GRAPPLE
Graphics Application Programming Language

D.L. Williams
Bell-Northern Research
Ottawa, Canada

ABSTRACT

The main barrier to greater use of computer graphics has been the lack of a suitable programming language. This paper describes a new language, called Grapple, which was developed at Bell-Northern Research during twelve months of interaction with users and potential users. It is in everyday use both as a means of compact storage of graphics data and as a tool for graphics systems programming. Grapple is based on a function notation and the use of elementary operations called 'primitives'. The primitives include simple picture-drawing operations and input-output operations for displays and computer files. Functions (which can be given mnemonic names) are built up from primitives and other functions.

RESUME

L'absence d'un langage de programmation approprié a été, jusqu'à présent, le principal obstacle à l'utilisation étendue des applications graphiques de l'ordinateur. Ce mémoiré decrit un nouveau langage intitulé Grapple (Graphics Application Programming Language, ou langage de programmation graphique) mis au point aux Recherches Bell-Northern, à la suite de contacts qui ont duré un an entre les spécialistes de la compagnie et des usagers actuels et futurs. Ce langage est aujourd'hui d'un usage quotidien, servant tant à mémoriser des donnees graphiques sous un forme compacte qu'à la programmation graphique du'n ordinateur. Grapple se base sur une notation par fonctions et sur l'utilisation d'opérations élémentaires appelées 'primitives'. Les fonctions (qui peuvent recevoir des noms mnémoniques) sont construites à partir de primitives et d'autres fonctions. Les primitives comprennent des opérations simples de dessin ainsi que des opérations d'entrée-sortie pour les affichages et les dossiers d'ordinateur.

INTRODUCTION

Before PL/1, before Algol, before Fortran and its kin, most computer programming was done in 'Symbolic Assembly Languages', though some people were still writing directly in machine codes. This formed a barrier to the use of computers. A person with a problem to be solved had to consult a programmer, who would write the program for him. The availability of high-level languages, which were fairly easy to learn, changed the situation. Programs could then be written by the user himself, without another person having to interpret his commands to the computer.

At present, graphics programming is at the pre-Fortran stage; users don't write graphics programs, professional programmers do. Various languages are being developed which should change all this. Grapple is one which is intended as an easy-to-use language for describing both pictures and graphics systems. These two facets are basic to the design of Grapple. A subset of the language forms a convenient means of describing pictures, while the language as a whole is a powerful medium for designing complete computer graphics systems. For example, the interactive graphics editor, which allows one to generate and modify pictures directly, without manually encoding them as numbers, is itself written in Grapple.

Development started early in 1972, when it became apparent that a means had to be found for encoding graphics data in a compact standard form. Within a few weeks, Grapple code was being used as the medium for transferring data between various graphics machines. At the present time, Grapple is a mature programming language. Development is continuing, but the present features are stable. This paper is intended as a brief primer to Grapple.

DESCRIBING PICTURES

As in most languages, Grapple has some very basic operations which can be used on their own, or can be put together to perform more complicated operations. These basic operations are called PRIMITIVE FUNCTIONS. To visualize what they do, it is best to imagine a pen moving over a drafting board; the location of the pen can be specified by giving its x and y coordinates.

For example: to put the pen at the position x : 20 units, y : 40 units, one uses the <u>SET</u> primitive. All primitives are specified by their initial letter, so the command would be:

```
        S(20,40);
```

This example illustrates three characteristics of the language: the arguments of the function are enclosed in parentheses and are separated by commas; the command is terminated by a semi-colon; and dimensions are specified in units whose value can be set to any length. The default value is .01 inch, but one can just as well work in millimetres or miles.

Having put the pen at that position, one can now draw a line to another position. To draw a horizontal line to the position x : 4 cms, y : 4 cms, using the <u>DRAW</u> primitive, the complete command is

```
        S(20,40),D(40,40);
```

Successive function calls are separated by commas; they can follow immediately after one another, be separated by blanks, or be on successive lines if one prefers, because it's the semi-colon that shows where the command ends.

Another way of specifying a position is to give, not the absolute location, but the x-offset and the y-offset from the current pen position. This is really telling the pen, not where to go, but how far to go from where it is now. The <u>VECTOR</u> primitive draws a line in this manner, and the example of a line could be drawn by:

```
        S(20,40),V(20,0);
```

One can now describe a small figure in Grapple. Consider a rectangle 2 cms by 3 cms, with its lower left corner at the point (2,1). The command

```
     S(20,10),V(20,0),V(0,30),V(-20,0),V(0,-30);
```

will draw its sides one after the other, going around counterclockwise. Using the VECTOR primitive emphasizes that the lengths of the sides are 20 and 30 units. This would not have been immediately apparent if the DRAW primitive had been used.

There is a shorthand way of writing the above command, still using the VECTOR primitive; several pairs of dx and dy values can be put in one function call, so the command would look like this:

```
     S(20,10),V(20,0,0,30,-20,0,0,-30);
```

If this statement looks a little confusing, it can be written in this manner:

```
     S(20,10),V(20,0,    0,30,    -20,0,    0,-30);
```

A rectangle is a special kind of shape, in which successive lines are parallel in turn to either the X-direction or the Y-direction. Two special primitives, X and Y allow lines of this kind to be specified in a very compact manner. Each primitive draws a string of line segments, alternately in the x-direction and the y-direction using the given values as the lengths of the line segments. The X primitive draws the first segment in the x-direction; the Y primitive draws its first segment in the y-direction.

Using one of these, the above command shrinks to:

```
          S(20,10),X(20,30,-20,-30);
```

Whether the above commands are typed at the keyboard or read from a file, they are executed immediately and then discarded. A command of this form, telling Grapple to perform certain graphical functions, is called a _FUNCTION IMPERATIVE_. By giving it a name, the command can be converted into a _FUNCTION DEFINITION_. It will be stored, and can be invoked more than once.

```
     BLOCK: S(20,10),X(20,30,-20,-30);
```

A colon separates the function name from the rest of the definition; the name itself can be up to eight characters long, and the first character must be alphabetic. Single-character function names are not allowed, since a user function called, say, S, would redefine the primitive S, and cause great confusion. Similarly, although the dollar sign ($) is deemed to be an alphabetic character, one should not begin a function name with a $ since the name could accidentally redefine a _SYSTEM DEFINED FUNCTION_.

Now that the figure has been defined and given a name, a Function Imperative will command that it be drawn. The statement:

```
               BLOCK;
```

will do this. Inside a function, all positions and dimensions refer to its own private measurement system. For example,

```
          S(60,10),BLOCK;
```

would give the picture shown in figure 1.

The point marked with an asterisk (*) is (60,10) as far as the outside world is concerned, but is the point (0,0) for everything within this copy of BLOCK. The first operation within BLOCK is S(20,10), but this is (20,10) with respect to BLOCK's own reference point, not (20,10) of the overall drafting board. One consequence of this feature is that one can put several copies of BLOCK at different places in the picture, such as

```
     S(20,0),BLOCK,S(60,0),BLOCK,S(100,0),BLOCK;
```

which would give a horizontal row of BLOCKs.

The examples so far have been executed 'as written'. It is often desirable to write a somewhat general function which can be modified as it is executed. One way of affecting the execution is to pass parameters to a function when it is called. A square, for example, needs one parameter, representing the length of side.

```
          SQUARE(1): X(&1,&1,-&1,-&1);
```

Then SQUARE(27) would draw a 27 mm square. Inside a function, the parameters are referred to by their position in the formal parameter list, &1 being the first, &2 the second, and so on.

Another way of controlling the execution is by 'Action Modifiers'. These control repetition, rotation, and mirroring. Thus

```
          ?(45)SQUARE(27);
```

will draw the square at 45 degrees. There is also an 'IF.......THEN......ELSE' mechanism that allows the programmer to control which function or action list is to be performed. There is not however any GO TO statement.

The above primitives have been described in detail, as they give the 'flavour' of the language. Further picture-drawing primitives are available, but the ones described are representative.

GRAPHICS SYSTEM PROGRAMMING

Before considering the special feature put into Grapple for graphics programming, some mention should be made of 'Euphemisms'. The penalty in terms of lowered efficiency for introducing a new function name is quite low. Thus a rather obscure primitive, such as F(3,n), can easily be renamed $COS(n). This principle has been used extensively. The average user probably is not aware of it, but one finds that a wide variety of mnemonically-named programming features is actually based on three or four families of primitives.

One family is mathematical functions, an example of which is shown above. Some others are $ABS, $ROUND, and $ARCTAN. The existence of these functions implies computational facilities in the language, and indeed Grapple has constants, variables, arrays, assignments, and the usual arithmetic and Boolean operators.

```
            (ARRAY(2)+B) * 36 -> C;
```

A second family (based on the P primitive) is concerned with file input and output. These functions give Grapple the ability to open and close, read and write standard computer files; Grapple can also generate files of graphics data for plotting on a drum plotter or cutting on a mask cutter.

Input from a light-pen, joystick or other device has proved a difficulty in many graphics systems. Grapple has taken a direct approach with two primitives. One merely requests the entry of a pair of coordinates from the operator, and passes them to the program. The other checks whether a given point scores a 'hit' on any line of a specified figure. This could be a long, expensive process; the secret of its success depends on two things: it happens that functions usually do not contain much code in themselves because they can call other functions; and picture-drawing functions contain 'extent' data, which gives the maximum extent of the figure. If the specified point lies outside the extent, there is no nedd to search the figure for coincidence. The 'extent' feature also increases the efficiency of the windowing operation, on display.

Apart from certain miscellaneous operations (such as $ERASE to erase the screen) the other major primitive group controls the compilation of source data; which function is to be compiled, which is to be re-compiled after a change, what libraries should be searched for undefined functions, whether a trace is to be made of compilation and execution, and so on. The library facility has proved very useful, as it enables designers working in various fields to build up repertoires of functions that then become available to all; each person does not have to re-invent the same function over and over. An example of a standard function is a menu-processor, which is given a list of names of functions when it is called. The list is displayed on the screen of whatever terminal is in use, and if a 'hit' is scored on any word in the list, then the named function is called. Some other standard functions are shown in figure 2.

Although Grapple is a versatile language, it was recognized that some parts of a program might more easily be written in some other language - for example, report generation. Two mechanisms were therefore provided for linkage to an external PL/1 or Fortran program. One mechanism makes a call to an external program each time each function is entered. The other calls the external program each time $CALLEXT is encountered (it will be realized that $CALLEXT is of course a 'euphemism').

No mention has yet been made of hardware. A Grapple interpreter could be written for almost any interactive computer, but the present implementation is on a time-shared 360/67. Five different types of display terminal are in use; one is a refresh-tube display, the others having direct-view storage tubes. The Grapple system can be accessed from any of these. Indeed, Grapple programs can be run from almost any keyboard terminal, with the 'display' mode turned off. Graphics input is a little tricky on a teletype, but is achieved by typing out 'X: Y: ' and letting the user fill in the blanks.

CONCLUSION

In twelve months of interaction with potential and actual programmers, a graphics language has been developed which can be used by people who are not necessarily professional programmers. Several major programs, written in Grapple, are in everyday use. The language is now stable, and further development will mainly take the form of establishing libraries of standard and special functions.

ACKNOWLEDGEMENTS

Reference has already been made to the interaction with users that took place during the development. The authors of Grapple would like to acknowledge the assistance given to them by these long-suffering people. Special mention should be make of G. Caple, R. Lewis and H. Rombeek, but there were also many more who who contributed helpful comments and suggestions.

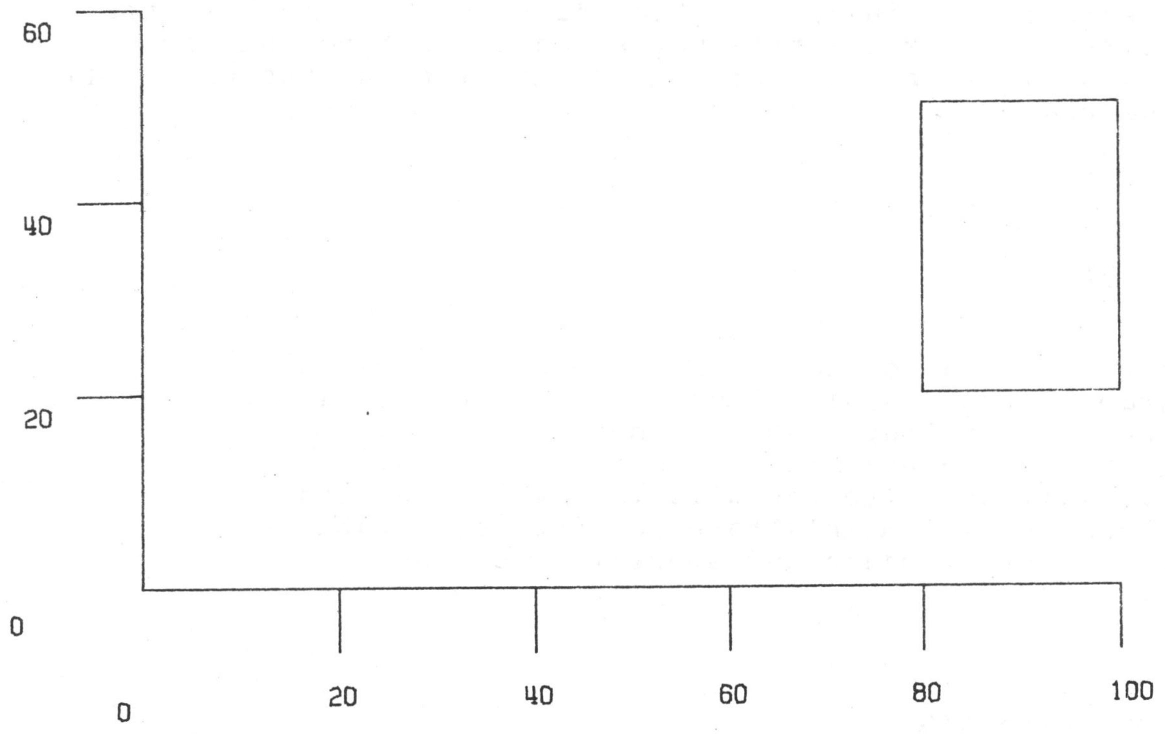


FIGURE 1

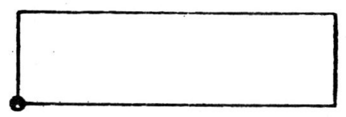
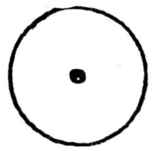
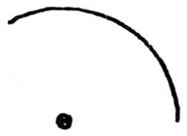
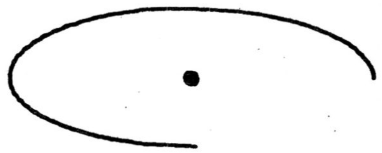
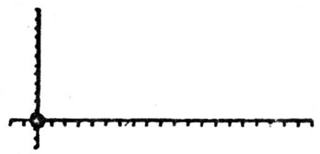
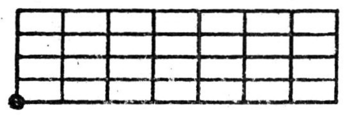
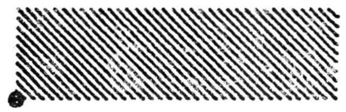
$RECT DRAW RECTANGLE	$RECT(X,Y), $RECT(700,200),	
$CIRCLE DRAW CIRCLE	$CIRCLE(RADIUS), $CIRCLE(150),	
$ARC DRAW CIRCULAR ARC	$ARC(RADIUS,ANG1,ANG2), $ACR(250,0,120),	
$ELIPSE DRAW ELIPTICAL ARC	$ELIPSE(XRAD,YRAD,ANG1,ANG2), $ELIPSE(400,150,0,270),	
$TMARK DRAW AXES WITH TICKMARKS	$TMARK(DX,NX,NNX,DY,NY,NNY), $TMARK(30,20,2,20,12,3),	
$GRID DRAW RECTANGULAR GRID	$GRID(DX,NX,DY,NY), $GRID(100,7,50,4),	
$SHADE SHADE A RECTANGULAR AREA	$SHADE(X,Y,DELTA), $SHADE(700,200,20),	

FIGURE 2

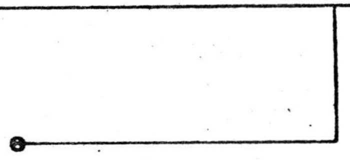
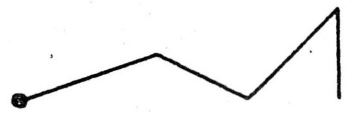
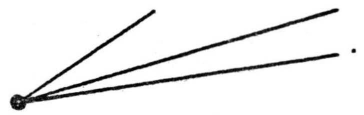
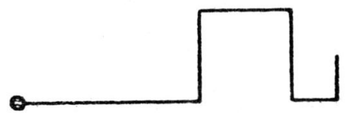
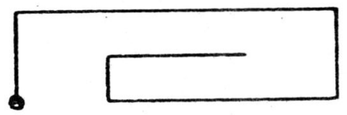
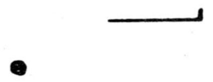
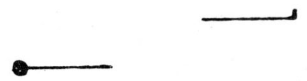
D DRAW ABSOLUTE LINES	D(X1,Y1,X2,Y2, ,), D(700,0,700,300,300,300),	
V DRAW RELATIVE VECTORS	V(DX1,DY1,DX2,DY2, , ,), V(300,100,200,-100,200,200,0,-200),	
R DRAW RELATIVE RADIALS	R(DX1,DY1,DX2,DY2, ,), R(700,100,700,200,300,200),	
X DRAW RELATIVE ORTHOGONAL LINES FROM X AXIS	X(DX1,DY2,DX3,DY4, ,), X(400,200,200,-200,100,100),	
Y DRAW RELATIVE ORTHOGONAL LINES FROM Y AXIS	Y(DY1,DX2,DY3, ,), Y(200,700,-200,-500,100,300),	
S SET PEN ABSOLUTE	S(X,Y), S(200,100),X(200,20),	
M MOVE PEN RELATIVE	M(DX,DY), X(200),M(200,100),X(200,20),	

FIGURE 3