers, was used as the base language for a graphics system developed by Abelson.[11,12]

The GRASS3 language is high level, but easy to learn and understand as well. Novices do not have to become super programmers to try out their ideas and experiment with the system. But as they gain experience they are able to expand their use of the more general features. GRASS3 can be useful to the naive user with minimal learning and, as she demands more powerful capabilities, they can be easily absorbed in small increments. This is important because problems in computer graphics are often tackled by nonprogrammers like educators and engineers. The designers of GLIDE, a language developed for CAD applications, discuss this in their book.[13]

If the graphics language is easy to learn and use, small projects can be done without hiring a professional programmer. For large projects, a readable language can allow a greater level of understanding and communication between designers and programmers. By reducing the gap between these two classes, systems can be tailored closely to the requirements of individual designers.

Much of learning involves making generalizations upon what one already knows. In learning a new programming language, one will often look at examples already known to work and modify them to suit a new purpose. If the semantics of a language are consistent (that is, operators in expressions always behave the same way, expressions are allowed whenever constants of the same type can be used, etc.) the learning process will be faster because the user's generalizations will be correct more often.

Consider, for example, the calculation of a subscript in FORTRAN. There are explicit rules governing the form such an expression may take, which may be found in any FORTRAN manual. Yet many FORTRAN programs are full of statements such as

```
K = I - 5
B = ARRAY(K)
```

when it is perfectly legal to combine the two (B = ARRAY(I - 5)). One may argue that, since the restrictions on subscripts are documented and consistent, cases like those above are programming errors and not limitations in the FORTRAN language. But constructions such as the above are rarely seen in C programs. Because there are no restrictions at all on subscripts in C, it does not occur to programmers to worry about whether a particular expression is permitted or not. Similar views are expressed by Weinberg in his book on the psychology of computer programming.[14]

In addition to being interactive, a graphics language must do a certain amount of housekeeping for the user. Most special purpose languages have a set of high level, nonprocedural primitives that free the user from the burden of managing details and allow her to concentrate on the real problem. ORACLE,[15] a relational database system, can do very complex queries in one or two statements. SIMULA,[16] an ALGOL[17] derivative designed for simulation, supports sophisticated multi-tasking capabilities. This "behind-the-scenes" management is especially important in graphics where data must be displayed as well as generated.

Part of system housekeeping includes maintaining datatypes. High-level datatypes such as strings, arrays, pictures, and list structures can be very useful for managing and organizing information. Consider the task of comparing two strings, something done often in programming. The C language does not have a string datatype.[4] A string is considered as a collection of characters. The following C program will return TRUE if the two given strings are the same:

```
index = 0;
while (string1[index] = = string2[index])
  {if(string1[index] = = END) return(TRUE);
   index = index + 1;}
return(FALSE);
```

GRASS3 allows the user to directly compare strings. Since strings are datatypes, the routine can be reduced to a single statement.

The same idea can be applied to computer graphics. A graphics programmer is often faced with displaying a series of pictures consecutively on the screen. If she can manipulate a picture as a single entity and group it with other pictures in an array or list she can trivially solve this problem. But if she must first create her own mechanism for dealing with pictures as whole objects, the simple display problem becomes a time-consuming programming task.

Another job the system can take over is memory management. All languages do this to some degree. Many, like FORTRAN and BASIC, have only static allocation. A program cannot reclaim memory used by arrays for other purposes (not even for different arrays). Others, like PL/1 and C, have primitives that will parcel out chunks of a dynamic memory area and reclaim them again. The programmer is responsible for maintaining the integrity of this area. Finally, there are languages like ALGOL and SNOBOL[4] that manage all memory automatically. The user thinks only in terms of the logical datatypes. To delete a list of items in PL/1, a subroutine is needed:

```
PTR = LIST;
WHILE PTR ! = NULL;
  DO
  TEMP = PTR -> LINK;
  FREE PTR;
  PTR = TEMP;
  END;
LIST = NULL;
RETURN;
```

In GRASS3 a single statement suffices:

```
list = null;
```

There are other housekeeping burdens the system assumes. Conversions between datatypes are done automatically whenever possible. The system provides simple mechanisms to input datatypes from the terminal or disk. User functions can be easily designed to accept arguments if supplied and prompt for them if omitted. Such things allow a programmer to describe her problem in terms which are closer to her logical conception of it.

GRASS3 is extensible and allows the user to program her own commands and configure environments easily and quickly. She can create independent subroutines and pass information between them. A logically clear method of passing parameters and returning values permits her to make extensions to the system. Local variables ensure that these extensions will be independent of one another.

User-extensible datatypes like structures and arrays help the user build complex constructions from simpler ones. Consider the implementation of an animation system. An artist typically creates a number of separate frames and displays them in a fixed order. The individual animation sequence determines how many frames there are and how long each one is displayed. If the programmer can associate a display time and duration with each frame and then group the frames together in a list, implementing a simple animation system becomes much easier.

String manipulation facilities also help the user configure environments. String manipulation is especially important for communication with the user on a terminal. If capabilities exist in the language to facilitate parsing, a programmer can develop a tailored sub-language whose syntax need not be a derivative of the syntax of the base system.

GRASS3 also has many easy-to-use debugging aids. Debugging tools include the ability to set breakpoints, examine variables, patch code, and trace a program's flow. Clear and plentiful error messages are part of this, too. Most programmers, especially novices, spend the majority of their time debugging. GRASS3 debugging features make programming less painful and can significantly decrease the time spent developing an application.

LANGUAGE OVERVIEW

The main way of communicating with GRASS3 is to type to it on the terminal. You can ask it to print information, create and run programs, or read files off the disk. Many statements are commands requesting the system to do something. For example, to print something on the terminal, you can type

```
print "The print command prints"
print "things on the terminal."
print 1,2,3
```

Other statements ask GRASS3 to evaluate an expression and perhaps save its value.

```
a = 2 + 3
```

And, of course, GRASS3 can evaluate expressions and use their results with a command.

```
print "The sum of 2 and 3 is",2 + 3
print "The value of a is",a
print "The average of 1 thru 5 is",(1 + 2 + 3 + 4 + 5)/5
```

Expressions need not involve only numbers. There are several other datatypes which can also be used in expressions.

integer	16 bit integers
float	32 bit floating point
string	variable length strings
array	*N*-dimensional arrays
node	programmer-defined datatypes
picture	device dependent
process	program which is scheduled

Most of the operators (like " + " and " − ") operate on numbers. A few, like "$" (concatenation) or "@" (indirection), need one of the other datatypes to operate on. In general, GRASS3 will attempt to convert whatever it is given to the type it wants.

```
print 1 + 2, '1' + 2, '1 + 2'
```

The statement above prints "3 3 1 + 2" on the terminal. In the first case, 1 and 2 are added and the result (integer 3) is converted to a string and printed. The second case requires the string '1' to be converted to integer 1 before the addition. The last case is already a string and is printed as is.

Most of the commands need their arguments to be of a certain type, too. For example, the print command will only print strings. Since numbers (integer and floating) can be converted to strings, it can print numbers or the results of expressions involving numbers, too. But the print command cannot print arrays, nodes or pictures.

To do complicated things you have to write a program. Programs are really the same as strings. To create a program, you just define a string containing the commands GRASS3 must execute. To run it, simply type its name.

```
hello = [print "howdy"]
hello
```

The example above creates a program called *hello* with a single print command in it. When executed (by the second statement above) "howdy" is printed on the terminal. There are four sets of string delimiters—single quotes, double quotes, square brackets and curly brackets. The quotes (" and ') may not be nested. The brackets ([] and {}) may be nested so long as they are paired.

You pass arguments to programs the same way you do to system commands. A program gets its arguments by using the *input* command.

```
max = [;return the maximum of 2 arguments
input int,A,B
if A > B, return A
return B]
```

In the example above, the *input* command fetches the next two arguments to the program, converts them both to type *integer* and stores them in the variables A and B. The *return*

command returns a single value (A or B) depending on their relative magnitudes.

The *prompt* command can be used in conjunction with input to provide the program with a means of prompting for arguments which were not supplied.

```
max = [;return the maximum of 2 arguments
prompt "enter first value"
input int,A
prompt "enter second value"
input int,B
return A * (A > B) + B * (A < = B) ]
```

The alternate version of *max* above will prompt the user for any argument that is not supplied. She may then enter it on the terminal and the program will proceed. Note that the relational operators (< > < = > = = = !=) can be used in expressions with other operators. Any line starting with a semicolon is considered a comment and ignored.

There are two types of names in GRASS3—dynamic and fixed. Fixed names are one or two characters long and may only have one kind of datatype associated with them. For example, fixed names a,b,. . .,z can only have integer values. Fixed names fa,fb,. . .,fz can only be floating point. Depending on your system there may be names d0,d1,. . .for dials and jx,jy,jz for joysticks as well.

Dynamic names can be up to seven characters long and can be assigned any kind of data. No declarations are needed in GRASS3. One simply assigns a value or expression to a name as needed. When a new value is assigned, the old value is deleted.

Dynamic names that begin with a lower case letter are known throughout the system to all programs. Those that begin with an upper case letter are local. If a local name is used in a program, it is deleted when that program exits. This allows programs to use the same names without confusion.

Transfer of control in GRASS3 can be done with the *goto* command and labels, or by the more elegant structured constructs like *while* and *do*. The following program prints the values of an array.

```
prompt "Which array"
input array,ARRAY
I = -1
S = size(ARRAY)
while + + I < S, print I,ARRAY(I)
```

Statements may also be grouped within brackets, as illustrated in the following loop, which prints the types of its arguments.

```
do[prompt "enter argument"
   input value,ARG
   if ARG = = '',break
   TYPE = type(ARG)
   if TYPE = = "array",TYPE = $" of " $ type(ARG(0))
    print "Type is",TYPE
   ]
```

Some explanation is in order here. The *do* command is like *while* except that the test (if any) is done at the end of the loop. Using *value* in the input command allows any type of argument to be fetched. The *type* command returns a string giving the type of its argument. Note that array arguments are further inspected as to the type of their elements. When a null argument is gotten, *break* is used to exit the current loop.

For beginning programmers, GRASS3 has some nice features to make programming easier (and more enjoyable). First, there are interactive helps. If a user forgets the syntax or arguments of a command, she simply types *help* followed by the command name and GRASS3 responds with a description of the command and examples of how to use it. You don't have to look in the manual if you forget what a command does or how to call it.

Second comes interactive debugging. When GRASS3 finds something wrong, (it is requested, say, to do something it can't do or the system runs out of some resource), an error message is printed on the terminal. If this error occurred inside a program, GRASS3 puts that program into *debug* mode. When in debug mode, the normal "*" prompt is replaced by "#" to let the user know she can issue debugging commands. With the debugger, the user can set breakpoints, single step, trace program execution, and even make simple patches. The *edit* command (which invokes the GRASS3 text editor) can also be used in debug mode. In addition to debug directives, the user can still issue any other GRASS3 commands, too. If one is not absolutely sure a program is correct, the *debug* command can be used to place the program in debug mode before an error actually occurs.

GRASS3 can be configured for small systems where memory is tight using the automatic swap feature. Some of the GRASS3 commands are not resident—they live on the disk. When a nonresident command is requested, GRASS3 will automatically read it off the disk and then delete it when it has finished execution. The user can request some of her own programs to automatically swap off the disk, too. The *keep* command allows a nonresident module to remain in memory after it has finished execution. Keep can speed up programs where a swapping command is used repeatedly or in a loop.

One of the most powerful features of the GRASS3 language is that the user can run two or more independent programs at the same time. For example, suppose we have already written a program called *walk,* which makes a little person walk across the screen. It accepts arguments telling it where to start the person and which way she is to walk. On most systems the program would have to be completely rewritten if you wanted to have two people walking at the same time. In GRASS3 you would use the *schedule* command as follows:

```
sched walk,100,10,left
sched walk,10,10,right
```

The *walk* program can be scheduled twice with different arguments to show two people walking. GRASS3 will execute one line from the first scheduled program, one line from the next, etc. to give the illusion that all scheduled programs are running at the same time.

THE VISION II INTERFACE

Although the GRASS3 language definition does not include graphics primitives, the system is designed to make the addi-

tion of new commands and datatypes as easy as possible. The VISION II raster graphics system[18] is the most recent device to be interfaced to GRASS3. The screen is a 256 x 256 array of pixels, directly addressable by their X,Y coordinates. Commands exist in GRASS3 to draw lines, boxes, and points, display text, save areas of the screen on the disk or in memory and display them again, etc. A *picture* datatype and utilities to create and manipulate pictures are also part of the system.

Suppose we have some function describing a particular sequence of X,Y coordinates. It could be algorithmic and coded as a program or it could describe some inputs from the outside world (joysticks, perhaps). Let us assume the GRASS3 variables x and y are being updated (in real time) according to this function.

If we have a previously created picture, CAR, and we want it to move around on the screen according to the path specified by x and y we would code

```
sched [plot CAR,x,y,erase
       repeat ]
```

This would schedule a program to continuously move CAR as directed by the variables x and y. Using this method, any number of pictures may be moved simultaneously on the screen.

CONCLUSIONS

Experiments with VISION II and other systems have shown GRASS3 to be very powerful in putting together complex graphics applications quickly. (The VISION II picture editor described by Rocchetti[18] was implemented by the author in a single evening). The language has proven to be easy to learn by a variety of nonprogrammers (several of them children). A subset of GRASS3, called ZGRASS, was used by Bally in their *FUN 'N BRAINS* home computer system.[2] About half of the programs used to demonstrate the above unit were written by nonprogrammers (advertising executives) over a period of several weeks. The other half were written by the developers within the span of a few days. Had the same applications been implemented the conventional way (in assembly language), the combined effort would have exceeded many man-months.

Isolation of operating system interface code made it trivial to port GRASS3 from UNIX[19] to the DEC operating systems. This was particularly desirable, since at the time GRASS3 was developed UNIX had no real-time primitives. An experienced assembly language programmer coded and debugged the RT-11 operating system interface in under a week. It was running under RSX-11M several days after that. Note that the above times represent only coding of language features (like file I/O, panic traps, etc.). The author does not mean to imply that device- or hardware-dependent applications can be ported to a new operating system nontrivially. (It would be impossible, for example, to fully support a refresh CRT under UNIX without making operating system modifications).

When GRASS3 was born (1976), the only implementation language that spanned all PDP-11 operating systems was MACRO-11 (PDP-11 assembler). Since then, the C language has grown in popularity and has been implemented on many different machines and operating systems. A portable version of GRASS3 (coded in C) is currently being written. The new version will have more powerful string manipulation primitives and enhanced multitasking capabilities (similar to those in the ADA language[20]). We hope that these efforts will also yield a GRASS3 compiler, which will produce C or some sort of portable macroassembler source.

Another feature in the works is a *picture compiler,* which will compile a subset of GRASS3 into a form that can be loaded and executed by the VISION II graphics processor. Thus, picture programs could be created and debugged interactively with GRASS3 and finally executed by one or more VISION II processors.

REFERENCES

1. Donato, Nola, "GRASS3—A Base System For Interactive Graphics," Masters Thesis. University of Illinois, 1978.
2. DeFanti, T. A.; Jay Fenton; and Nola Donato, "Basic Zgrass—A Sophisticated Graphics Language for the Bally Home Library Computer," *Computer Graphics,* Vol. 12, No. 3 (August 1978), pp. 33-37.
3. DeFanti, T. A. Dissertation. Ohio State University, 1973.
4. Ritchie, D. M. *C Reference Manual.* Bell Telephone Laboratories, Murray Hill, 1974.
5. Griswold, R. E.; J. F. Poage; and I. P. Polonsky. *The SNOBOL4 Programming Language.* Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1968.
6. Kay, Alan and Adele Goldberg. *SMALLTALK-72 Instruction Manual.* Xerox Corporation, March 1976.
7. Standish, Thomas A., "Remarks on Interactive Computer Mediated Animation," *Proceedings of the Ninth Annual UAIDE Meeting,* 1970.
8. Kemeny, John G. and Thomas E. Kurtz. *BASIC Programming.* New York: John Wiley & Sons, Inc., 1971.
9. Howard, Forrest William. *LISP Programmer's Manual.* HRSTS Science Center, September 1975.
10. Freeman, Peter. *Software Systems Principles.* Chicago: Science Research Associates, Inc. 1975.
11. Abelson, Hal.; Nat Goodman; and Lee Rudolph. *Logo Manual.* Massachusetts Institute of Technology, 1974.
12. Goldstein, Iran, and others. *LLOGO: An Implementation of LOGO in LISP.* Massachusetts Institute of Technology, June 1974.
13. Eastman, Charles and Max Henrion. "GLIDE: A Language for Design Information Systems," *Computer Graphics.* Vol. 11, No. 2, Summer 1977.
14. Weinberg, Gerald M. *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold Company, 1971.
15. Preger, R.L. "ORACLE User's Guide." Relational Software Inc., Menlo Park, California, 1980.
16. Birtwistle, G.M., et al. *SIMULA begin.* Auerbach Publishing Co., 1973.
17. Tanenbaum, A. S., "A Tutorial on ALGOL 68," *ACM Computing Surveys.* Vol. 8, No. 2, June 1976.
18. Rocchetti, R. J., "VISION II—A Small Scale Expandable Graphics System," Masters Thesis. University of Illinois, 1978.
19. Ritchie, D. M. and K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal.* Vol. 57, No. 6, July-August 1978.
20. Ichbiah, J. D. et al., "Rationale for the Design of the ADA Programming Language," *SIGPLAN Notices.* Vol. 14, No. 6, June 1979.