# Graphical Manipulation in Programming Languages : Some Experiments.

John Livingstone.

For the degree of M.Sc.

Department of Computing Science.

University of Glasgow.



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

# Contents.

# Acknowledgments.

I am extremely grateful to Professor Malcolm Atkinson for his guidance and support during my work and for his proof reading and assistance during the write up.

I would also like to thank all the members of the PISA project team at the Universities of Glasgow and St. Andrews for their assistance throughout my work.

I acknowledge the work done by Iain Armour in implementing the *print* statement, on the Perq computer, and adding the edge violation handler, and also the programming done by Richard Cooper to test out the new implementation.

# Organisation of Thesis.

A survey of graphics programming languages was conducted to identify the current provisions for graphics. In particular, **PS-algol** [PPRR12], a persistent programming languages which includes bitmap graphics facilities, was analysed. A review of the graphics of this language is given in chapter 1. The findings of the survey as a whole are discussed in chapter 2.

The investigation into graphics language facilities was continued by the implementation, in PS-algol, of various example programs previously written for the **Whitechapel MG-1** workstation [WCWM85] in C. A comparison was made of the facilities provided by the two systems and of the approach to provision of graphics in a language. The work done and conclusions drawn are discussed in chapter 3. This looks at the graphics side of the language from the programmer's point of view.

The investigation was continued by looking at the machine independence of PS-algol. This was done by porting the full PS-algol system to the Whitechapel MG-1 workstation. Further points could then be made about the design of the language pointing out its weak and machine dependent parts. The work required to port the system and conclusions drawn from the work are discussed in chapter 4. The MG-1 implementation of PS-algol has several differences in the user interface from the original **ICL Perq** [ICLP84] implementation. These changes are discussed in chapter 5.

After the porting was complete changes were suggested and made

to the language's input / output facilities allowing these functions to operate via the graphics screen. These changes were intended to improve the language's usability and machine independence. The changes made are outlined in chapter 6.

These changes are reviewed in chapter 7, here experiences with the new constructs and other user s' thoughts are discussed and techniques of use are outlined.

# Introduction.

Today in the computing world the trend of machine size is moving away from the large mini or main frame computer towards the workstation. Most workstations have a bitmap screen which supports a window manager and input devices such as a mouse or tablet. Making full use of these systems allows the user to build a user interface into programs which is superior to that which could previously have been provided on a keyboard driven system. Output can be presented in a clear and interesting way and input can be limited to that relevant to the current state, and consequently can require little effort by the end user. Output can be formatted and, visual cues, which help the user's perception, can be presented using a variety of fonts, highlighting, boxes etc. Selections can be made by use of the mouse through menus rather than by entries from the keyboard. Programs, which use these effects well, are easier to learn about, more interesting and faster to use. A good example of increased usability provided by a mouse based system is an editor which takes commands from the mouse through the use of menus and relies on the keyboard for text input only. Most users of such systems rapidly come to like them and feel their old keyboard driven systems are inadequate by comparison.

The graphics facilities provided by a workstation can be accessed through a system language such as C. This allows the programmer to use all the operations at the lowest level. Direct access is allowed to all parts of the system, such as the structures used to hold bitmaps and changes may be made to these in any way desired in order to build up the required display. The methods used to implement graphics on the

machine can be used directly and full access to the machine model allows the user to write efficient fast running programs which make full use of the machine and graphics implementation. This type of language is currently used by the serious graphics programmer intent on producing elaborate pictures. Such programs must be as efficient as possible due to the amount of processor time required to run them and since use of a system language is the only access such programmers have to the power required they are forced to make use of the machine's graphics facilities this way.

It is becoming increasingly necessary for programming languages other than systems languages to allow access to the graphics facilities of the workstation. Use of system languages leads to errors more readily than a high level language (HLL) and requires users to be experienced and trained in their use and also to be trained in the use of the particular machine, since each type of workstation has a different interface to its graphics facilities. Retraining programmers to use a different graphics machine is costly in time and money and would not be required if the same graphics language could be used on all machines. This indicates the need for a machine independent graphics language. In principle high level languages may provide easier programming by abstraction of operations to a higher 'cleaner' level. Strongly typed languages provide more reliable programs since the programmer is forced to consider different objects as being differently typed and cannot use incompatible types in an expression, for example a character cannot be manipulated as an integer. Operations carried out in a weakly typed language can be implemented in a strongly typed language but may have to be expanded to make explicit the required operation which does not break the type rules since the strongly typed language disallows many short cuts frequently used in the weakly typed language. Preventing the use of such short cuts and forcing requirements to be stated explicitly gives

more predictable behaviour from the code produced by compilers which leads to more reliable programs. However, *normal* ( or commonly occurring ) operations should be expressed more succinctly in a HLL. High level and strongly typed languages are machine independent which means that the user does not require the expensive retraining for each new machine used and can make use of experience and software gained from previous work. It is now desirable to introduce graphics facilities into these high level and strongly typed languages. These intend to provide all the facilities the user needs in a simple and concise manner, and allow programs to be transferred, almost unchanged, from one machine to another. The serious graphics programmer, discussed above, may not yet find such systems provide the computation speed required. However, the support and performance can be expected to improve via the following two mechanisms :

(i) the formulation of facilities in an HLL, and the feedback from their use, should lead to a better understanding of the best constructs with which to formulate graphics programs;
(ii) well identified constructs give a precise target for implementers focusing their attention appropriately.

The latter mechanism stimulates both language implementors and machine designers to provide efficient support. (Earlier examples of this effect are the support for a standard number manipulation system, embodied in the IEEE floating point chip standard [FATR82] and the support for procedure calls in hardware design.) It is particularly important, therefore, that language designers should explore the graphics facilities required by the HLLs now, as their is much interest in the design of hardware to support graphics at present, and since support

is more likely to be generally applicable if it has been shown to fit well in a HLL.

The graphics facilities a HLL provides should allow for easy programming by saving the programmer the problem of constantly keeping track of screen positions, sizes and other tasks unconnected with the application. They should give a high level of performance for a small amount of work on behalf of the programmer. This implies that the language facilities must hide all the machine dependent features such as mouse event queues or memory allocation, and provide the programmer with a clean and simple set of operations allowing full concentration on the problems of the application and of data presentation. The designer of the graphics facilities in the language must predict the needs of the programmer and produce a system which is an appropriate compromise between feasibility of implementation and convenience for the programmer. This thesis attempts to set out the requirements of a graphics programming language and to begin the exploration of that compromise.

# 1. A Description of the PS-algol System.

PS-algol is primarily a persistent programming language, which means that data objects can persist from one run of a program to the next. This allows data structures, such as binary trees, to be used during execution of a program without being built each time, or to be shared by several programs. The databases allow storage of any objects within the language such as vectors, structures, images and procedures as well as the base types integer, real, string and boolean. The language treats procedures as first class data objects [ATKM85] meaning they can be assigned, executed and stored in the databases.

The language is a block structured object oriented language based on the language S-algol[COLA82]. It is an extension of this language allowing the persistent facilities and presenting procedures as first class data objects.

A full description of the language can be found in [PPRR12]. After presenting a minimal background, only the graphics facilities will be described.

## 1.1 General Features of the Language.

Objects variables and constants are declared by use of the *let* statement. This allows the introduction of an object to the program, specifies its type and gives it an initial value in order to eliminate the problem of undefined variables from the language. The *let* statement, in effect, contains the first use of an object. Two uses of the *let* statement are given in *fig.* 1.1.

```
let days.per.week = 7
let days.per.month := 31
days.per.month := 28
```

**fig. 1.1** Example use of the *let* statement.

Since the number of days per week will always be 7 it is appropriate to declare it as a constant. This is done by using an "=" in the declaration as opposed to a ":=" in the declaration of a variable. *days.per.month* declared as a variable can be changed as is done in the third line of the example. An attempt to change the value of days.per.week would produce a compilation error since it is a constant implying it may not be changed.

A frequent cause of confusion to new users of the language is the inclusion of *dots* in the identifier syntax, as seen in *fig.* 1.1. People used to working in Pascal assume this implies field specification within records whereas dots are used to separate parts of the PS-algol identifiers to improve clarity and readability.

## 1.2 Output Facilities.

Output can be displayed in two ways in PS-algol. Images can be displayed on the screen by the use of raster operations, screen being a predefined image. Text, made up of strings, integers and reals, can be output to standard output by the *write* statement which was designed for use on character output devices.

### 1.2.1 Output of Images.

Images can be thought of as two dimensional arrays of pixels, where pixels can consist of one or more planes, which allows the programmer to manipulate coloured images or to group several monochrome planes in the one structure. Images of only one plane are monochrome as each of their pixels can have the value **on** or **off.** An image declaration specifies the size of the image and the initial value of each of its pixels, an example is given in *fig.* 1.2.

**let** example.image = **image** 50 **by** 70 **of off**

**fig. 1.2** An Example image declaration.

This would initialise a 50 by 70 pixel image to **off**, implying it should have one plane. It will also be a constant image with variable pixels, which implies it can not be made to refer to another image but its contents may be changed.

A *limit* operation can be carried out on an image in order to produce a sub-image. An example of the limit syntax is given in *fig.* 1.3.

**let** a.window = **limit** screen **to** 100 **by** 120 **at** 50, 50

**fig. 1.3** An example of a limit operation.

The variable *a.window* can be used in the same way as any other image but any changes made to it will also be seen on the image which has been limited at the position specified in the *at* clause of the limit operation. In the example case any changes made to *a.window* will be seen on the screen.

Raster operations can be executed between two images, they take each pixel of a source image and combine it with the corresponding pixel of the destination image using the specified logical combination rule. The raster-op function is discussed in [NEWW79]. By using the *limit* operation raster-ops can be carried out between parts of images. PS-algol provides eight raster operation rules. These are given in *fig.* 1.4.

| | |
|---|---|
| **copy** | dest := source |
| **xor** | dest := source xor dest. |
| **ror** | dest := source or dest. |
| **not** | dest := not ( source ). |
| **rand** | dest := source and dest. |
| **nand** | dest := not ( source and dest ). |
| **nor** | dest := not( source or dest ). |

**fig. 1.4** The PS-algol raster operations.

The syntax of the raster opration is given in *fig.* 1.5.

<raster.clause> ::= <raster.op> <IMAGE-clause> **onto** <#pixel-clause>

**fig. 1.5** The syntax of the raster operation expression.

Two predefined images, referenced by the global identifiers *screen* and *cursor*, can be accessed in any PS-algol program. They can be used in the same way as any image declared within the program but the changes made to them will be displayed on the workstation screen. The cursor is a 57 by 64 pixel image in every instance of the system, and the screen is of variable size depending on the size of the window which has

been used to run the process.

PS-algol provides various functions for manipulating images and obtaining input from the mouse. The functions *X.dim* and *Y.dim* return the size of an image which is passed to them as a parameter. *Pixel* takes an image and the co-ordinates of a pixel within the image and returns the value of the specified pixel. The function *fill* takes a point within a specified image and a pixel value and fills the image with that value until either the edge of the image or a boundary of the specified value is found. A function called *menu* can be used to create popup menus. This function takes all the menu entries as images, a vector of procedures and a vector of actions to be taken on selection of a corresponding entry and it returns a procedure which, when called will display the menu and execute one of the procedures if an entry is chosen.

The language allows the user to determine the position and state of the cursor on the screen. This can be done by a function *locator* which returns a structure containing an x and y co-ordinate of the cursor on the screen image, a boolean stating whether or not the screen is selected for keyboard input and a vector of booleans indicating which mouse buttons are pressed. An example of a simple PS-algol program is given in *fig.* 1.5. This is intended to demonstrate the text output raster operations and limit expressions. The program starts off by creating a limit on the screen and drawing a box round it, by clearing an area round the limit to black and then clearing the limit itself by the *xor* raster operation. An image containing the string "text on an image" is then produced by the *string.to.tile* function using the font *fix13*. This is copied onto the limit at the position 20 pixels from the top. In this copy operation the dimensions of the limit which specify the destination of the copy are not included since the size of the area in question can be derived from the source image *a.word*. An image of 20 by 20 pixels is then declared being initialised to black which is then *copy*-ed onto 5 positions on the

limit. Note the use of the variable *xpos* which is declared in a block outwith the declaration of *ypos*. It is initialised to zero indicating that it is an integer and then subsequently changed inside the next loop, whereas the constant *ypos* is declared at the start of the block which uses, but never changes it, and may therefore be declared as a constant. This is not the most efficient method of programming this operation but has been done this way to demonstrate the difference between variables and constants in the language.

```
xor screen onto screen              ! clear the screen.

let the.limit = limit screen to 320 by 200 at 40, 40

not limit screen to 322 by 202 at 39, 39 onto
        limit screen to 322 by 202 at 39, 39
xor the.limit onto the.limit

let a.word = string.to.tile( "text on an image", "fix13")
copy a.word onto limit the.limit at 20, Y.dim( the.limit ) - 20
let a.square = image 40 by 40 of on
let xpos := 0
for pos = 1 to 5 do
    begin
    xpos := 5 + 5 * pos
    let ypos = 50 + 5 * pos
    copy a.square onto limit the.limit to 40 by 40
        at xpos, ypos
    end
```

**fig. 1.6** A simple PS-algol program.

The output from the program in *fig.* 1.6 is given in *fig.* 1.7. This shows the bottom left hand corner of the graphics window to contain a black box, which is the line drawn round the limit.

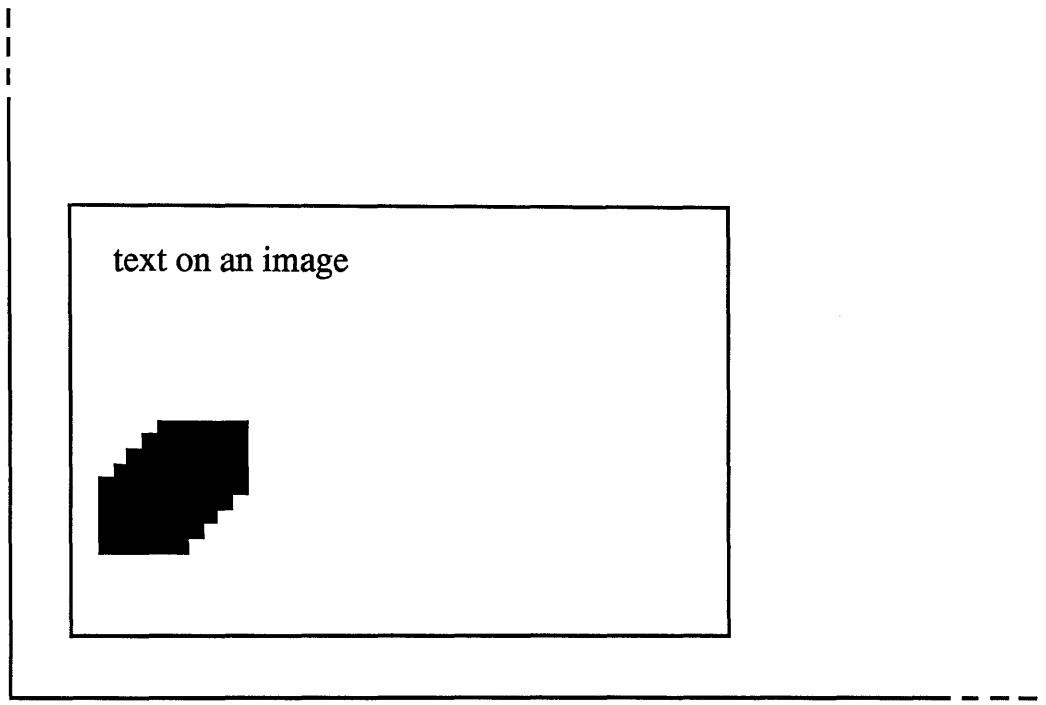



**fig. 1.7** Output from the program in *fig.* 1.6.

### 1.2.2 Output of Text.

The second form of output provided by PS-algol is text output. This can be achieved through the language in two ways, the first places the text on an image, the other places the text on standard output.

Text output onto images is implemented by a function *string.to.tile,* as described above, which takes a string and a font name

and returns an image containing a bitmap representation of the string. This returned image can be copied, or combined in any way required, with any other image the user wishes, using one of the raster-op rules. Alternatively the user can achieve text output to images by retrieving a font structure from the FONTS database and use the character images this contains to build text images.

This form of output requires that the user must keep track of the screen positions and calculate the space required to hold the *tiles* produced by *string.to.tile*. This implies extensive manipulation of screen positions.

The second type of text output on offer to the programmer is provided by the *write* statement, which writes any combination of strings, integers, reals and booleans to standard output. The syntax of this statement is described in *fig*. 1.7.

<write.list> ::= <SIMPLE-clause> {:<int-clause>}{,<write-list >}
<write> ::= **write** <write.list>

**fig. 1.7** Syntax of the *write* statement.

This does not display characters as images but just as ascii characters as on the terminal screen. The *write* statement allows the user to set out text in an organised way, specifing field widths, as an integer clause, for each item written, which make it particularly easy to use. Since it was originally designed for *S-algol*, which only required output on character devices, it produces problems in the *PS-algol* implementation in that the output from *write* statements must share the screen with any graphical output from the program, causing problems since the *write* output may appear in any position on the screen and

possibly scroll the graphics screen. These problems make the statement unusable in many programs. The problems associated with text output are discussed further in chapter 6.

## 1.3 Picture Manipulation.

In addition to image manipulation operations in integer space *PS-algol* allows the user to create *pictures* in real space. Pictures are stored as points which can be joined to each other or to a group of points. A new point or picture can be joined to an existing picture by one of two operations, "^" joins the two pictures and includes a line in the picture between the joining points and "&" joins the pictures without including a line between them. Once a picture is built it may be scaled, shifted and rotated. At any time a part of a picture can be displayed on an image the joined points being represented by lines. This operation is carried out by the *draw* function.

## 1.4 The PS-algol System.

Since PS-algol is a persistent programming language, allowing procedures to be stored in databases, most of the standard procedures provided by the language have been written in PS-algol itself and stored in the *system database*. These procedures are extracted from this database each time a program is run. This is done by the *poms* code (persistent object management system code) executed by the interpreter before a program is run. This code opens the system database and extracts the standard functions.

To allow these procedures to be treated as any other PS-algol

procedures they must be declared in the same way any other procedure is. This is done by the compiler which opens and compiles the contents of a standard declarations file. This file contains all the standard functions declared as *nullproc*s. The user code is then compiled, any mention of a standard function being treated as legal since it has already been declared. When the code is executed by the interpreter the poms code is employed to give the standard functions, previously given a nullproc value, the value contained in the system database. The language also provides some standard structure definitions which are dealt with in the same way as the functions in that their declarations are compiled from the *prelude* file processed by the compiler before the users source code.

## 1.5 The PS-algol private functions.

In addition to the standard functions there is a set of procedures called *private* standard functions which are to be used only by the system implementor. They include system functions not intended for inclusion in the language but required by the compiler and the other system software. The file handling routines, for example, are required by the compiler for the reading of source files and the creation of object files but are not to be used by the average PS-algol user since they do not comply with the principles of the language. The private functions include a number of graphics functions. *pnx.line* takes two points on an image and joins them with a single straight line of pixels. The name of this function came from the fact that it was first implemented on the Perq running the PNX 5 operating system which provides a line function which can be accessed through C. The name *pnx.line* indicates that the function calls the operating system function directly. This

function is private since it has a poorly defined user interface. The operation of the locator function, described earlier, can be altered by a function *set.locator* which changes the conditions under which locator returns. The default locator "mode" is to return from the function only when an event occurs, an event being, by default, a movement of the mouse by three or more pixels, a click of a mouse button or a key stroke from the keyboard. These default values can be changed by the *set.locator* function to instruct locator to return immediately with a manufactured screen position or to return when different types of events are raised. Unfortunately the set.locator function passes four integers directly to the Perq operating system and so, being difficult to use, it is also defined as a private function. The cursor mode can be changed by use of a similar function *set.cursor*. This function is used by the higher level public functions *cursor.on* and *cursor.off*.

The internal representation of an image contains a vector of integer vectors. It is useful to get access to one of these integer vectors, or planes, in some cases to allow fast loading or dumping of the contents of an image. To allow access to this facility the function *plane.of* is provided which takes an image and returns one of the planes as a vector. This function is utilised by the build font program which has to load an image from a font file. Since this function is machine dependent and requires that its users know the internal structure of an image before they can use the function successfully it has been classified as a private function.

All the private functions of PS-algol are documented in [PPRR11].

## 1.6 The system's failings.

The private functions are classified as such since they are not intended for use by the average programmer. This may be because they have a badly defined user interface or they implement facilities not intended to be included in the final PS-algol system but are required for the initial system implementation. The private functions are declared in a system standard declaration file which the system implementor can access when compiling his programs. This is meant to prevent a programmer accessing the private functions. Since the private functions allow access to extended file input/output routines a fast line drawing function, functions which allow access to the operating system and other routines which the programmer feels are required in programs, many programs produced depend on use of the private functions for speed and access to facilities not provided by the public *PS-algol* system. This will continue until the public routines are changed inorder that they provide the user with the power and speed required in programs.

## 1.7 The Abstract Machine.

The PS-algol system is built on top of an abstract machine which implements some of the lower level standard functions. Since a large number of the graphics functions are written in PS-algol the abstract machine is required to provide the basic functions *raster-ops*, *pnx.line*, *locator*, *set.locator*, *set.cursor*, *plane.of*, *X.dim*, *Y.dim* and *Pixel*. All these functions are implemented in C as part of the runtime system, which in most implementations is merged within the interpreter. Their implementation makes use of the existing machine library functions for raster-ops, line drawing and mouse control.

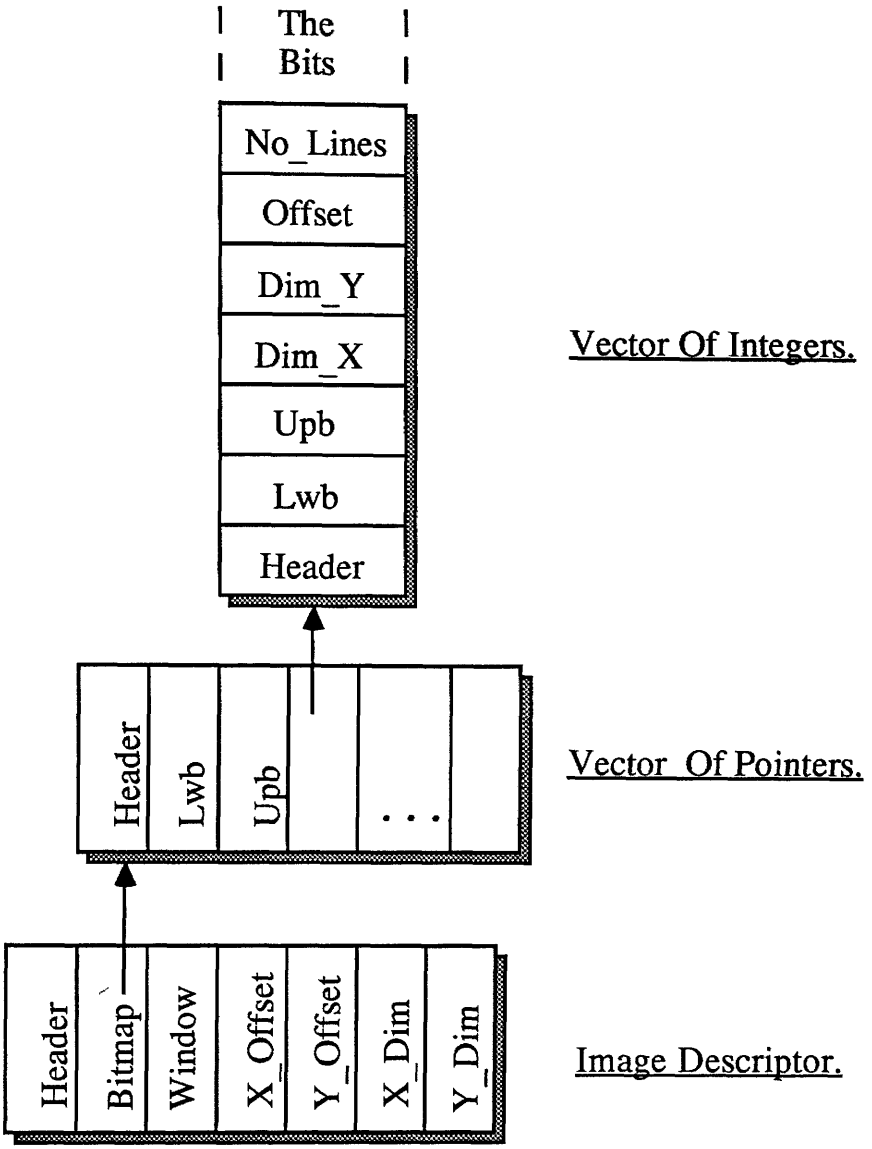



**fig. 1.8** A PS-algol image descriptor.

Every object in the PS-algol abstract machine starts with a header word which contains information on the type of the object followed by several bits indicating changes to the object, used by the poms, and a mark bit used by the garbage collector. The contents of the second half of the header word depend on the type of the heap object and generally indicate its size and, in the case of structures, gives the number of

pointers within the structure, information needed by the garbage collector.

The *image* object, shown in *fig.* 1.8, may refer to an image or a limit on an existing image and the image refered to may be an internal bitmap or a device. The image descriptor points at a bitmap, from the *Bitmap* field, and specifies the rectangular area of that bitmap it refers to by giving the co-ordinates of the starting point within the bitmap (*X_offset,Y_offset*) and the dimensions *X_Dim* by *Y_Dim.* In the case of the actual image as opposed to a limit, the dimensions will be the same as that of the bitmap and the offsets will both be zero. The image descriptor may refer to a device such as the *screen* or *cursor* in which case the file descriptor of the device is stored in the *Window* field of the descriptor and changes are sent to the device rather than to an internal bitmap.

The bitmap portion of the image consists of a vector of "planes", the number of which can vary from one upwards to allow "coloured" images to be declared in the language. The vector has the upper and lower bounds fields which indicate the number of pointers to planes it contains followed by the pointers themselves. A plane is, in reality, a vector of integers which has been used to hold information on a bitmap and its associated pixel information. The first header word is followed by the upper and lower bounds of the vector, which in this case are used to calculate the size of the vector. The first four integer entries in the vector have been used to hold its dimensions, the offset from the start of the pixel information and the number of lines in the image. The rest of the entries are used to hold pixel values.

**let** example.image = **image** 40 **by** 60 **of on** & **off** & **on**

**fig. 1.9** A declaration of a multi plane image.

A declaration of an image with three planes is given in *fig.* 1.9, the bitmap of the image structure produced by this image will have three planes represented by a vector of three integer vectors. Each of the integer vectors contain a monochrome bitmap which starts after the entries indicating the dimensions of the bitmap.

Objects such as the image descriptor described above are stored and manipulated in the PS-algol heap. They may be moved at any time by the garbage collector to provide space for new heap objects. In addition any object may be stored in a *PS-algol* database. This heap organisation is essential to allow the dynamic allocation of the *PS-algol* data objects and the easy manipulation of data structures from within the language.

# 2. Some Other Graphics Languages.

In an attempt to find the requirements of a graphics programming language, several languages and extensions to languages were considered. Outlines of some of these are given in this chapter. The reviews range from early extensions to higher level bitmap oriented languages, where the graphics facilities are designed as an integral part of the language for the purpose of building a system. The first languages presented were implemented for line drawing displays, they are followed by those designed for bitmap displays. The order in which the languages are considered follows hardware trends as these are away from calligraphic systems towards the bitmap workstation.

## 2.1 Euler-G.

Euler-G [NEWW70] is an extension of the language Euler to permit interactive graphics.

All graphical operations are carried out in real space, the programmer therefore must specify a window onto this space and its associated viewport on the screen, an operation carried out as described in [NEWW79].

Line drawing may be done relative to the current beam position or to an actual point by the operations LINE and LINETO. In addition to plain line drawing there are two operations DOT, for drawing dotted lines, and ZIP used when curves are being drawn since it rounds the ends of lines. To allow operations relative to the current beam position the language supplies the MOVE and MOVETO statements for the

manipulation of this current position. All the operations described take one co-ordinate pair which refers to real space.

Text output on the graphics screen is produced via the DISPLAY statement, designed to take the same parameters as the language's PRINT statement used to output text to a teletype. DISPLAY outputs text, at the current beam position, formatted in the specified manner.

In order to allow repeated objects to be manipulated the language provides a display procedure facility. A display procedure creates and draws an object, allowing calls with a specified position and possibly operations such as rotations or enlargements in order to produce the required display.

The library function SMOUSE when called returns a five element list containing the status of the three mouse buttons and the screen co-ordinates of the cursor. The function returns only when a mouse button has been pressed. The two screen co-ordinates returned from SMOUSE may be passed to the HIT function which will indicate which object, if any, has been pointed at. To allow objects to be manipulated by name each call of a display function can be associated with a unique name, it is this name which may be accessed through HIT.

Euler-G has been designed with a refresh display in mind, producing several interesting effects. Display procedures produce "objects", rather than plain output, allowing easy object manipulation abstracting away from screen positions. The locator functions provide access to object names rather than screen co-ordinates which saves the programmer working out this information from the co-ordinate pair; but the locator function is limited to returning a value when the button is pressed.

As an early graphics language Euler-G has not got the powerful attributes found in many more recent languages but has the basic facilities

still present in most languages today.

## 2.2 LOGO - an extension for graphics.

This system [NEWW73], based on the LOGO language, was designed for teaching in such a way as to be simple, flexible and inexpensive, it therefore has reduced power of expression and speed of execution.

LOGO attempts to provide the user with a full programming environment. The language itself keeps the number of available constructs as few and simple as possible, a trend which is demonstrated in the graphics facilities.

The LOGO screen is accessed as a 1000 by 1000 integer grid, since real space was found to complicate the language excessively.

The line drawing facilities are a subset of those provided by Euler-G, previously described, continuing the notion of current beam position with relative and absolute moves. Text output is handled by the DISPLAY statement which may only take strings.

Display procedures are implemented allowing a procedure to "draw" in its own space which can subsequently be mapped onto any specified space by the DRAW statement which specifies a window on the screen intended to hold the output of the procedure. This facility allows the programmer to specify the bounding box of a shape on a procedure call rather than actual points to be used by the procedure and allows procedures to be used without any knowledge of the space they were defined in. An example LOGO program is given in *fig.* 2.1.

When the final line of the example is entered to the system the HOUSE procedure is called and any output, specified to be between (0,0) and (1000,1000) in the procedure declaration, is mapped onto the area of

the screen between the points (300,300) and (700,800).

```
TO HOUSE
10  MOVETO 0 500
20  LINETO 500 1000
30  LINETO 1000 500
40  DRAW BOX IN  "0 0 1000 500"
END
DRAW HOUSE IN "300 300 700 800"
```

**fig. 2.1** An example LOGO program.

The draw command creates a new segment which contains the output from the procedure. If another "DRAW HOUSE" statement were executed, in *fig.* 2.1, the original house would be replaced by the new output. New instances of HOUSE can be created by use of the ADD statement, which allows a new segment to be created. Segments may be changed or removed without effecting other segments.

Input to the program is provided by REQUEST which takes a string from the keyboard and PENPOINT which reads a co-ordinate pair from the tablet when the pen is pressed on the surface, a restriction similar to that imposed by Euler-G but in this case due to inabilities of the target machine.

The language intends the programmer to make use of files to store pictures as a list of records, each record would contain the parameters of a procedure call. For this purpose a file handling system is provided allowing reading and writing of records, test for end of file and file creation and deletion.

Again this language was designed with a refresh display in mind and

so provides the expected object manipulation and locator functions. The graphics are simple and concise, in accordance with the rest of the language, but some operations are nevertheless limited by the intention of running the system on a low power machine. The simplicity of the language would make it easy to learn and use but could become limiting if an attempt was made to write complex programs. Although Euler-G and other languages had used the idea of display procedures previously LOGO was the first language to bring this method of picture building into popular use, due to the simple, succinct implementation represented by the language.

## 2.3 MIRA - A graphical Pascal extension.

Mira [MAGN81] is an extension to Pascal to allow the building of graphical output by the use of graphical types. It is designed in such a way as to fit in with the existing type system of Pascal.

The two basic graphical types introduced to the language, which may be used to build more complex types are :

The **vector** type - A vector consists of two real co-ordinates and may be manipulated as any other variable. Full vector arithmetic has been implemented using vector addition and scalar product. Vectors may be read and written, and are declared as follows.

```
VAR
    V1, V2 : VECTOR;
```

The **figure** type - This is the most important addition to the language. A figure type contains the *definition* of a shape i.e. the commands required to draw the shape. The syntax being similar to that of a procedure. A type *quad* and two variables of this type are declared below.

```
TYPE
    QUAD = FIGURE(X1,X2,X3,X4 :VECTOR );
        BEGIN
        CONNECT( X1, X2, X3 ,X4 )
        END
VAR
    Q1, Q2 : QUAD;
```

Two statements specific to figure type definitions are CONNECT, which joins any number of vectors in order that a shape may be built, and INCLUDE which joins the output of one graphical type to that of the calling type. The INCLUDE allows a graphical type to be built from many sub types.

Since variables of figure types are held on the heap they must be created in order that their representation may be added to the heap. The statements described below are provided for heap manipulation.

```
CREATE <figure> (<actual parameter list> )
DELETE <figure variable>
```

The create statement requires the variable type to be stated together with the actual parameters declared in the figure type declaration. An example is given below :

```
TYPE
    CIRCLE = FIGURE(CENTRE : VECTOR; RAD: REAL);
        BEGIN
        - - -
        END
VAR
    ACIRCLE : CIRCLE;
    APOINT  : VECTOR;

BEGIN
    APOINT := « 2.5, 3.5 »;
    CREATE ACIRCLE(APOINT, 10.5 );
    DRAW ACIRCLE
END.
```

The DRAW statement is required to display the figure.

The program's window and viewport may be changed via the standard procedures SCALE, FORMAT and MAP. These have the functionality described in [NEWW79] but only control one window and viewport.

The language abstracts well from the low level routines described in Euler-G but lacks the functionality required by a graphics language, in that, for example, the line drawing routines are limited, no graphical text output has been included in the language, an essential part of any graphics language, and no proper provision for a locator input has been made, only the function *inputg*, which returns structures of different type in different implementations of the system. A consistent locator function is required before this part of the language can be used for serious programming.

## 2.4 Graphical Kernel System.

The Graphical Kernel System (GKS) is a standard machine independent graphics package. It has been designed in such a way as to treat different types of display and input device in a consistent manner. Implementations have been produced for systems ranging from refresh displays with light pens to mouse based bitmap systems. The system is described in [HOPF86] and defined in full in [ISO82].

Every GKS facility is accessed via function calls which either execute graphics operations or manipulate global variables which alter the result of the graphics routines.

GKS implements line drawing by the function *polyline* which draws lines between a number of points provided in arrays. The x and y co-ordinates of a specified number of points are passed to the function, all points being actual co-ordinates, as opposed to relative co-ordinates, since GKS does not support the notion of current screen position. The type of line produced by *polyline* may be changed by making a change to the *polyline index* via the function *set polyline index.* An integer is passed to this function rather than a machine dependent line description. The actual meaning of the integer passed to this function may be set by the function *set polyline representation* which allows the colour, the workstation name, the line type and line width to be associated with a particular index. The two functions are kept separate by GKS to allow machine dependent parameters to be kept at a different level from the actual graphical operations.

GKS allows specified areas to be filled. The area bounded by an array of points is filled with the current colour, specified by the *fill area index.* This index may be manipulated in a similar manner to the

*polyline index*, in order to achieve different interior patterns and styles, bounding line styles etc.

Text output is performed by a function *text* which places a string at the desired point. The character height, slope direction and alignment may be set globally by function calls in order to achieve the required results from the *text* function. Text may be output at any angle in any font or orientation by a combination of function calls.

Graphical input in GKS is obtained from a combination of six logical input devices. These may be mapped onto the existing physical input devices in order to provide device independent input. The use of logical devices is the method used by GKS to enable the same functionality to be displayed by each implementation. The *string* device returns a text string, this device will usually be implemented by the keyboard. The pointing devices are represented by logical devices *locator*, *stroke* and *pick*. *Locator* returns a single pair of screen co-ordinates and could possibly be implemented by a mouse, tablet, light-pen or other such device. The *pick* device returns an object rather than a position, this object will be produced from a segment and may be manipulated by name. The *pick* device is idealy suited to machines which store displayed data in segmented storage. The *stroke* device returns a sequence of positions passed over by the pointing device since the last call to the function. *Stroke* could possibly be implemented by most of the physical input devices already mentioned.

The GKS view of input devices removes the machine dependence from devices and allows different types of device to be viewed in the same manner e.g. a tablet may be treated in the same way as a mouse at the program level.

Since GKS is required to cater for all types of display, segmented storage manipulation has been included in its operations. A segment may be created and filled with a drawing, once closed it may be

manipulated through function calls which change its visibility, priority or highlight the resulting drawing. The contents of a segment may be transformed by allowing every point to be changed by a transformation matrix allowing drawings to be enlarged, moved etc. before being displayed.

GKS attempts to provide device independent access to graphics facilities. In doing this it "hides" the capabilities of some machines. The full potential of bitmap machines are not realised by the package in that many algorithms for bitmap manipulation, as described in [GUIL82], require additional internal bitmap storage in order that they may be implemented efficiently. GKS does not allow this facility therefore is not fully suited to bitmap manipulation.

The *text* function and its associated control functions presents a fully device independent text primitive. The functionality of this operation covers a lot of the possibly envisaged requirements of text output; it is therefore one of the successes of GKS.

The methods the package uses to abstract from input devices, especially the locator type, "hide" the physical characteristics of particular machine implementations effectively. Considering the different functions of the locator device as the two logical devices *locator* and *stroke* separates the possible functions of one physical device well and reduces the problems associated with locator mode discussed in chapter 8.

GKS is a step towards a high level graphics language, but cannot be considered as a final solution. Possibly its difficulties arrise from attempting both caligraphic and raster graphics in one interface. The package avoids the concepts, such as raster combination rules, not implemented by the two types of machine, therefore users loose the possible advantages gained through use of this facility. A similar

graphics package, providing operations solely for bitmap machines, which could therefore make full use of a typical workstation's potential might be very useful.

## 2.5 Amber.

Amber, as a whole, is an attempt to produce a complete programming environment for a bitmap personal computer with a pointing device. The *whole* system will be written in terms of the language Amber itself. Amber contains many advanced features such as concurrency, persistence and graphics. "Graphics is an essential part of the language." [CARL84]. The inclusion of graphics routines is to allow the writing of the system support environment including the editor and windowing system.

Amber allows the programmer access to the bitmap representation of the screen since it is intended to be used on a bitmap workstation. An Amber bitmap is a rectangular array of pixels which can have one of the two values *true* or *false*, black or white. They are produced by the function *bitmap* which when passed four integers, dimensions and the co-ordinates of the top left hand corner, returns a new bitmap with all the pixels set to false. The four integers passed to *bitmap* may be retrieved at a later date by the function *bitmapTile*. The pixels of a bitmap may be indexed individually by their co-ordinates, these may be passed to the *pixel* function together with the bitmap to determine the boolean value of a particular pixel.

The raster-op facilities of Amber are provided by the *bitblit* function which takes two bitmaps and the six integers required to specify a rectangular area on each one, a width and height and two sets of co-ordinates. The final parameter, an integer called *op*, specifies which

one of the eight raster combination rules is to be used during the operation.

In order to allow "shading" of regions on a bitmap a function *texture* is provided. *Texture* takes a pattern bitmap and a destination bitmap together with the four integers specifying a rectangle within the destination. Again a combination rule must be specified. The result of a call of the function is that the pattern bitmap is replicated, in a correctly aligned manner, throughout the destination area specified.

Line drawing is accessed by another function, *line*, which takes a bitmap and five integers specifying the co-ordinates of the end points and the combination rule. A line is drawn between the specified points on the bitmap.

Graphical output from the language is achieved by gaining access to the screen bitmap through a call to the function *screen* and the cursor bitmap via *cursorIcon*. These two bitmaps may then be changed as any other, with the results of the operations displayed on the screen. The cursor tip, or hot spot, and the actual screen position of the cursor may be changed by calls to the functions *cursorTip* and *setCursor*.

Graphical input may be obtained from the mouse by use of the functions *cursor*, which returns the screen position of the cursor and *button*, which returns a boolean indicating the pressed state of the mouse button.

Amber's graphics provide the minimum set of facilities required for writing serious bitmap manipulation programs, since "higher level" operations could possibly be written in terms of these "primitives" as required, a difficult task when the operation of the locator is considered. This function may be too low level for serious use, although the concurrency of the language may be used to control the locator input.

In order to keep the Amber compiler as simple as possible all the

operations have been included in the language as functions, which can, in the case of *bitblit*, require the programmer to specify many integer parameters. This has the effect of producing unclear code and makes the task of programming even harder.

## 2.6 Graphics Libraries on Single User Systems.

Single user machine designers and software teams provide the user with a set of libraries of window management and graphical operations accessible through the system language.

Here such an extension, to the system language C, which is provided with the Whitechapel MG-1 workstation is reviewed. The libraries are defined in full in [STEN85] and discussed in [NEWW85]. The libraries used for graphics and window manipulation on the MG-1 are considered to be typical of such machines. The Apple Macintosh and Sun workstations come with similar language extensions.

Some of the work described in chapter 3 and the implementaion in chapter 4 was carried out using the MG-1 graphics facilities and so it seems appropriate to describe the facilites used in this section.

The user of the graphics libraries is given the option of accessing the functions at several levels of abstraction. The highest level allows comparatively easy creation of windows and bitmap manipulation. These higher level operations are built from the low level routines which are closer to the actual machine implementation of the bitmap screen. The user may wish to use the lower level routines to avoid overheads incurred by the window manager or avoid the restrictions imposed by this higher level view of the machine's capabilities.

The upper level operations are contained in a library, called the tool library, as a set of functions. These include operations which allow

the creation of a window and interaction with the window manager to change the size and position of the window. Stow and unstow routines reduce the window to an icon and then recover it on request. The WinCreate routine has ten parameters which describe all attributes of the window - its initial size and position, the window title and title font and a set of parameters describing its associated icon. To use this one routine requires extensive knowlege of the window manager if full advantage is to be made of the possible options. Default values may be assumed in some parameters of the function call by passing null parameters but the complexity of the function is never hidden from the naive user, who may only wish to specify the size and initial position of the window. The package is not designed to cater for such a "naive" user but for one who requires as much power as is possible and the highest functionality available from the machine. This system enables any specifiable attribute of a window to be set and maniplated from programs in a fully explicit way. Before a created window is displayed it must be associated with a Raster, a bitmap, which may be dynamically allocated for this purpose. This operation which, it seems, could have been done automatically by the window creation routine is left to the user in order to allow several Rasters to be associated with one window. One of these bitmaps is "selected", to be displayed, at a time allowing switching between several displays easily. This lower level of abstraction has been provided here again to allow a more flexible functionality. Unfortunately a user not concerned with multiple displays has to do as much work as one intent on using the full facility. For reasons of implementation efficiency the lower level of access is not hidden from the tool library. The user must explicitly "update" areas of the window raster which have been changed before these changes are displayed on the screen, a property of the system which allows a large number of changes to be made before the over head of communicating the new

Raster to the screen is suffered.

The tool package provides enough power for most users by allowing full access to the machine's window manager, abstracting away from the machine level as far as possible without seriously effecting efficiency. Users who do not require the provided window manager or who are willing to lose the functionality of the tool package in order to gain higher program efficiency may access graphics functions at the panelist level, a package, which was used to write the tool package routines, gives a view of the screen at a much lower level. Panels are created instead of windows, they are not as flexible and are limited in size and shape by machine constraints, but have the advantage of not requiring explicit updating and other overheads so provide a faster graphics display.

The user is not constrained from using inappropriate mixtures of graphics calls from the different levels of software. For example panelist and tool library functions could be mixed with unpredictable results.

The graphics routines, provided by a graphics library, may be accessed at either level of abstraction described since they act upon the Raster structure associated with a panel of a window. The library supplies routines which draw lines, arcs and circles on rasters at specified points, flood fill shapes or bounded areas. Text output is provided by functions similar to those of C, with the additional requirement that a destination Raster, font, combination rule and start position are specified. The font passed to these functions must be explicitly read from a font file holding a particular format of font. Any user wishing to output a simple string in any font must still go through the process of reading a font file, thus complexity is forced into every program. The graphics library includes a RasterOp function which takes seven parameters explicitly describing two Rasters and associated

areas to be processed in addition to a combination rule. The function provides all the functions of the standard raster-op but again assumes no defaults. The graphics library provides a reasonable set of functions but can be complex to use since the full functionality of the operations is not defined.

The MG-1 libraries are not designed to provide a clear, concise environment for its users but to allow access to the machine's capabilities at a reasonably high efficiency. Programs may be written which communicate with the window manager, enabling users of programs to interact with dynamically created windows, while still exibiting tolerable runtimes. New users to the system find the reasons for the various levels of abstraction obscure and the use of functions error prone. Simple programs are made complex by the necessity to specify and set up facilities irrelevant to the user's application, and indeed there is no easy way of doing simple graphics.

The MG-1 libraries do in fact allow access to excellent graphical manipulation facilities, despite the overhead, in programming terms, required to set many of them up. Programmers are allowed to integrate their created displays with the MG-1 window manager, thus presenting a familiar environment to their users. It is the richness of the facilities potentially provided that attracts so many programmers to such packages, they are willing to put up with disadvantages of such a system in order to experience the advantages.

## 2.7 Smalltalk-80.

Smalltalk-80 [GOLA83] is an object oriented language supporting a complete programming environment. The language allows actions to be taken on objects by sending messages to them, having the effect of

returning some other object or producing a change in the target object. An example of an object declaration is given below.

```
| exampleImage |
exampleImage <- Form new extent: 80@40.
```

The first line declares a temporary variable. *New* is a message which is sent to the object class Form ( a bitmap type of Smalltalk ). *New* tells Form to create a new bitmap, the sizes of which are indicated by the *extent:* message. The result of this operation is that Form returns a new bitmap of 80 by 40 pixels and assigns it to the object *exampleImage*. Further messages may now be sent to *exampleImage* in order to display it or change areas within it.

```
exampleImage displayAt:100@20.    "shows the contents of
                                  exampleImage on the screen
                                  at the specified position."

exampleImage reverse displayAt:100@20. "displays the image, as
                                        before, but reverses each
                                        pixel before doing so. "

exampleImage darkGray.            "shades the image with a
                                  halftoned pattern."

| colourBox |
colourBox <- Rectangle origin:20@20 extent:50@100.
exampleImage lightGray:colourBox. "shades the area of
                                   exampleImage specified
                                   by the colourBox variable."
```

If no area is specified in such a statement, as in the previous example, the whole image is operated upon by default. Several other shading messages are available to the user, allowing a choice of halftoned patterns: black, white, gray, lightGray, darkGray, and veryLightGray. The operation carried out above should be considered in the following way. The object colourBox, a rectangle, is being used as a mask by exampleImage, changes occur only on the area where the mask overlaps exampleImage. The user may provide his own mask bitmap, if this is smaller than the area specified it is tiled over the destination.

*displayOn:at:* may be sent to an image indicating that it is to be displayed on another image. Rectangular areas within the two images may be indicated along with a mask bitmap and combination rule. An example is given below.

```
exampleImage displayOn anotherImage at: aPoint
                clippingBox:aRect rule:bitOp mask:maskRect
```

The full form of the *displayOn:at:clippingBox:rule:mask* message given is equivalent to the raster-op functions found in many languages, but the fact that most implementations allow default values to be assumed by omission of clauses, simplifies it greatly.

Line drawing in Smalltalk is done by the use of objects of Class Pen. A Pen is formed as below:

```
| bic |
bic <- Pen new defaultNib:2.
```

*defaultNib:* tells Pen to use the default nib pattern, a 2 by 2 image, this is

then assigned to *bic*. Messages may now be sent to *bic* to draw lines, *home* moves *bic* to the start of the image, *up* and *down* control whether or not movements of *bic* leave a trail, *turn:*, *north*, *south* etc. change direction of *bic*, *goto:* moves *bic* to a specified point and *place* changes the current position of *bic* without leaving a trail on an image. If a line is drawn it is made up of many instances of the image held in the nib.

Mouse input in Smalltalk is accessed through the object Sensor. Appropriate messages may be sent to Sensor to obtain the type of input required.

```
| box  mouseAt |
box <- Form new extent:5@5.
[Sensor redButtonPressed] while True:[
                    mouseAt <- Sensor mousePoint.
                    box displayAt: mouseAt ].
```

The code starts by declaring *box* and *mouseAt* as variables, then makes *box* a 5 by 5 image. The while loop continues as long as the Sensor returns a true value for the pressed status of the red button. The block of code inside the while loop sets *mouseAt* to the current mouse position and places a *box* at this position, so a box will be displayed at each mouse position returned while the red button is pressed.

The two other mouse buttons, yellow and blue, may be accessed in the same manner as the red button. *waitButton* waits for any button to be pressed and then returns with the current mouse position as a co-ordinate pair, *waitNoButton* waits for any button to be released and then returns the release position. These two messages may be used together to allow the user to specify a rectangular area on the screen, an operation also provided as an abstraction from the Sensor object by the Rectangle object.

```
area <- Rectangle fromUser.
```

will prompt the user, by changing the cursor, to indicate a rectangle on the screen by dragging the mouse with a button pressed over the required area. This abstraction hides the Sensor object from the programmer and has the effect of making his code concise and more readable.

Sensor accepts further messages which change the current cursor image and cursor tip.

Keyboard input is obtained one character at a time also via the Sensor object.

```
| name  index  temp |
name <- String new:20.
index <- 1.
[ index < 20 ] while True:[
                        temp <- Sensor keyboard.
                        name at:index put:temp.
                        index <- index + 1].
```

This code will read in a 20 character string from the keyboard. Other Sensor messages allow inspection and manipulation of the keyboard buffer *flushKeyboard*, *keyBoardEvent* - returns true if a key has been pressed and *keyboardPeek* - returns next character without removal from buffer.

Text may be output from Smalltalk by sending a string the *displayOn:at* or *displayAt* messages.

```
| name |
name <- String new.
name <- 'John'.
name displayOn:exampleImage at:10@100.
```

has the effect of writing name onto the Form *exampleImage*. The Macintosh implementation used allows multifont output. The notions of current font, style and point size are used. These current values may be changed by commands such as:

```
Mac textFace: Mac bold ; drawString: 'Bold'.
Mac textFace: Mac shadow; drawString: 'Shadow'.
Mac textFont:4 ; drawString: 'A new Font'.
```

Smalltalk provides a large range or graphics facilities, only a subset of which are described here, allowing the programmer to manipulate bitmaps and receive input any way required. The programmer will be able to express any requirements in the language and not be forced to change these requirements to fit the language.

The Smalltalk technique of considering objects to be changed by message passing allows the full capabilities of an object to be hidden from any user not requiring all the functions, resulting in a system which is only as complex as the user requires. Users of the system require only a little knowledge at first and may build on this with experience. The graphics facilities of Smalltalk may be accessed at several levels of abstraction allowing irrelevant options to be omitted. The example of the raster-op given above demonstrate this, at the top level no combination rule need be stated since this is irrelevant in most uses. Smalltalk, by providing such a large number of facilities, makes its use without a decent manual difficult.

It seems that in order to provide all the users requirements in full, a language must grow to this size.

## 2.8 Andrew: A distributed personal computing environment.

Andrew [MORJ86] is an attempt by The Information Technology Center (ITC) at Carnegie-Mellon to provide a complete machine independent programming environment. As part of the environment it presents the users with a fully functional window manager and a set of utility programs exploiting the systems capabilities. One of the main features of the system is to allow the programmer to use the window manager and manipulate graphics via a set of primitives. There are about 70 procedures for this purpose presenting text output, drawing primitives, input primitives and multiple window creation. Externally the window manager tiles a set of windows onto the bitmap screen in such a way as to avoid overlaps, it may have to change the size of windows to enable all open windows to be displayed. Application programs run under the system must detect these size changes and redraw their display to fit the new size of display. The user of the Andrew system, on creating a new window, will see the existing windows being moved and their size and the size of their content being changed in order to accomodate the new window.

Andrew, in addition to the "primitive" graphics facilities, provides graphics facilities at a higher level tool-kit interface which presents easy access to facilities, through data types, such as a *scroll bar* and considers text displays as *documents*. Using such facilities simplifies the job of the applications programmer, preventing various programmers duplicating the same work in different programs. If all application programs make use of the same output packages the users of these programs will be

continually presented with the same concepts and terminology, introducing consistency to the user interface of the system. The consistent user interface in system programs will simplify a new user's learning task; once one program is mastered some of the skills gained are relevant in the use of other programs. Workstations running the system can be finely tuned to run these particular operations as efficiently as possible, whereas it could not be tuned to run each of a large number of operations quite so efficiently.

Andrew demonstrates the advantages which could be gained from a standard window management package in both implementation efficiency and user efficiency. The two level approach to graphics library routines allows system programs to be written very easily, in that all the common operations have been factored out and implemented by library routines, while still allowing complex, lower level, graphical manipulation programs to be written under the same system. Most graphics programmers require, or see it as advantageous, to use a window manager to allow several processes to share the bitmap display or to organise their output in a hierarchical manner. Andrew presents a novel approach to window management; by disallowing overlapping windows and introducing automatic resizing, the complexity presented to the user is reduced but the complexity seen by the programmer is increased. Some rearrangements of windows will require many processes to resize their displays which could take a great length of time. Window management should pose a very low overhead on the system while presenting a display of maximum flexibility and simplicity to both the programmer and the user of the system. A more "usable" system could possibly be provided by the overlapping window approach. Window resizing would be a lot less frequent since the user, or programmer, would only do this when it was essential, and indeed in many cases

windows not being used may simply be placed in the background until required rather than be reduced in size to fit on an unused area of the display. Overlapping windows allow the user of the system a larger number of windows open at any one time and are thus a lot more flexible than that proposed by Andrew.

Andrew is an interesting development and possibly presents the best type of access to graphics on a workstation seen so far. The interface with the window manager is an essential part of any graphics programming language and should be incorporated as efficiently as possible. Andrew may not be the ultimate solution to the problems of graphics programming but goes part of the way to solve them. The facilities of Andrew have been expanded in a recently developed descendant of the system called X.

## 2.9 Conclusions.

The early languages described, Euler-G and LOGO, only provided the programmer with the very basic facilities required for drawing pictures resulting in a "low level" programming style. GKS takes the ideas developed in the early languages further and presents them in a standard form together with some of the operations for bitmap machines used today. The later languages, Amber and Smalltalk-80, have been designed in such a way as to provide the basic facilities necessary for building a system which could provide higher level operations in terms of the language. Andrew incorporates full window manager access in a graphics environment designed to make the most of bitmap workstations in every respect. The operations provided by these later, high level languages, give the programmer the power in graphics programs which other, non graphical, attributes of a high level language provide for

conventional programs. The user need not be concerned with irrelevant details and may use the provided operations at any level of abstraction required in order to gain access to the type and complexity of operations needed.

Smalltalk attempts to provide an abstraction away from the conventional view of bitmaps and raster-ops which hides the machine representation and simplifies the users view allowing concentration on the main function of the program being written. Many users will be able to use the system at the higher level of abstraction avoiding use of the raster-op facilities directly, but these still have to be accessed to write display operations and so are not completely hidden.

Mira hides the display device being used from the programmer by allowing access to it via graphical types which abstract away from the machine level view of graphics. Mira allows a user to express shapes and produce various instances of them simply and succinctly. The language is ideal for producing diagrams but lacks the input/output power required for serious interactive graphics. The writers of Mira produced a second language Mira-3D [MAGN83] which allows vectors to be considered in 3 dimensional real space enabling the user to produce representations of solids easily. This was the best 3-D vector processing language reviewed allowing solids to be manipulated as easily as 2-D shapes. The original Mira was written in such a way as to make this extension, a natural progression in the language, easy to develop.

Andrew considers the workstations capabilities as a whole, and rather than consider a subset of the users needs, makes available window management operations usually only accessed from within a system language extension such as that described for the Whitechapel MG-1.

Several languages considered have not been described in this chapter since they were thought to present no new constructs or ideas not covered by other languages. These are described below.

[YIPC84] presents an extension to Pascal based on the CORE graphics package. This extension gives the user a lot of power allowing a variety of texts, text orientation styles, line styles and intensities. These options are chosen by setting global values by use of procedure calls before calls to the line drawing and text output procedures. The large number of procedures provided by this extension give it good expressive power while the operation carried out by each is simple and can be clearly defined. However the large number of procedures may make their interactions hard to understand and define. The language allows interactive graphics by the use of menus and a locator.

[NAMN78] proposes a graphical extension for high level languages which is demonstrated to be language independent by describing how it may be accessed through several languages. The consistency in the primitives presented in the various extensions allows new graphics programmers access to graphics through the language they are familiar with, saving retraining in a new language as well as the graphics extension.

[DENE75] describes GRAPHEX68 which allows the building of graphical data structures via an extension to Algol-68. Pictures, once built, may be rotated, scaled and combined with others in order to produce new pictures.

These language extensions have a similar functionality to that of

Euler-G and LOGO but obviously achieve the same ends by use of different syntax and build types of data structures specific to their host language.

The graphics facilities provided by either an extension to an existing language or by a language designed with graphics in mind must allow access to a large variety of functions allowing operations to be carried out in a variaty of styles and modes, otherwise programmers will be forced to simplify the specification of their program to allow it to be expressed in the language being used, which has the effect of encouraging inferior software and reducing the power of programs implemented in the language. The syntax of the operations and language constructs which allow access to the graphical operations must be readable and preferably allow the user to consider them at various levels of abstraction as described earlier, allowing default values in operations to be assumed where the user is not concerned with their value or perhaps even their existence. Accessing operations at a variety of levels makes simple operations a lot more concise than the equivalent operations expressed at more primitive level. Early programming languages, such as FORTRAN, required that the format of text and numerical output be specified precisely whether or not the user was concerned with the layout resulting in unnecessarily complex code and wasting the programmer's time. Pascal allows the user the option of specifying field widths but assumes default values on their omission, giving the same functionality as FORTRAN, but producing a system which is tidier and easier to use. The same type of progression must be made in graphical operations to simplify graphics programming, making the languages easier to learn and use.

# 3. Programming Experiment.

In order to evaluate the existing PS-algol system it was necessary to carry out a project which made use of the facilities provided by the language. An implementation of a preview program for the typeseting system TEX [KNUD84] was decided upon. The TEX typesetting system takes an input file containing type setting commands and text and produces a device independent output file, or *dvi* file, which contains the necessary information required to display the desired output on any device, whatever the output method or resolution. The project involved writing a processor for this *dvi* file which translated the commands it contains into a bitmap representation of the required text; a task which involved reading in the characters from the correct font files and copying these onto the screen at the required places. Since the fonts could be stored in a database the chosen project gave a good insight into PS-algol from the programmer's point of view.

Such a system had previously been implemented in C on the MG-1 workstation so comparisons between the two languages could be drawn during the work.

## 3.1 The Problem.

The *dvi* file produced by TEX contains information on the fonts used in the file and some other general information on text layout, page numbering etc. The rest of the file consists of commands which describe the output in a device independent form, for example *SET 10* means

that character 10 of the current font must be placed at the current position on the page, *RIGHT 6* means that the current position must be moved to the right by 6 units. Neither of these commands make any mention of pixels or any particular device and indeed this is the case in all the commands, thus the device independence of the *dvi* file.

The job of the TEX preview program is to go through the file and interpret each of those commands in turn translating the device independent units to pixels, a conversion which will depend on the resolution of the display device. When a font definition is found in the *dvi* file a calculation must be done to determine which font file should be read. The previewer takes the name of the font from the *dvi* file and decides which of its own font files contains it, since each font comes in several sizes for use on different devices and in order to represent different magnifications.

## 3.2 Implementation In PS-algol.

Most of the PS-algol program could be obtained by translating directly from the C version written earlier allowing full concentration on the peculiarities of PS-algol rather than on the design of the system. The major difference between the two programs was to be the structure used to hold the fonts when read in from the font file. In the PS-algol implementation each font was held in a structure which contained a vector of characters. A character was a structure which contained a few integers and an image containing the bitmap representation of the character. This structure was built in such a way that it contained all the information in the font file in an organised easy-to-follow manner.

The preview program read in all the fonts required from the font

files at the start of each run, a process which involved building the structure described above from each of the font files, a slow process, as each character, stored in the font file as a bitmap, had to be read in byte by byte and an image representation of each byte produced by using the raster-op facility to copy a pixel onto the character image each time a *one* was found in the file, thus a raster operation has to be executed for almost every bit in the file.

Since the building process was time consuming it was decided to write a program which would read a font into the structure which could be stored in a database. The preview program was then converted to take the required font structure straight out of the database rather than building it from the font file itself. The result of this was that there was no delay while fonts were read in and the total execution time of the program was greatly reduced, in fact it approached the execution speed displayed by the version written in C despite the fact that the PS-algol system is implemented by an interpreter. The favourable performance of the new version of the PS-algol program is an example of the high level language allowing major global optimisations by presenting the user with design constructs which give help in producing efficient programs. The HLL makes approaches simple which would not be considered in a system language because of their complexity. This particular optimisation is one commonly made possible by persistence provision; that is, an activity is factored out of each program run, by building a data structure which holds the result of that activity, and preserving it in the database.

## 3.3 Implementation in C.

The C version of the preview program built a font data structure from each of the font files required in the processing of the *dvi* file at the start of a run. The reading in process was fairly quick since the data structure built to contain each font was simple, in that the bitmap for each character was stored as a string of bytes of pixel information rather than in an organised structure such as an image used in the PS-algol version. The effect of the simplified data structure may have produced a fast building algorithm but the use of the data structure in displaying characters on the screen was slow because significant processing was required for each character displayed compared with just copying in the PS-algol version.

## 3.4 Comparison of C and PS-algol.

Several major differences were found between the styles of *C* and *PS-algol*. The major differences found in the preview program were in the handling of data structures. The example of the building of the character image from the font file is used again here. Since C represents the image, or raster structure as it is called on the MG-1, as a pointer to an area of memory and the programmer can freely change memory locations, it was easy to access any byte of the image and change it, therefore bytes could be copied from the font file directly into the image without any computation. This easy access to data structures allows fast building which makes the most efficient use of the capabilities of conventional machines. PS-algol on the other hand restricts access to data structures, such as images, to conventional methods such as raster-ops. This leads to tidy programs which are easy to follow and

maintain, although they will not give quite the speed of execution allowed by C.

C on the whole is a far more complex language than PS-algol in that operations such as raster-ops take several lines of code to set up since each position and size must be specified in the correct manner, whereas a raster-op can be specified in one line of PS-algol code. Images can be created with relative ease in PS-algol whereas the comparable operation in C requires explicit creation of structures and memory allocation rather than a mere declaration.

```
struct char_entry {
        unsigned short wp, hp;   /* width and height of smallest
                                    bounding box of character    */
        short xoffp, yoffp;      /* offsets of start of bounding box
                                    from start of character      */
        int tfmw;                /* device independent width     */
        int pxlw;                /* actual width on MG-1 in pixels
        */
        char *pixels;            /* pointer to start of pixel info  */
}


struct font_entry{
        int k,c,s,d,a,l;
        char n[STRSIZE];         /* parameters of font definition in
                                    dvi file                     */
        int font_space;
        int font_mag;
        char name[STRSIZE];      /* full path name of font file     */
        int magnification;
        int designsize;
        struct char_entry ch[NPXLCHARS];
                                 /* an array of characters       */
        struct font_entry *next; /* a pointer to next font in list  */
}
```

**fig 3.1** The font data structure definition in C

The declaration of the font data structure used in the C version of the preview program to hold all the fonts used in a *dvi* file is given in *fig*. 3.1. The data structure consists of a list of *font_entries* which contain information on one font and an array of *char_entries* for the characters of the font. Each *char_entry* contains its dimensions, layout and pointer to the start of its associated pixel information.

```
structure font.structure(int NbrFonts ; *pntr fonts )
structure a.font( string name ; int fontspace , number ; *pntr
chars )
structure a.character(int xoffp, yoffp, tfmwidth ; #pixel ch )
```

**fig. 3.2** The font data structure declaration in PS-algol.

The PS-algol declaration of the data structure is given in *fig*. 3.2. This data structure is contained in a *font.structure* as a vector, or array, of *a.font* structures, which in turn contain a vector of *a.char* structures. No redundant information has been stored in this data structure unlike the C version and information on the dimensions of characters is obtained by inspection of the image width rather than being stored explicitly, this has the effect of producing a tidier data structure and tidier algorithms acting upon it.

The PS-algol version of the data structure holds the information at a "higher level" than the C version, by omitting information required only in building the data structure and storing the characters in the more useful way, as images, rather than just bytes of pixel information. The redundant information only used in the building process is included in the C version since, in order to omit it, a temporary data structure must be built to hold this information until building of the font structure is finished. The C language would require that the user allocate memory

for this temporary structure, which would reside in memory for the full execution time of the program. The user will see no advantage in separating the temporary information from the essential data since no memory will be saved and the building algorithm will become more complex. The PS-algol data structure contained none of the building data since it could be stored in a temporary data structure easily, and the memory used is reallocated by the heap management system once the building is complete. This leads to a greatly simplified structure producing easier programming and more readable code.

The factor which limited the execution speed of the program was the building of the font structure from the font file, the persistent capabilities of PS-algol, when utilised to store these font structures in a database, called TEX.FONTS, allowing the preview program to get access to the ready built font data structure by executing a lookup using the name of the font as a key. This method gave fast and tidy access to the fonts, since most of the work had been done by the building program, and made up for any speed lost due to the high level language access to data structures.

The C version made better use of the facilities provided by the existing MG-1 architecture in that it gave direct access to the machine capabilities in the way they were meant to be accessed. The PS-algol program ran at a comparable speed to the C version once it was converted to take full advantage of the persistent databases. The PS-algol system is built on top of the Unix™ system as an interpreted abstract machine and therefore has not got the same hardware backup which the MG-1 graphics routines have, but if a machine architecture were designed with the PS-algol persistence and graphics in mind the programs run on it would have a vastly reduced execution time. Therefore a system could be produced which provided the desirable

facilities of a high level language, discussed earlier, as well as the execution speed of the existing systems, such as the MG-1. This would lead to more reliable programs, better programming styles and programs which would be easily ported from one machine to another.

## 3.4 A Comparison Of Implementation Efficiency.

Since the two languages considered were designed with different aims in mind, one to produce efficient code and the other for programming efficiency, the styles, ease of programming and the code produced were interesting to compare.

Since all the research into the TEX system was carried out during the initial C implementation, the production times for each program cannot be fairly compared, but the ease of programming can be looked at. C requires the user's needs to be specified explicitly in that a program must allocate the memory it requires by function calls, manipulate structures through pointers and deal with memory locations directly and indeed may use one location as an integer in one line of code and as a pointer in the next. This leads to confusing programs and some potentially "illegal" statements can compile, which builds unreliability into many programs. PS-algol, on the other hand, deals with a lot of the repetetive work which is left up to the programmer in C, in that memory allocation is handled automatically by the program on declaration and structures, images and other data types are considered as objects rather than just areas of memory. This leads to clearer programs and allows thorough type checking done mainly at compile time.

C allows access to data structures at a low level and does not insist

on initialisation of declared variables. This increases the number of run time errors produced and means that such errors do not have a specific type, the structure involved in the error cannot be deduced which makes debugging of programs particularly laborious. Such errors in PS-algol are not so frequent since the compiler's type checking mechanism catches many of the potential errors and the initialisation at declaration policy reduces the instances of illegal access to memory. Thus the number of run time errors experienced in program development is greatly reduced and correction of any which do occur is made easier. It was found that these differences in the HLL lead to easier and faster implementation of programs and the finished system is hopefully more robust and reliable than its equivalent C implementation.

## 3.5 Conclusions.

The experiences with both the languages considered demonstrated the power of high level language programming. Contrasting PS-algol and C, the PS-algol program was easier to write and the result was more comprehensible. The stricter type checking, mandatory initialisations, high level data structuring and persistence of values made PS-algol a more productive language. C however can be more efficient, although PS-algol proved to be fast enough. Any new programmer introduced to the code could possibly understand it a lot easier and make more reliable changes a lot sooner than could be done if considering a C program.

Currently many graphics programmers must access graphics facilities of a machine through a low level system language such as C, since there is a lack of high level languages which provide such facilities. Bitmap graphics machines are increasing in number due to falling

hardware prices, therefore it is becoming increasingly necessary for languages to give the high level support to graphics programmers which conventional programmers currently enjoy having the desirable effect of producing an increase in the amount of graphics software available, due to ease of production, and allow many more programmers to incorporate a graphics interface in their programs.

# 4. Implementation Experiment.

The initial task in working with the *PS-algol* run time system was to port the compiler and interpreter onto the Whitechapel *MG-1* workstation. This involved implementing the machine dependent graphics routines within the interpreter and deciding how to set up the graphics display. It was hoped this would give an insight into the language's weaknesses and machine dependencies, so changes and extensions to the language graphics routines could be considered in order to provide a better environment for the programmer, and a better implementation which would be easier to port and faster to implement.

## 4.1 Porting of the Compiler.

The existing *PS-algol* system on the *Perq* workstation consists of a compiler written in *S-algol*, which produces PS-algol abstract machine code or PS-code, and an interpreter for the PS-code written in *C*. To simplify the job of producing a working PS-algol compiler on the MG-1 the existing compiler, written in S-algol was changed to PS-algol. This was purely an editing job since the S-algol language is a subset of PS-algol differing only in the syntax of procedure headings and calls. The editing produced a full PS-algol compiler written in PS-algol. This compiler source could then be compiled on the *VAX*, the *PS-code* produced copied over to the *MG-1* and run. Since the compiled code is interpreted it is machine independent, assuming the two machines have the same byte order. Porting the compiler in this way saved the additional task of porting the *S-algol* system and was only possible since the *VAX* and *MG-1* have the same byte order.

The PS-algol interpreter source was then copied to the MG-1 and compiled with the existing VAX options set. The chosen options were to produce an interpreter which would run object files containing integers stored in VAX byte order and would skip operations concerned with the graphics facilities of PS-algol.

The only problem encountered in producing the non-graphical implementation of the interpreter was in the implementation of the *digit* function. This problem was found to be due to a slight difference in the MG-1 C compiler from other C compilers. Once the problem was sorted the interpreter was used to run the compiler already produced on the VAX. The programs compiled using this system produced identical code to that produced by the VAX PS-algol compiler.

## 4.2 Introduction of Graphics Facilities.

The compiler produced for the MG-1 would generate full code for graphics routines included in a source, but the interpreter being used to run the compiler, since it had been taken from the VAX, could not interpret the graphics operations encountered in the object code, but would just skip the operations merely changing the stack pointers appropriately in order that execution of the object code could continue. The next task of porting the PS-algol system to the MG-1 was to introduce operations to the interpreter which would allow the graphics routines to be executed with the expected results and displayed on the bitmap screen of the MG-1. These routines made calls to the existing *MG-1* graphics library routines in order to implement the PS-algol raster operations, line drawing, locator functions, image formation and initialisation as well as some other image manipulation functions. The major change made, as seen by the user, from the *Perq* implementation

was in the way the screen was presented. The *Perq* displayed the graphical output in the window from which the program had been called, whereas the new *MG-1* implementation dynamically created a graphics window. A page showing the screen displayed by running a program is included in Appendix A. The creation of a second, graphics, window kept the standard output and graphical output separate allowing the user to make use of the standard output and preventing this output from corrupting the graphics display. The problem on the Perq was caused by the positioning of the standard output text at a point unknown to the programmer and the fact that it was forced to share the screen with the graphical output. Any text produced by a program would appear on top of any existing graphics previously output, in a position determined by the terminal emulator. If the standard output were to reach the bottom of the screen and then continue, the contents of the screen would be scrolled upwards, behaviour expected from a terminal window but not from a graphics window. This scrolling had the effect of moving any graphical output on the screen upwards with the scrolled text thus corrupting the display.

One restriction of the new implementation was that the user did not have total control over the size of graphics window created. This was due to the fact that the size of the window was determined by the size of the console window, which itself could be varied in size before a program was run, so altering the size of the graphics window produced. This approach did not take full advantage of the machine's capabilities and indeed was very restrictive in some cases. Since many programs required a larger window a method was developed by which the user could specify window position and dimensions at run time. At the start of a run the cursor would change and wait for the user to press a button and drag the mouse diagonally over the required window rectangle.

When the window was created it would be positioned where the user had indicated. The method of window creation used in the *Perq* window manager was copied as closely as possible. This approach seemed to be a lot more popular with users than the original method used.

The new implementation made several differences within the interpreter. The Perq accesses the screen as a device, opening this as a file, so it is distinguished from the internal bitmaps, whereas the *MG-1* represents a window internally as a bitmap which the user must change to display graphics. Originally the method used in the Perq interpreter was mirrored in that the screen was considered as a unique bitmap, storing the *MG-1* raster structure associated with the screen bitmap and using this to gain access to the window. This approach lead to problems in heap allocation and so it was eventually decided to store the screen raster in the same way as any other internal bitmap, which satisfied the heap and garbage collector routines and produced tidier and more efficient code. Changes to this *screen* bitmap had to be detected and the screen updated accordingly since the MG-1 software is designed in such a way as to require changes to the screen bitmap to be "announced" to the window manager before they are displayed. Another problem encountered was found to be due to the garbage collector. On garbage collection the screen image descriptor was moved as are other objects to allow compaction, this had the effect of losing some of the subsequent changes to the screen as the operating system continued to update the screen from the previous area of memory. This was solved by redefining the screen at the end of a garbage collection if its image descriptor had been moved. Similar problems were found with the cursor bitmap, eventually it was decided to store it in the same way as the screen although it had to be redefined completely every time it was changed, a reasonable overhead since the cursor is not changed nearly as

frequently as the screen.

PS-algol provides a function *fill* which fills an enclosed area on an image in a certain colour. The *Perq* implementation had written this in *PS-algol* and extracted the code from the system database at the system setup. This routine called a function *line.end* which was implemented in the interpreter. The *line.end* routine was replace with a *fill* function written in *C*. This had the effect of increasing the speed and producing a usable function.

Several programs were written to compare the speed of execution of the *MG-1* implementation of *PS-algol* to that on the *Perq*. The programs were run, on both implementations, several times and the average taken in order to compare execution times. *Fig.* 4.1 below gives the results of these tests.

| Program \ Machine | Perq | MG-1 | Factor Perq / MG-1 |
|---|---|---|---|
| | Execution time in seconds | | |
| Raster.ops onto screen | 102 | 119 | 0.86 |
| Raster.ops onto bitmap | 349 | 153 | 2.28 |
| lines onto screen | 3 | 16.5 | 0.18 |
| lines onto bitmap | 21.5 | 11.5 | 1.87 |
| 5,000 integer multiplications | 56.5 | 33 | 1.71 |
| Store images in database | 82 | 101 | 0.81 |
| Compile time for db. program. | 87 | 112 | 0.78 |

**fig. 4.1** Results of speed tests.

The programs all repeated one function several hundred times in order that the efficiency of the implementation of that part of the language could be measured. A short description of each program is given below :

(1) An image of dimensions 20 by 20 was *xor*-ed onto the screen 10,000 times. Each raster-op was done to a different part of the screen to allow the program's activity to be observed.

(2) An image of dimensions 20 by 20 was copied onto an internal image, declared in the program rather than the screen as above. This was executed 40,000 times before the image was copied onto the screen.

(3) The function *pnx.line* was used to draw 400 lines onto the screen.

(4) The function *pnx.line* was tested again but the 4,000 lines in this case were drawn onto an internal image.

(5) The speed of the processor was tested by executing 5,000 integer multiplications.

(6) A 30 by 30 image was created and stored 30 times in a database. This program would compare the commit times of the two machines.

(7) The compilation times on each system of program number 6 were measured and compared.

The results showed the *MG-1* to be faster at internal operations such as raster operations from one image to another but slower when operations were to be displayed on the screen. The reason for this is due to the implementation of the screen window on the *MG-1*. Each operation on the screen image has to be carried out as any internal image operation would be then the changed part of the screen image has to be "updated" or changed on the *actual* screen. The internal operations are faster because the *MG-1* library functions can be used on any image structure, positioned anywhere in memory, and so do not require that tests be carried out on the positioning of a bitmap before it is used in a raster operation. This had the effect of simplifying the operations within the interpreter and so producing faster running code. The code of the program used to test the raster operations from memory to screen is given below as an example of the type of program used. The other tests were similar to this so it was felt to be unnecessary to include the source code for each test. See *fig.* 4.2 below.

```
let square = image 20 by 20 of on
let bleep = ""
write "return to start'n"
let wait := read.a.line()
for i = 1 to 100 do
    for j = 1 to 100 do
        begin
        xor square onto limit screen at i, j
        end
write bleep
write "stop the clock'n"
wait := read.a.line()
```

**fig. 4.2** The source of the first test program.

The time difference in the multiplication program must be due to the existence of faster hardware operations for multiplication.

The compiler used on the *MG-1* was implemented differently from the *Perq* compiler, in that the *MG-1* version was written in *PS-algol* as opposed to *S-algol* on the *Perq* and is therefore run on the *PS-code* interpreror rather than the *S-code* interpreter. Since the *MG-1* compiler is seen as just another program by the interpreter which runs it the *PS-algol* standard functions must be read in from the system database and all the other initialisations required before any program is run by the system must be carried out. The *Perq PS-algol* compiler, being written in *S-algol*, does not have such an overhead and can therefore start the compilation immediately thus saving time. This fact accounts for the time difference in the compilation of program number 6 on the

two machines.

## 4.3 Conclusions.

The graphics routines were implemented on the *MG-1* by use of the Whitechapel graphics library routines [STEN85]. These provided access to the machine facilities at several levels of abstraction, allowing the functions on the Perq, which were only the fairly limited library routines, to be implemented. It seems that an interpreter written for the *MG-1* would have been harder to port to the *Perq*. The final implementation gave the same displayed images as the *Perq* implementation but did not use the *MG-1* to its full potential. Examples of this are the locator function and the line drawing function of *PS-algol*, whose functionalities are limited to that of the *Perq*. The locator function in *MG-1 PS-algol* was functionally equivalent to that provided by the *Perq*, which is a subset of the capabilities potentially provided by the *MG-1* and indeed would be looked upon as restrictive by the average *MG-1* programmer in that the facilities provided would not allow programming of some operations done easily by the MG-1 when accessed via C. The line drawing routine of *PS-algol's pnx.line* which takes two points on an image, the image itself and a style consisting of an integer 1..3 for black, white or inverse, and draws a line between the two points. This function is a direct implementation of the Perq function *wline*, which takes the same parameters. The equivalent *MG-1* graphics library routine *GLine* can take any one of *sixteen* raster-ops instead of the *Perq's* three styles. The lack of a fully implemented line drawing system in the language is a regrettable omission since many, if not all, graphics programs require the use of line drawing. The importance of line drawing was demonstrated by the

inclusion in the new Angel graphics software recently produced by Whitechapel [STEN86] of facilities which allow lines to be drawn in a variety of styles. Lines can be dotted where the style of dotting can be specified or alternatively can consist of a repeated pattern rather than just a line of dots. This facility could potentially give the user great freedom in the specification of style of lines required. These are facilities a user would find potentially useful but would be denied access to from *PS-algol* since the *Perq* cannot provide them. A *PS-algol* standard function *Line* attempts to produce these functions but is rarely used since it runs slowly and is complex to use. To make use of the desirable line drawing facilities provided by the *Angel* graphics library, or to provide an equivalent system through *PS-algol*, a line drawing statement should be included in the language which would provide the facilities required by all graphics programmers. This suggestion is expanded in chapter 8.

If all the graphics routines of the language use the same format and conventions then programs would be easier to write and tidier once written. The language must provide the user with suitably powerful tools to allow programming of required functions, otherwise programs will become increasingly complex to achieve their aim.

# 5. Implementation Changes.

The completed version of PS-algol on the MG-1 had several major differences from the original implementation on the Perq, most of which are due to Perq specific attributes of the original implementation. Some of these changes were to eliminate machine dependencies and others were to make improvements in the PS-algol environment. The changes made are discussed below.

## 5.1 Font Changes.

The font files used on the Perq were transfered to the MG-1 and used the existing *build.font* program, written in PS-algol, to place the *font* structures built from them in the FONT database. Each word of bits in the font file had to be mirrored before it was displayed on the MG-1 screen. This is due to the MG-1 having a "small end" screen as opposed to the Perq "big end" screen meaning that the least significant bit of each word is displayed furthest left. This problem was solved by a simple addition to the build.font program which mirrored each byte. This machine dependency was produced by the function used to build the image structure from the Perq font file. To increase the speed of loading the font into the PS-algol *font* structure the function *get.plane* was used, which allows access to one plane of the image as a vector of integers. The words of the font file can then be copied into the image. This gives access to the machine's representation of the image structure leading to machine dependent programs. An alternative approach could be used to build the *font* structure which does not use the function

*get.plane* but would involve reading one byte at a time from the font file and executing up to eight raster operations onto the image depending on the number of ones in the byte. This would hide the different implementation of images but produce an extremely slow running program at present.

## 5.2 Locator Changes.

PS-algol has a *locator* function which returns the position of the mouse within the PS-algol graphics window. A structure is returned by the function giving the screen position of the cursor, a vector of four booleans which indicate the status of the mouse buttons and another boolean stating whether the window is selected or not. It was decided that this function was machine independent in that its functionality is what would be expected to be provided by any machine. The number of buttons on the mouse can vary from one machine to another, usually between one and four, which would result in the need to alter the size of the boolean vector *"buttons"* returned, making one entry for each button. The *"selected"* field is meaningful on the Perq since the graphics window can be selected for keyboard input which has the effect of setting the cursor to the PS-algol cursor and displaying any changes made to it from within the program. It is therefore advantageous, on the Perq, for the programmer to know whether or not the window is selected. The MG-1 implementation, on the other hand, by creating a second window ( see chapter 4 ) for graphical output made the *"selected"* field meaningless since the graphics window cannot be selected for keyboard input and the PS-algol cursor is always displayed while the mouse is over the window. The MG-1 implementation always returns the selected field with a value of false.

## 5.3 Set.Locator Changes.

The existing PS-algol system allows the programmer to change the mode of operation of the locator, the circumstances which prompt a return of the function call, through a function *set.locator* which passes the four parameter values directly to the Perq operating system. This function was particularly difficult to implement on the MG-1 since it is so Perq dependent. The full range of facilities were not implemented but only the ones felt to be required by the programmer. It is obvious that a *set.locator* function of some sort is required, but its user interface should be changed in order to make the function machine independent. It is reasonable to assume that any machine would be capable of providing a locator function with the modes of use required by the programmer.

## 5.3 Cursor Changes.

PS-algol has a standard image, called *cursor*, predefined in every program. Changes to this image, which can be changed in the same way as any other image, are displayed on the mouse cursor associated with the PS-algol window. The Perq implementation specified this to be a 57 by 64 pixel image, a size which is determined by the Perq operating system PNX. The MG-1 operating system specifies a 64 by 64 image which can be accessed through the MG-1 implementation of PS-algol. Programmers using the language should take into account the different cursor sizes provided by various machines. The decision to use the size of cursor provided by the MG-1 was made since the original 57 by 64 size provided by the Perq implementation was not decided upon by the

language designers to be a desirable size but to be the easiest to implement. Although the previous size could have been provided easily on the MG-1 the larger size was preferred.

The fully implemented PS-algol system will possibly provide a database of useful cursor images which would be read and copied onto the cursor image when required. Useful images could possibly include a clock, an area input prompt, a text positioning prompt and various arrows and crosses for a variety of occasions. Each implementation of the PS-algol system may provide differently sized cursor images but since they will be copied directly onto the cursor the size is irrelevant and may thus be made transparent to calling programs. Each new implementation of the system when set up will require a new set of images to be drawn to take full advantatge of the size of the cursor. Programs may be kept machine independent despite the fact that the cursor bitmap is of unpredictable size.

## 5.4 Conclusions.

The remaining graphics functions of PS-algol were found in general to be reasonably machine independent although it was evident that they had been originaly designed with the Perq facilities in mind. An example of this is the range of *raster operations* provided by the language which correspond directly to the operations defined in the Perq. These operations are not as complete as those provided by most machines, in particular the functions 'clear to black' and 'clear to white' were omitted but these operations can be achieved by use of existing raster-ops although the code required for these operations is not immediately understandable. *fig.* 5.1 shows the code for a clear screen operation. A new programmer introduced to the system may not realise

immediately this is the conventionally accepted method of clearing an image.

**xor** screen **onto** screen

**fig. 5.1** A clear screen operation.

Since the graphics facilities of PS-algol were originally designed and implemented on the Perq many of the facilities provided are direct copies of the facilities provided by this machine. Porting the language to the MG-1 demonstrated this to a certain extent. Many small differences, such as the value returned by *locator* when the cursor is outside the window, could be found by a programmer whose programs depend on such implementation eccentricities. These differences must be removed by including their definition in the language description. Designing the graphics on the Perq also limited the designers to functions specific to a tablet based system as opposed to a mouse based system. Functions to control the speed of movement of the mouse on the screen and to position the cursor over a desired location could be useful in a mouse based system but have not been included in the language since they are meaningless when a tablet is being used.

Most of the functions which demonstrate machine dependence are currently classed as *private* functions indicating that they are not for general use but only for the system implementer. Some of these functions are required by the average programmer so must be given an improved user interface in order that they may be reclassified as *public* functions. Suggested changes are given for some of these functions which would make the improvements required but in general they must be changed in such a way as to eliminate the direct link with the operating system, such as the parameters of the *pnx.line* routine

discussed in chapter 8. It is commonly the case that programmers gain access to these functions in order to use their functionality or power. This has the effect of producing machine dependent programs.

# 6. Language Definition Changes.

Any PS-algol language definition changes considered were to have the effect of making the language more machine independent, increasing the usability of the language from the programmer's point of view, making the job of programming easier and helping to produce tidier programs.

Changes to the language where decided upon after consideration of the existing graphics languages described in chapter 2.

## 6.1 Text Output.

When thinking about improvements to the language it was decided that text output could be improved. The language facilities currently existing allow the user to output text in two ways :-

(1) Strings, integers, reals and booleans can be written to standard output using the *write* statement.

(2) Strings can be written onto an image using the *string.to.tile* function. This is done in a specified font, taken from the database, and the image returned from the function is then "pasted" onto the screen anywhere the user chooses by use of one of the raster operations.

### 6.1.1 The *write* statement.

The *write* statement has the disadvantage of producing non-graphical output in that the text it produces is displayed as text by

the terminal emulator on the graphics screen in a position unknown to the programmer, making the statement difficult to use in a tidy manner. It was felt this was inconsistent with the rest of the language in that all output should be explicitly to the bitmap screen. The *write* statement allows the programmer to set out the output in an organised way by stating field widths for integers and strings. This makes it extremely convenient to use in that it allows the use of newlines, line overflow, layout characters and the positioning for successive *write*s needs no explicit effort on the part of the programmer. Unfortunately the statement is spoiled by the lack of control of absolute positioning of the output text, a lack of facility to write to stored ( not displayed ) structures, and a lack of choice of character font, style and size. Perhaps its worst drawback is the interference between write statement output and bitmap output causing the bitmap output to scroll, in some cases, on the Perq; this was avoided on the MG-1 implementation by sending the standard output to a separate window. Since the write statement has these drawbacks it is often used ( in the context of graphics programs ) only as a debugging aid, to print out intermediate values of strings and integers.

### 6.1.2 The *string.to.tile* function.

The second method of text output presented to the user is the *string.to.tile* function. This is passed two strings, one containing the string to be printed, the other containing the name of a font which specifies the style and point size in which the string is to be presented. The function looks up the font in the database and returns an image of the required dimensions containing a bitmap representation of the string passed to it. The user can then combine this image with any other by use of the raster-op functions. In practice this function is slow since the

FONTS database has to be opened each time it is called and any characters used taken out. It also complicates the program since the user must position the returned image on the destination image, a process which usually requires tedious programming computation, except in the case where the programmer is anotating a diagram where such precise user control over position is necessary. The additional computation work should be avoided since a task as simple as text output should be straight forward and automated. Most of the text positioning should be done automatically, to keep the programmer's job as simple as possible, allowing concentration on the real application of the program. Further limitations of this approach include the method used to output numerical values. Each numerical value must be converted to a string before being passed to the *string.to.tile* function. The user is also left to work out field widths which in some cases could unnecessarily complicate a program. *string.to.tile* allows the user to place multi-font text on the bitmap screen in a controled manner but does not present the simplicity of the write statement.

### 6.1.3 The new *write* statement.

An attempt was made to design one mechanism, meeting the requirements currently met by both the above language features, having the best properties of both of them and avoiding the above difficulties. A new **write** statement was proposed and implemented which would take the arguments of the existing statement but send its output to an image rather than standard output. The old style of write statement could still be accessed by use of the *output* statement, allowing existing programs to be easily converted to run in the new system. Additional arguments are specified within the new statement :

(1) the name of the image to be operated upon.

(2) the position on the image where the output should start.

(3) a font to be used.

(4) a raster combination rule, raster-op, used in placing the characters.

The syntax for the new statement incorporates the above information by use of the syntax specified in *fig.* 6.1.

<write> ::= **write** <write.list>
{ **onto** <#pixel-clause> } { **at** <int-clause>, <int-clause>}
{ **in** <pntr-clause> } { **using** <raster.op> }

**fig. 6.1** Syntax of the new *write* statement.

Any of the reserved words **onto, at, in** or **using** and their associated expressions can be left out. This has the effect of assuming default values as follows. In the case of no image being specified the output is sent to the *screen* image, if no position is given the output starts at the "current position" in the specified image, the default font is taken to be the "current font", discussed later, and the default raster-op is **copy.**

The image current position is a co-ordinate pair stored as part of the image descriptor within the abstract machine. Every image and limit has an associated position which indicates the starting position of the next write onto that image. This position is initialised to the top left hand corner when the image is created allowing a whole "page" of text to be output before an error occurs. The current position is only a guide to the position of the next character to be output; the actual position

depends on the size of the font used. The **write** statement attempts to organise the output on the image by adjusting the co-ordinates of the starting position of each character positioned on the image. The actions taken to position each character are listed in *fig.* 6.2.

(1) If the character is to be placed in such a way as to put part or all of it off the top of the image the **y** co-ordinate is reduced to place the character on the "top line" of the image so that no part of it is lost.

(2) If the character is to be placed off, or partly off, the bottom of the image an error occurs - This will eventually raise an exception to allow the user to move the current position to the top of the image or to scroll the existing text up by executing a scroll procedure in the exception handler.

(3) If the character is to be placed off the right hand side of the image both the **x** and **y** co-ordinates are reduced to give the effect of the character being placed on the "next line down" in the image. The reduction in the **y** co-ordinate depends on the font size.

(4) If the character is to be placed off the left hand side of the image the **x** co-ordinate is increased in order to place the character at the start of the line.

**fig. 6.2.** Considerations on positioning a character in an image.

The above alterations of the current position of each character has the effect of outputting text onto an image in the same format as standard output would do to the screen. This has the effect of reducing the programmer's work by removing the need to work out the position for each integer or string and calculate the space it will occupy before a

write to the image.

Two functions are provided to the user to enable the current position in an image to be manipulated. The user would require to examine the current position and change it to a specific value, the operations are allowed by the following functions declared in *fig.* 6.3.

**let** get.current.pos = **proc**( **#pixels** the.image **-> pntr** )
- the pointer returned gives a *point* structure, one of the PS-algol predefined structures, containing the co-ordinates of the current position of *the.image.*

**let** set.current.pos = **proc**( **#pixels** the.image; **int** x, y )
- changes the current position of *the.image* to ( x, y ).

**fig. 6.3** The standard functions used to manipulate the image current position.

If the *write* statement is used, without a font being specified by an **in** clause, the "current" font is used. The current font is initialised by the execution of the poms code ( chapter 1 ) before, the user code, at the start of each instance of the interpreter by extraction of the *fix13* font form the FONTS database. This font is used as a default until it is changed by the function provided for the purpose, declared in *fig.* 6.4.

**let** set.font = **proc**( **pntr** new.font -> **pntr** )

**fig. 6.4.** The procedure to change the default *write* statement font.

The default font is changed by passing a font structure taken from

the database to the *set.font* procedure which in turn returns the previous default font. This is to enable the user to switch back to the previous default easily. An example of the use of this procedure has been included, in Appendix A, as an MG-1 screen dump, in order to demonstrate its simplicity.

Using this proposed **write** statement would allow the programmer to organise output with great ease. A text output "window" can be defined on the screen by defining a limit on an area of the screen and passing its name to the *write* statement. This area can be used only for text output giving the impression of a window dedicated to this purpose. Eventually the text in this window will be able to be scrolled, as described above, allowing almost automatic text output.

The above statement was implemented on the MG-1 and experimented with and discussed by several users. The actions taken in an image edge violation (fig. 6.2) were viewed by many to be restrictive and produced inconsistent results in its treatment of characters at various sides of images. A more flexible, user controllable, system had to be produced before the write statement could be accepted as part of the language. The suggested alternative ties up the actions taken on an edge violation in a variable PS-algol function. Since a variable function is used, the user may produce an alternative edge violation strategy to implement facilities such as scrolling text, which would be handled by a "bottom edge violation" handler or allow text to be listed on the screen a page at a time, in which case the handler for a bottom edge violation would be able to prompt the user before erasing one page of text and displaying the next. In both cases the right edge violation handler could possibly be set up to provide double line spacing for text.

The function handling the edge violation strategy is described below.

```
let edgeViolation := proc( string character; pntr  edges, font;
                           #pixel theImage; int x, y  -> bool )
```

*edgeViolation* is considered to be called by the *write* statement when a character is going to be partly off one of the edges of *theImage* if it is placed in the proposed position. The character causing the violation and the name of the edge are passed to the procedure as strings. The destination image and the font being used are passed to the function mainly to provide information on their sizes. $x$ and $y$ describe the "proposed" co-ordinates of the character. A call to the function occurs when an edge is violated, therefore the procedure must include code to "correct" the position of the character for each type of violation i.e. correct the position for violation of each edge. The procedure sets the image current position to a point which will not cause an edge violation and in addition may also carry out some other operations if the programmer wishes.

Correcting one edge violation within the procedure may cause another edge to be violated, a problem encountered if the character causing the violation is wider or taller than the destination image. To help solve this problem the *edgeViolation* procedure returns a boolean, the value of which determines the action taken in such a situation. If the value is true the interpreter will not attempt to recall *edgeViolation* continually but will clip the offending character. Clearly the situation could arise of an infinite loop caused by repeated calls to the procedure, each in turn causing another violation.

A user wishing to change the treatment of edge violations will write a procedure of the same type as that described in the declaration above. If the new procedure was called *double.spaced.scroll* the

statement below would insure that the required treatment was given to text output.

edgeViolation := double.spaced.scroll

A PS-algol system could include a database of commonly used procedures which could be easily assigned as above to provide the required functionality without the user writing a new function allowing the facility to be exploited to the whole by the casual user and not just the experienced programmer willing to investigate the writing of an edge violation procedure.

Since the edgeViolation view of text output presents a fully flexible interface to the programmer while still being straight forward to use it has been implemented in order that it may be experimented with. This will be introduced to the standard PS-algol.

## 6.2 Text Input.

Once text output had been changed, it was felt necessary to improve the text input facilities provided by the language in order to bring them into line with text output. A function was required which read in text from the keyboard and produced an echo on the bitmap screen. A new standard function was written which does the required operation. The declaration of this function is given in *fig*.6.5 along with the *read.a.line* function which it could replace.

*input.line* reads a string from the keyboard, which it returns to the program. The string typed in is echoed on the screen bitmap at the current position in the current font. Any number of deletions can be handled as for standard input. A demonstraton of this function is given

in Appendix A.

```
let input.line  = proc(->string)
let read.a.line = proc(->string)
```

**fig. 6.5** The new and old text input operations.

## 6.2 Functions Added to the Language.

After looking at the language ***Amber*** [CARL84] it was decided to add a function *texture* to the language, the declaration of which is given in *fig.* 6.5.

```
let texture = proc( #pixel the.pattern , the.image )
```

**fig 6.5.** The declaration of the texture function.

This function takes two images, the first containing a pattern the second being the destination of the pattern. The function replicates the pattern image throughout the provided destination image. This produces the effect of shading the image by replicating the pattern throughout the image in a tiled format. *texture* may be used by the programmer to shade a header or boundary of the screen. Currently PS-algol forces the user to achieve this effect by doing multiple raster-ops from the pattern image to the destination. This is a worth-while addition to the language since so many programs currently achieve its effect in inefficient ways. Many application programs carry out this kind of operation at start up therefore it is desirable that it may be done quickly in order to allow the

application to begin as soon as possible. The texture function provides the required facilities in a tidy easy to use way and executes quickly giving the fast setup required by programs. Several patterns could be stored in a database for use by programs using this function. A demonstration of this function is given in Appendix A.

The changes made to the language are discussed further in chapter 7 where the experiences of programming in the new system are reviewed.

# 7. Evaluation Of Changes Made.

Since the implementation of the **print** statement within the PS-algol system several users have experimented with the extended version of the language and have been able to comment on the extension and compare the facilities provided with the previously available system of text output.

Before the **print** statement PS-algol had two methods of text output, one being terminal output, the other produced text on the bitmap screen ( chpater 6 ). The *write* statement sent text to the standard output thus allowing no text positioning and restricted the output to the default keyboard font. The *string.to.tile* function allowed multi-font output but was slow and required that a user calculated and kept track of the screen positions of strings and required that the returned image be "pasted" onto the screen explicitly. The functionality of these operations made it necessary for the users to work hard to get the most out of the bitmap screen.

## 7.1 Improved Speed.

The **print** statement allowed fast text output since it was implemented by the interpreter rather than by being written in the language itself. The improvement in speed was demonstrated by the successful implementation of a game highly dependant on execution speed and therefore required fast input/output. This consideration will become less and less relevant as machine power and processor speed

increases but is currently important, since new facilities are partly judged on their performance.

## 7.2 Versatility of print.

Certain features of the **print** statement were exposed in the statement's use, techniques which were not envisaged before the implementation of the system.

Previously when a programmer was developing a program, in order to pinpoint faults and follow the flow of control *write* statements would be inserted to show the intermediate values of variables. The output from these inserted statements would scroll up the screen in a disorganised manner. The **print** statement allowed improved organisation of such output. If a value were to be output each time round a loop, instead of each value being separated by many other lines of such output each value could be printed at the same point on the screen thus replacing the previous value and allowing the current state of the program to be read from the screen easily. This use of the statement allows "organised" debugging and verification of programs.

The raster combination rule which may be specified in the *using* clause of the statement permitted easy programming of several useful effects. Text can be output in inverse video easily by use of the **not** raster-op.

**print** "John Livingstone." **at** 10, 10 **using not**

The above statement would produce the string with a black backgraound and white letters, a format which is ideal for highlighting page headings etc.

Text produced by the **print** statement can be underlined easily. The following code demonstrates this.

```
print "John Livingstone." at 30, 5
print "_____" at 30, 5 using ror
```

An **ror** raster-op, PS-algol's "or" combination rule, is used in the second statement to prevent the underscore characters overwriting the text of the string.

Use of the **xor** raster-op in a print statement ensures that the text will be visible against its background. White text will be produced if the background is black and black text produced otherwise. **xor** would be used when it is known that parts of a string will be overwriting both possible backgrounds.

## 7.3 Readability.

It was found that code produced using the **print** statement was a lot more readable than text output routines written under the old system. This is due to the fact that a user may code an operation in one line which would have previously taken many lines of code. The required operations are explicitly and succinctly stated while the functionality required is easily accessed.

## 7.4 Conclusions.

On the whole the response to the new implementation was favourable and it was viewed to be a significant improvement over the

previously available systems of text output. The flexibility of the *edgeViolation* procedure was considered to cover most envisaged uses of the statement.

The only criticism of the extension aired was that it increased the size of the language. One of the principles of PS-algol is that it must be kept as small as possible to keep its complexity to a minimum. The advantages gained from the statement would seem to outweigh this disadvantage.

Overall the experiment seems to have been a success.

# 8. Suggested Further Changes to the Language.

The graphics facilities provided by a language should be consistent syntactically with the other language constructs, and manipulate the graphical data structures in a similar manner used to manipulate other types of data structure. This consistency allows the graphics facilities to be considered as more than a library of routines added to the language as an after thought, and leads to easy programming and easier understanding of the language as a whole.

Any further changes to *PS-algol* suggested in this chapter are thought to comply with the above requirements.

## 8.1 Line drawing.

Line drawing is an operation frequently used by all graphics programmers, therefore must be supported fully by any graphics language.

Currently line drawing in *PS-algol* is handled by a function *Line* which may be used to execute any operation, provided as a function, a specified number of times between two points on an image. An image, two end points and the changes made to the current position before each execution of the function are passed to *Line* as parameters. This function could be used to program all line drawing requirements of the programmer but presents an unnecessary degree of complexity in attempting to cover all possible requirements. The programmer may use this function to replicate a certain image at frequent intervals between two points on the image.

A private function *pnx.line*, described already in chapter 1, draws a straight line one pixel wide between two specified points on an image. This function implements the simplest line drawing required by programmers.

PS-algol has a line drawing package, described in chapter 1, which draws lines in real space, allowing manipulations such as rotations, and then allows them to be mapped onto integer space and displayed on an image. Such a line drawing package is required in situations where transformations must be carried out on the lines before they are displayed. The underlying data structure and transformation operations are accessed easily when required which saves the user rewriting such routines each time they are required. When drawing diagrams, underlining text or performing other simple line drawing tasks there would be no need to consider lines in real space before they are displayed in integer space, therefore in these cases a simpler approach to line drawing would be taken.

The operations provided by the two line drawing functions would be easier to program and follow if they could be accessed via a language construct rather than by a function call. The syntax of the construct provided would be required to conform with the existing operations of the language such as the raster-ops and the new *print* statement, discussed in chapters 1 and 6 respectively. Replacement of the line drawing functions by a language construct would have the desirable effect of making line drawing simple to use and improve readability of programs.

*PS-algol* requires a construct which presents the simplicity of *pnx.line* but the power of *Line*. The syntax of the line drawing routine proposed is given in *fig*. 8.1.

**drawline** {**on** <#pixel-clause>}

{**from** <int-clause>,<int-clause>} **to** <int-clause>,<int-clause>

{ **by** <int-clause>, <int-clause>} { **with** <#pixel-clause> }

{ **using** <raster.op> }

**fig. 8.1** Proposed syntax for line drawing statement.

The drawline statement contains six clauses, five of which are optional. Most calls of the statement will be able to omit one or more or the optional clauses assuming the defaults. The destination image, specified in the **on** clause, defaults to the *screen* as in the *print* statement. **by** allows a step size to be specified, giving the number of pixels between each operation on the destination image, it defaults to 1. The **with** clause, if included, provides a "brush image" which will be replicated at each point along the line, rather than an image containing one dot, which would be the default. The **using** clause provides the user with the same facilities as those provided by the equivalent clause in the new *print* statement, described in chapter 6. It allows the user to specify the combination rule used when adding new pixels to those already on the image. No **using** clause implies a *copy* raster-op. The end points of the line are specified by the **from** and **to** clauses each taking a co-ordinate pair. **from** is optional since this may be taken from the current image position introduced in chapter 6. After an execution of a **drawline** statement the current position of the image will contain the co-ordinates of the end point of the line drawn allowing another line to start at this positions without the program explicitly stating the point. Since the *print* statement also uses the current position it is suggested that users of the system create a *limit* (chapter 1) on the whole of any images being written and drawn on in order to keep the current write and current drawing positions separate.

```
let draw.screen = limit screen at 0, 0
let print.screen = limit screen at 0, 0
drawline on draw.screen to 100, 200
print "John Livingstone" onto print.screen
```

**fig. 8.2.** Keeping the draw and print current positions apart.

All lines drawn onto the screen would be drawn onto *draw.screen* and all text sent to print.screen, thus keeping the two current positions separate.

All the line drawing operations a programmer would require could be implemented easily by use of this operation. A procedure implementing dotted line drawing is included in *fig.* 8.3 to demonstrate the statements use.

```
let dotted.line = proc ( int x1, y1,  x2, y2;
                         int space.x, space.y;
                         int dash.x, dash.y )
    ! draws a dotted line between the points (x1, y1) and (x2, y2)
    ! in the first quadrant on the screen. The dashes will be of a
    ! size enclosed by a dash.x by dash.y rectangle and the spaces
    ! at least the size of a space.x by space.y rectangle.
begin
let pen.image = image  dash.x by dash.y  of off
    ! the image which will hold one dash.

    ! now draw the dash on the image.
```

```
drawline on pen.image from 0, 0 to x2 - x1, y2 - y1
    ! the part of the line not on the image is clipped.

    ! now draw the dotted line onto the screen.
drawline from x1, y1  to x2, y2
    by dash.x + space.x, dash.y + space.y
    with pen.image

end ! dotted.line
```

**fig. 8.3.** A dotted line drawing procedure using drawline.

A line of text displayed on the screen by the *print* statement may be underlined in the manner proposed in *fig* 8.4 below.

```
let text.start = get.current.pos( screen )
print "An example line of text" onto screen
let text.end = get.current.pos( screen )
drawline from text.start( x.pos ), text.start( y.pos )
          to    text.end( x.pos), text.end( y.pos )
```

**fig. 8.4.** Underlining text.

## 8.2 Locator operations.

The locator operations of PS-algol, discussed earlier, provide all the functionality the programmer requires, but the combination of the functions *locator* and *set.locator* is complex to use, *set.locator* presents extremely low level access to the machine's graphical input facilities and

the meaning of its parameters is far from intuitively obvious.

A combination of the two functions which it is hoped would be easier to use and a lot more powerful than the existing facilities is proposed below.

Two PS-algol functions would be accessible to the user, one would provide the same functionality as the current *locator* function, the other would present a lower level view of the system. The lower level function, *primitive.locator,* would be used to implement various *locator* functions each presenting different modes of operation.

*primitive.locator* when called would return a pointer to a list of "mouse records" which had been stored by the system since the last call of the function. The system would store every record generated by mouse movements, mouse clicks, keystrokes, window selection and other such operations rather than being selective in which were stored. Each "mouse record" in the list would contain information such as the current screen position of the mouse when the record was created, the status of the screen and the cause of the event, that caused the record to be created. The higher level *locator* function would start with a call to the *primitive.locator* function. It would then process the list of records returned selecting which ones were required and storing any required in the future. A *locator* function to return every point where button 1 was pressed would look for mouse click records and examine the pressed status of each button and only return those which satisfied the two conditions.

A *locator* function implemented under this system required to return all positions passed over by the mouse, in order that processing could be done at each one by the calling program, would carry out the following operations :

(1) A call to *primitive.locator* would return a list of new mouse records, this list would be appended onto the global list of unused, or unprocessed, records.

(2) *Locator* would now traverse the list of unprocessed records looking for the first relevant record by examining the cause of each record until one was found which was caused by a mouse movement. Any records found to be irrelevant would be removed from the list.

(3) The first record found to be relevant could be passed back to the calling program in the usual mouse structure.

The current locator system could be implemented by this method easily. Currently a call to *locator* returns with the first record created after the call, therefore in many cases a call to the locator function waits for the user to move the mouse or click a button. To implement this system a function which made several calls to *primitive.locator* would be written. Any records returned from the first call would be discarded, since they could be considered as old records, and the function again called continually until a non empty list was returned. The information in the first entry of the returned list would be passed back by the function to the calling program and the rest of the list discarded.

It is hoped that such a locator system would allow the user to specify the functionality required a lot easier than by the current system. The PS-algol system implementor would be able to provide a database of *locator* functions thought to be the most frequently required.

## 8.3 Colour operations.

Currently PS-algol provides the facility of multi-planed images which may either be considered as an image consisting of multiple monochrome images or, on a coloured device, could notionally represent coloured images. If the multi-planed images were to be considered as coloured images, the colours displayed on the device can be manipulated by the functions *colour.map* and *colour.of* which act on the device's colour table.

Colour display devices, on the whole, consider an image to be a two dimensional rather than three dimensional rectangle of pixels with each pixel consisting of more than one bit of information, therefore all the information associated with one pixel is held in the one place rather than spread over several planes. To allow colour to be implemented in PS-algol this second type of image may have to be introduced. The "type" of an image could be specified at declaration.

When working with coloured images the existing raster-op rules become meaningless as they do not produce any predictable results. Additional operations would have to be introduced to the language which would combine coloured images in a predictable and useful manner. Colours have to be mixed, taking into consideration transparent colours, and intuitively expected colours must be produced. Suggested operations are made in [PORT84] and [STEN86].

# 9. Conclusions.

## 9.1 Introduction.

Bitmap graphics machines are becoming more readily available than the previously popular calligraphic line drawing systems, a trend allowed by the drop in price and increase in availability of the large amounts of memory required to hold information being stored on the bitmap display. The transition to bitmap machines is associated with an increase in power and facility. Complex pictures may be displayed without the limiting factor, inherent in calligraphic displays, of screen flicker experienced when an attempt is made to display too many lines; consequently areas may now be filled rather than shaded by multiple lines. Pictures may now be manipulated internally in a displayable form, rather than as a list of co-ordinates and vectors, allowing increased display versatility and production of "true to life" pictures. The old system provided by calligraphic machines of model manipulation may still be advantageous in some situations and can still be implemented on bitmap machines to a certain extent.

High level languages capable of exploiting the full power of calligraphic displays have been developed after much experiment. Some of the languages produced during this work are discussed in chapter 2, they range from Euler-G a very basic, almost low level, language to Mira which allows manipulation of graphical objects through high level constructs. This same process of experimentation is now under way for the bitmap display, some languages and work contributing to this process are also reviewed in chapter 2. Smalltalk-80 provides perhaps

the best attempt so far but by no means provides a complete solution to the problems of bitmap graphics. It seems that since bitmap workstations can potentially provide a very much larger range of facilities than the calligraphic machines, the languages intended to access these facilities must be a lot more complex than the previously designed calligraphic languages. It is the job of the language designer to provide easy access to the full facilities available.

## 9.2 Identification of the problem.

Usually a machine designer and operating system team will provide a system programming language and a library of functions accessible through it for the control and display of graphical data and input related to that data ( see chapter 2 for the description of such a system language extension for the MG-1 ). Graphical programming by calls to these libraries from the system language is the method used by most programmers as it is all that is available or the only method of obtaining all required effects. System languages, by their nature, do not provide full support for type checking and high level data structures in contrast with high level languages. They are often implemented differently, and provide access to different library procedures on different ranges of machines. Thus programs developed in a system language, although they may work, take longer to develop and are prone to hidden bugs as well as being difficult to transfer between different types of machine. This thesis therefore identifies the need for a HLL providing the abstractions which facilitate the programming of effects achievable on bitmap displays. These abstractions must fulfil the three objectives listed below.

(1) allow commonly required effects to be programmed simply;

(2) give access with resonable efficiency to all the effects achievable on a bitmap display;

(3) be defined without reference to, or influence of, specific machines.

These requirements were derived in chapter 2 and proposed implementations fulfilling them are presented in chapters 6 and 8. The final requirement is studied in chapter 4 where the machine independence of PS-algol is investigated.

The first specific problem that is identified and addressed by this thesis is therefore to search for and design such abstractions. This problem does not permit a single solution, since the requirements conflict and the particular solutions given in the thesis are at best appropriate to current technology and envisaged applications. Therefore the final part of these conclusions, after the specific search has been described, contain a suggestion of design principles which it is hoped would prove helpful to others who perform the search in a different context.

## 9.3 The high level language solution.

An approach to the stated problem is the development of graphics in a general purpose high level language which gives the freedom of expression and full functionality the programmer requires thus reducing the dependence on system languages. A HLL is designed to produce efficient programming, by checking for type consistency while still allowing freedom of expression, and not to reflect the properties of

a particular machine, and ideally it will be implemented with precisely the same definition on many machines. The machine independent definition of a HLL is intended to allow all programs written in that particular language to run with identical results on any machine capable of supporting the language. The abstractions, formal definitions and constructs supported by HLLs, since they are free from machine specific attributes, are intended to lead to improved styles of programming and speed of software production, with more reliable programs produced that will run on a variety of machines. The precise definition of all language attributes removes any ambiguity such as those which sometimes occur in low level languages resulting in unpredictable and inconsistent performance of certain language statements from machine to machine. Also the programmer's skills are less machine dependent, experience gained on one machine may be useful on a variety of machines. The challenge of producing such HLL abstractions for graphical input / output on bitmap displays is particularly difficult since the properties of individual devices and supporting hardware are visibly different from machine to machine. It is a challenge recently taken up by a few HLL designers. It is an easily justified theme for this work, as the benefits of success would be considerable. A high proportion of programming is concerned with organising communication with the user (estimated at 60%), therefore the facilities provided for programming this interface must be carefully considered as an important integral part of a bitmap language. As bitmap displays are becoming increasingly common, the user's expectations of the quality of this interface are rising, programming languages must improve in order to enable easy production of the users' expectations and requirements and to allow efficient access to the full facilities of such machines.

The limiting factor of existing high level graphics languages is that they provide only a small subset of the functions potentially accessed at the lower system language level. Simply adding large libraries of functions, a solution offered by most system language extensions and graphics packages, would result in programs written with the small subset of the existing functions so far explored or understood by the programmer. High level languages must develop powerful, easily understood graphics facilities which will encourage the user to exploit the full potential of bitmap graphics interaction rather than a mere subset. The operations required may be determined by examination of the functionality of existing graphics libraries as many of these have been thoroughly used and have been shown to allow access to the required operations.

High level languages must allow the production of succinct and understandable, and therefore more easily maintained code by introducing levels of abstraction into possible uses of the language constructs, since they may be complex in some cases. Unnecessary details can be omitted, by the use of defaults, an approach not possible with library routines since they are accessed via function calls whose parameters must all be included in each call whether or not they are required. Language capabilities can then be used in a simple way by a naive user. Their full power may be exploited by a more experienced (or ambitious) programmer for the production of elaborate algorithms or sophisticated non-standard effects.

## 9.4 High level language graphics requirements.

The primitive facilities of a graphical language must be designed in such a way as to either directly provide a required visual effect or to allow the easy definition of higher level functions which provide the remaining effects. A description of the functionality required in terms of visual effects is given below. These effects may be classed as either input or output facilities.

### 9.4.1 Basic output facilites.

The graphical output facilities required by a user could all, in principle, be provided via the two primitives which allow the value of a single pixel to be read and written. Such a primitive approach is rejected since all the remaining primitives and language facilities would have to be provided in software, producing intolerably slow systems. The machine hardware must be allowed to support the language requirements at a higher level than the single pixel level, a compromise must be made in the level of hardware and software support for languages in order that a desirable implementation efficiency is provided.

Three basic output facilities must be provided by languages whether by primitives or by higher level operations built from existing primitives. These three facilities are described below.

**1. Text output.** Text must be output to the screen and internal bitmaps in the variety of fonts and styles available on a bitmap machine in such a way as to allow the user as much control in the organisation and

format as is required. Text output is discussed in chapter 6 where the old primitives of PS-algol are compared with the improved facilities.

**2. Line drawing.** The line drawing facilities of a language must allow, in addition to simple line drawing, the production of textured lines of various styles consisting of repeated patterns, and dotted lines. Any lines drawn must be easily styled by the programmer in order that the desired effects may be obtained in a straight forward manner. Raster combination rules may also be specified if their inclusion is relevant. The functionality of line drawing mechanisms are described in chapter 8.

**3. Area shading.** Bitmap displays allow easy shading of areas by repetition of small "pattern" bitmaps over a large area, having the effect of producing a halftoned colours ( in the case of monochrome bitmaps ). Areas of an existing bitmap may be copied onto another bitmap, the source being combined with the destination in a variety of ways. The PS-algol implementaton of this is provided jointly by the raster-op facilities ( chapter 1 ) and the *texture* function ( chapter 5 ). Smalltalk provides an abstraction away from the raster-op which allows the inclusion of mask bitmaps giving easy access to the operations described.

### 9.4.2 Basic Input Facilities.

Graphical input is intended to allow the user of programs to indicate positions on the screen and provide strings and characters on request from the keyboard. Text input is classified as graphical input since the language may be able to control any echo of text on the bitmap screen. Thus graphical input may be split into two classes :

**1. Keybord input.** Keyboard input could, in principle, be provided by a single function which would read a single character from the keyboard buffer and return this to the calling program. Probably a function to inspect the state of the keyboard buffer could also be useful.

For bitmap machines it is desirable to provide text input at a higher level. Strings or characters may be read from the keyboard without any echo being displayed on the screen, or if the programmer so chooses the string should appear on the screen as it is typed at a chosen position in the required font and style before it is returned to the calling program. The style of input used would depend on the requirements and functionality of the particular program.

**2. Locator input.** A HLL must provide access to the full functions potentially required by a programmer or provided by each locator device such as a mouse or tablet in such a way as to hide the type of device being considered. It is essential that the full potential of locator input be accessed by the langauge since providing only a subset of possible facilities could restrict the power of programs significantly. The programmer must be able to control the cause of return of the locator function, whether it be mouse movements, clicks or key strokes, and unread mouse input must be able to be flushed or considered in its entirety. A program which follows the mouse cursor around the screen and outputs information at each position it crosses may not be capable of executing the desired operations on each point fast enough to allow the next call of the locator function to pick up the next point crossed by the cursor, resulting in possibly important data being lost. The solution to this problem is to consider the locator input as a queue of records which may, on request, be returned sequentially by the locator function rather than flushed from the queue. A similar program which is only required

to consider points at which a mouse button is pressed would be much simplified if the locator function could be made to return only when a button was pressed, therefore all input from the locator function seen by the program would be relevant to the application.

This full functionality is required to be provided by a language. Locator input is discussed in chapters 4 and 8.

### 9.4.3 Access to environment.

One major attraction of system programming languages for the purpose of graphics is to allow the programmer access to the full window management system capabilities of the workstation. Windows may be created and manipulated on the screen to display the contents of any bitmaps as they change. Windows may also be moved, reduced in size or priority by the system user or the controlling program to display the currently most important bitmap.

High level graphics languages must provide machine independent access to such facilities possibly by making use of a standard window manager or by accessing a language specific implementation. Since the production of the standard window manager Andrew [MORJ86] the former alternative has become a possibility in that any machine capable of supporting the language would be able to run the window manager also, allowing programs which made use of the window manager to be machine independent.

## 9.5 Summing Up.

This thesis reviews several graphics languages and studies implementation of language facilities in PS-algol. Implementation of some of the above proposed essential attributes of graphics languages were experimented with by attempting implementations in PS-algol and were found to perform satisfactorily in that the changes made to the language improved its machine independence, consistency and usability.

The machine independence of PS-algol was examined by the porting of the language from the ICL Perq workstation to the Whitechapel MG-1. This presented several problems which led to further discussion of the language facilities such as the standard output and locator input primitives.

After the completion of the new text output routines the proposed new *write* statement was renamed to *print* to allow the existing programs, which used standard output, to continue to compile and run without change. The alternative edge violation approach to text output (described in chapter 6) was implemented and considered to provide a fuller range of options to the user and therefore seen as desirable. The new system ported back to the Perq, requiring no changes to the user interface, indicating that the extension was machine independent, unlike the old *write* statement which used standard output inconsistently on different machines producing machine dependent programs.

The final completed *print* statement has been added to PS-algol as a standard facility since it was seen by other users to be beneficial and consistent with the rest of the language.

The extensions to the language discussed in chapter 8 have not as yet been implemented, but it is felt they have been justified on paper and fit in with the existing language constructs and philosophy and provide the functionality required from any graphics language.

All the work done with PS-algol and the consideration of other existing languages is felt to have allowed a full description of the requirements of a graphics language and gone part of the way to demonstrate the viability of these propositions. A clear idea of the work which will be carried out in this field in the future has been formed and it is felt that this thesis will point the way for any person working further in this field.

# Bibliography.


[ATKM85] Atkinson, M.P. & Morrison, R., *Procedures as persistent data objects,* ACM TOPLAS 7, 4, 539 - 559, Oct. 1985.

[BROA86] Brown, A.L. & Dearle, A. *Implementation Issues In Persistent Graphics,* Universities of St. Andrews and Glasgow, Persistent Programming Research Report 23.

[CARL84] Cardelli, L., *Amber,* AT&T Bell Labs., New Jersey.

[COLA82] Cole, A.J. & Morrison, R. *An introduction to programming with S-algol,* Cambridge University Press, 1982.

[DENE75] Denert, E., Ernst, G. & Wetzel, H. *Graphex68 Graphical Language Features in Algol68,* Technical University of Berlin, Comput. & Graphics, Vol 1, 195-202, 1975.

[FATR82] Fateman, R.J., *High-Level Language Implications of the Proposed IEEE Floating-Point Standard,* University of California, ACM Transactions on Programming Languages and Systems, Vol 4, 239-257, 1982.

[GOLA83] Goldberg, A. *Smalltalk-80, the language and its implementation,* Addison Wesley, 1983.

[GUIL82] Guibas, L. & Stilfi, J., *A Language for Bitmap Manipulation,* ACM Transactions on Graphics, Vol. 1 No. 3, 1982.

[HOPF86] Hopgood, F.R.A. et al., *Introduction to the Graphical Kernal System,* Academic Press, 1986.

[ICLP84] International Computers Limited, *ICL Perq User Guide,* 1984.

[ISO82] ISO, *Graphical Kernel System Functional Description,* Draft International Standard, 1983.

[KNUD84] Knuth, D.E., *The TeXbook,* Addison-Wesley, 1984.

[MAGN81] Magnenat-Thalman, N. & Thalman, D., *A Graphical Pascal Extension Based on Graphical Types,* Software Practice and Experience, Vol 11, 53-60, 1981.

[MAGN83] Magnenat-Thalman, N. & Thalman, D., *A Three Dimensional Graphical Extension of Pascal,* Software Practice and Experience, 797-808, 1983.

[MORJ86] Morris, J.H. et al. *Andrew: A Distributed Personal Computing Environment,* The Information Technology Center, Communications of the ACM., Vol. 29, No. 3, 184-201, March 1986.

[NAMN78] Nam, N. & Marsland, T.A., *Introducing Graphics Capabilities to Several High Level Languages,* University of Hong Kong, Software Practice and Experience, Vol 8 629-639, 1978.

[NEWW70] Newman, W.M., Gouraud, H. & Oestreicher, D.R., *A programmers guide to PDP-10 Euler,* Appendix VI, University of Utah, June 1970.

[NEWW73] Newman, W.M., *An informal graphics system based on the LOGO language,* AFIPS Conference Proceedings, Vol 42, 1973.

[NEWW79] Newman, W.M., Sproull, R.F., *Principles of Interactive Computer Graphics,* Second edition, McGraw-Hill, 1979.

[NEWW85] Newman, W., Stephens, N., Sweetman, D., *A Window Manager With A Modular User Interface.,* Whitechapel Computer Works, London, July 1985.

[PORTT84] Porter, T., Duff, T., *Compositing Digital Images,* SIGGRAPH 84, July 1984.

[PPRR11] Universities of St. Andrews and Glasgow, *PS-algol abstract machine,* Persistent Programming Research Report 11.

[PPRR12] Universities of St. Andrews and Glasgow, *PS-algol reference manual,* Persistent Programming Research Report 12.

[STEN85] Stephens, N., *MG-1 Genix 2.0A Manual,* Whitechapel Computer Works, London, 1985.

[STEN86] Stephens, N., *New Whitechapel Angel Graphics Library Manual,* Whitechapel Computer Works, London, 1986.

[STED81] Stevenson, D., *A proposed Standard for Binary Floating Point Arithmetic,* Computing, IEEE, March 1981.

[WCWM85] Whitechapel Computer Works, *MG-1 Owner Operator Manual,* 1985.

[YIPC84] Yip, C.K., *The Pascal Graphics System,* Software Practice and Experience, Vol 14, 101-118, Feb. 1984.

# Appendix A.

Some examples of the MG-1 implementation of PS-algol are demonstrated on the next few pages by screen dumps produced on the machine. These demonstrate the following situations :

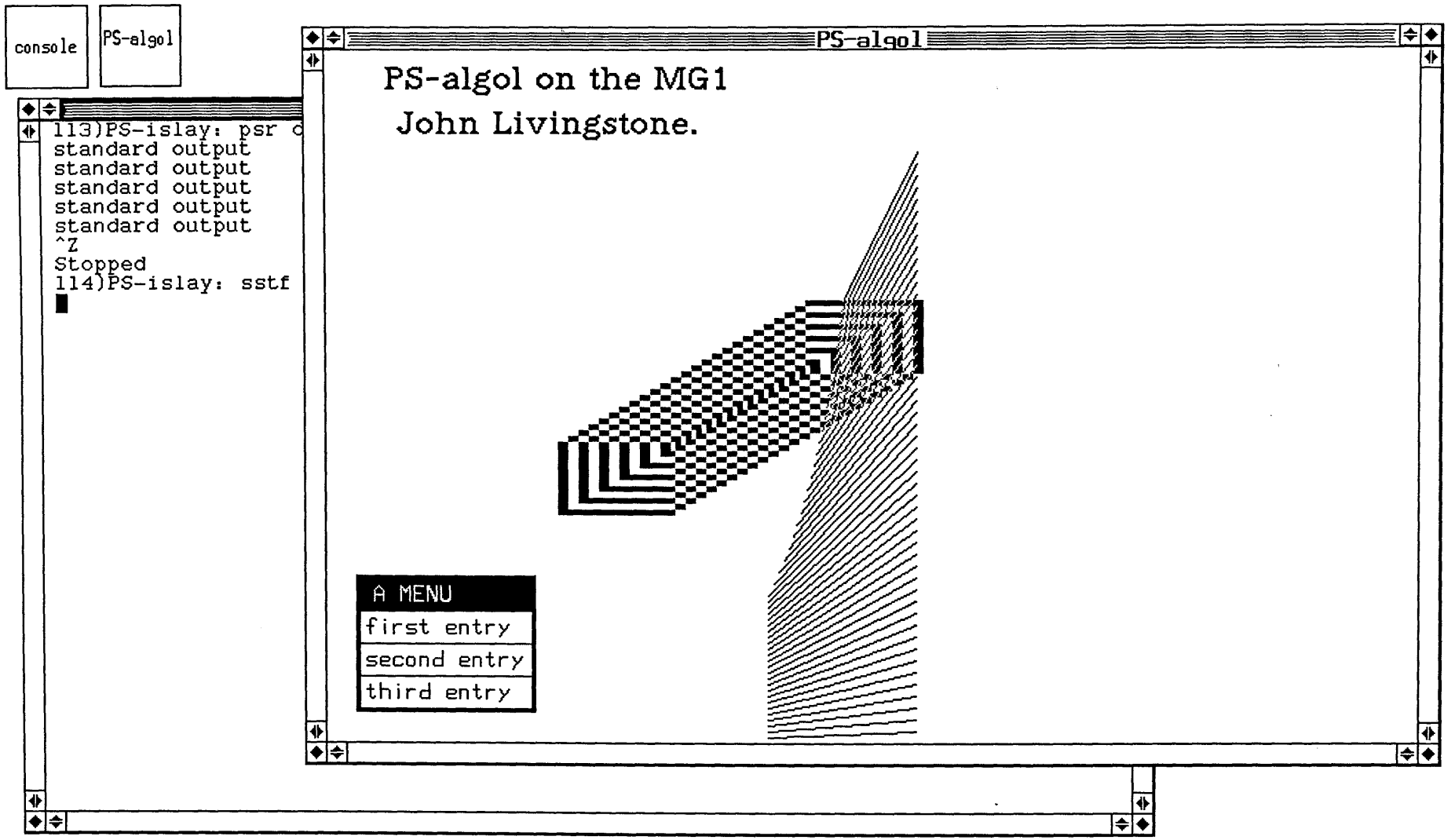
(1) The output from a demonstration program showing the graphics window created by the PS-algol system on the MG-1 and showing the treatment of standard and graphical output from the language.

(2) The PS-algol code which produced the output in the above example.

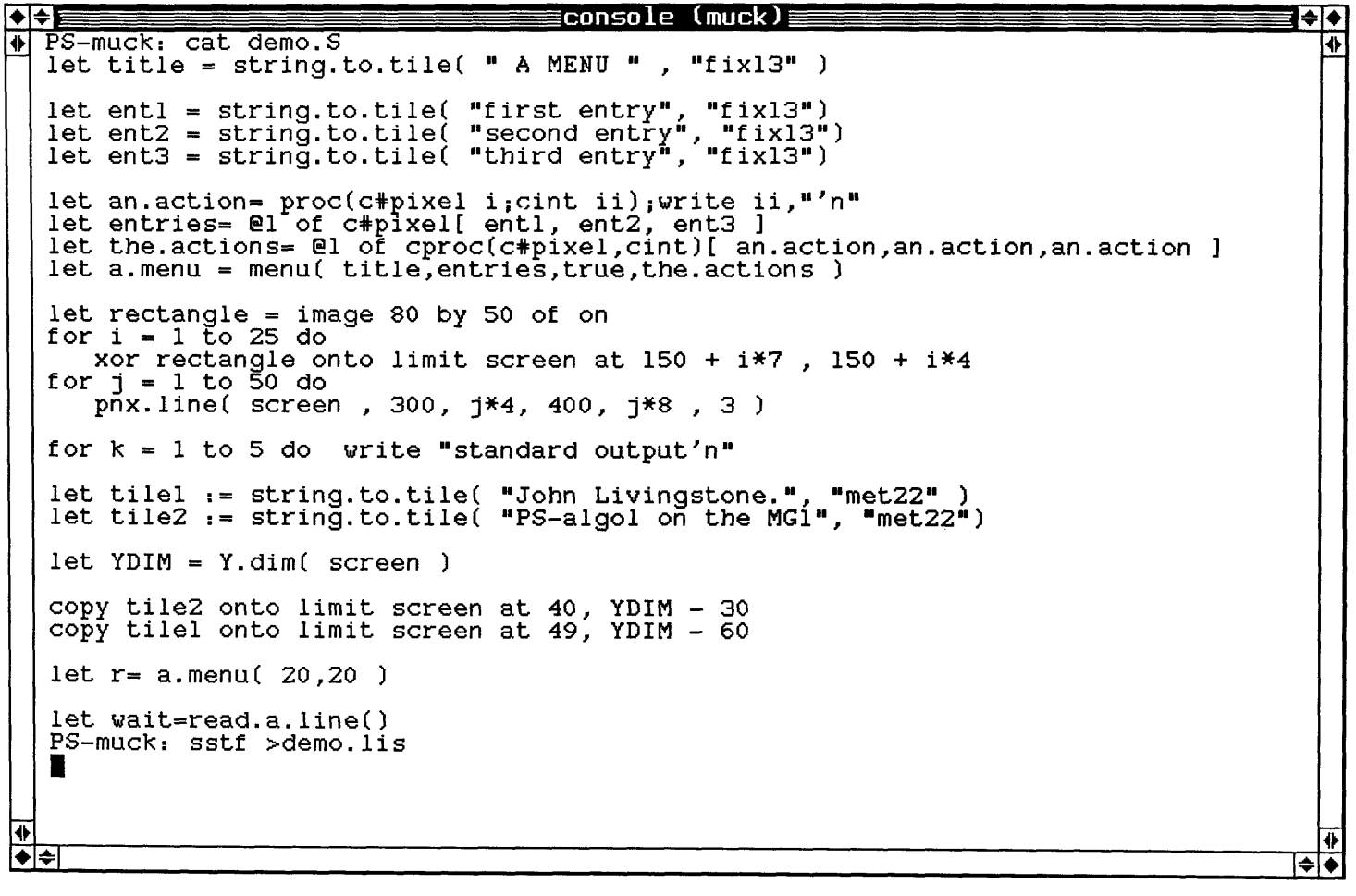
(3) A demostration of the new *write* statement and the *set.font* function used to change the default font used by *write*.

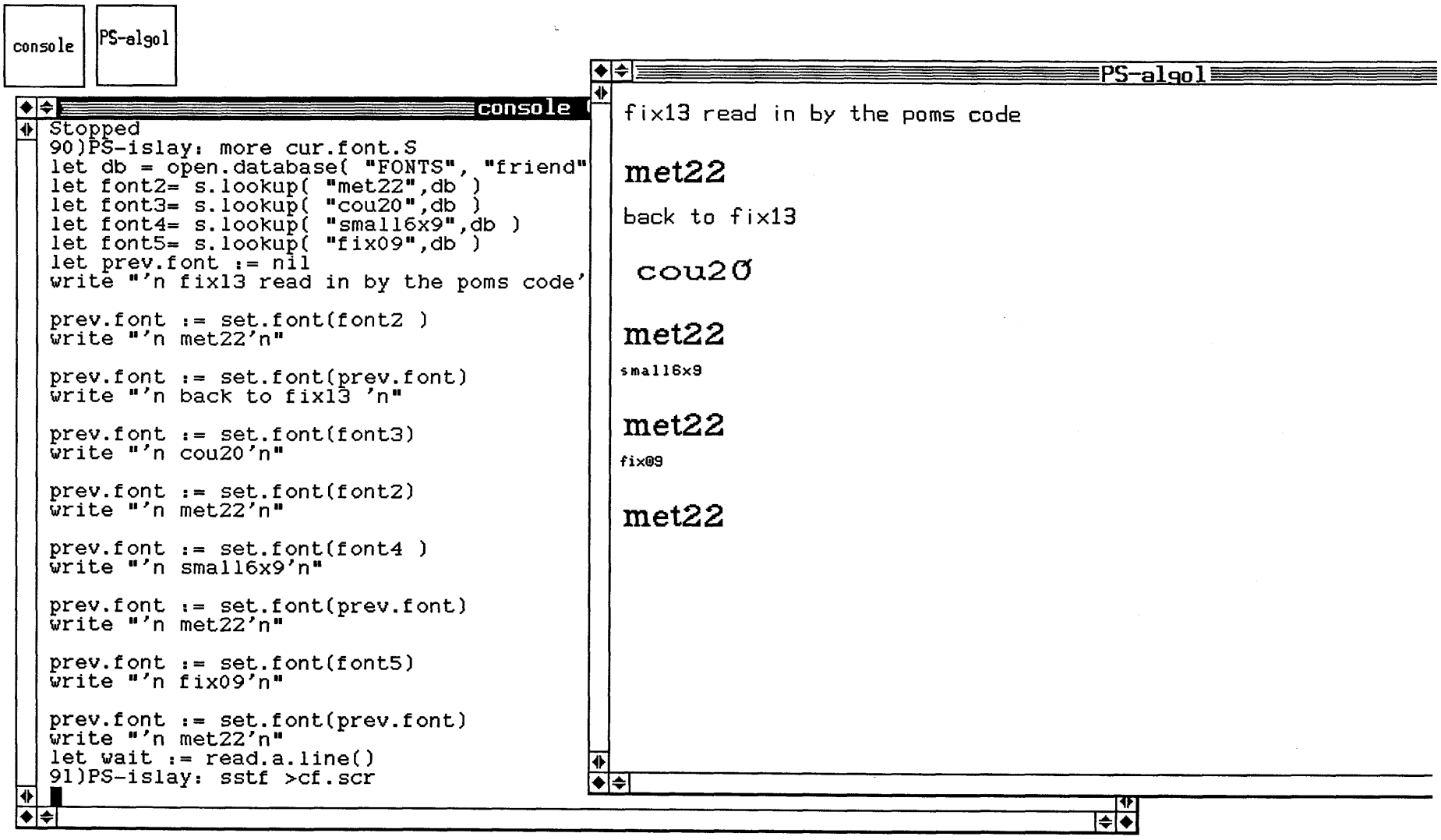
(4) A demostration of the use and the behaviour of the *input.line* function which reads a string from the keyboard displaying the characters on the screen at the current position in the default font.

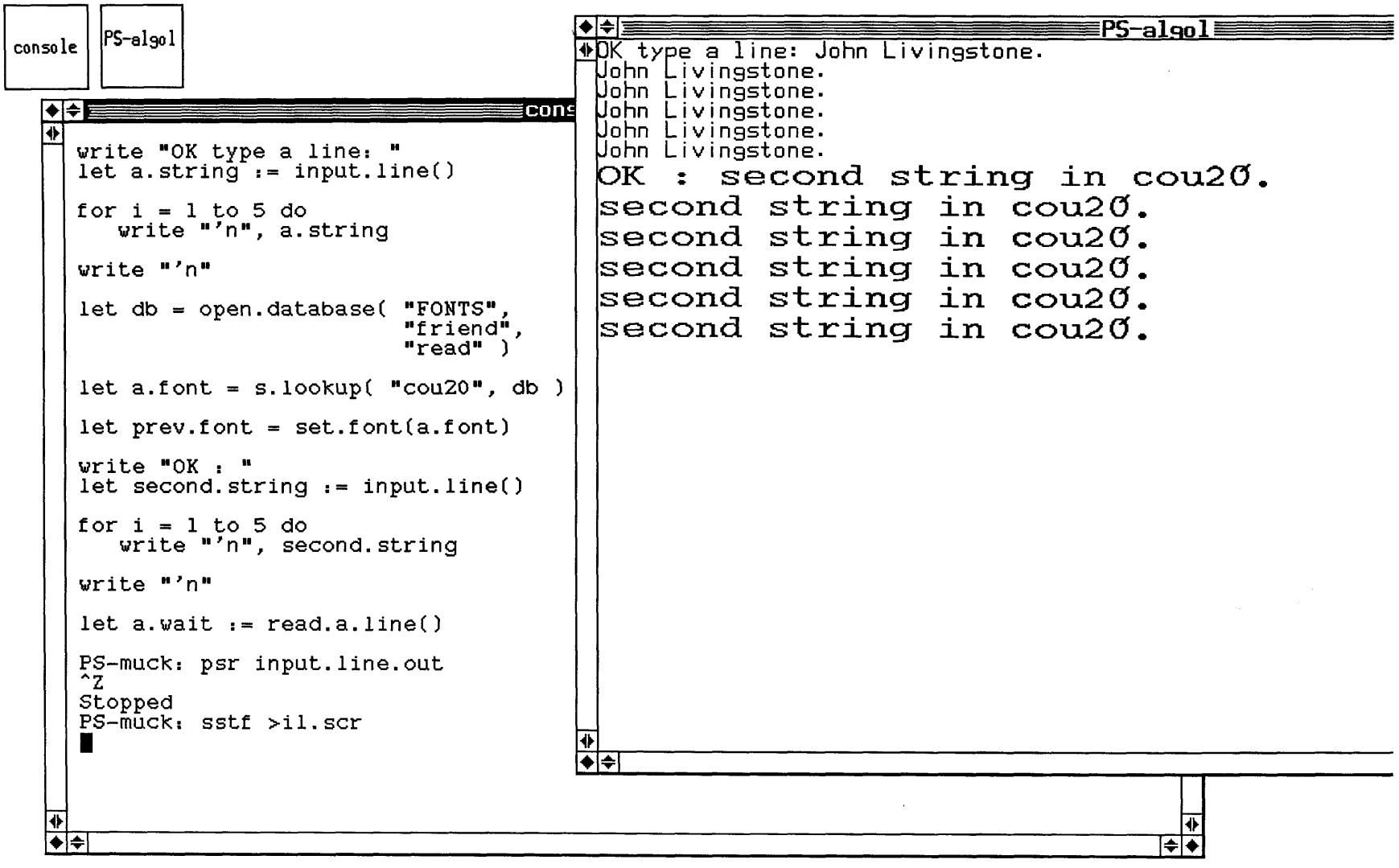
(5) Various uses of the *texture* function added to the language to allow shading of parts of images.


console
PS-algol
PS-algol
PS-algol on the MG1
John Livingstone.
113)PS-islay: psr c
standard output
standard output
standard output
standard output
standard output
^Z
Stopped
114)PS-islay: sstf
A MENU
first entry
second entry
third entry


console


console (muck)
PS-muck: cat demo.S
let title = string.to.tile( " A MENU " , "fix13" )
let ent1 = string.to.tile( "first entry", "fix13")
let ent2 = string.to.tile( "second entry", "fix13")
let ent3 = string.to.tile( "third entry", "fix13")
let an.action= proc(c#pixel i;cint ii);write ii,"'n"
let entries= @1 of c#pixel[ ent1, ent2, ent3 ]
let the.actions= @1 of cproc(c#pixel,cint)[ an.action,an.action,an.action ]
let a.menu = menu( title,entries,true,the.actions )
let rectangle = image 80 by 50 of on
for i = 1 to 25 do
xor rectangle onto limit screen at 150 + i*7 , 150 + i*4
for j = 1 to 50 do
pnx.line( screen , 300, j*4, 400, j*8 , 3 )
for k = 1 to 5 do write "standard output'n"
let tile1 := string.to.tile( "John Livingstone.", "met22" )
let tile2 := string.to.tile( "PS-algol on the MG1", "met22")
let YDIM = Y.dim( screen )
copy tile2 onto limit screen at 40, YDIM - 30
copy tile1 onto limit screen at 49, YDIM - 60
let r= a.menu( 20,20 )
let wait=read.a.line()
PS-muck: sstf >demo.lis


console
PS-algol
console
Stopped
90)PS-islay: more cur.font.S
let db = open.database( "FONTS", "friend"
let font2= s.lookup( "met22",db )
let font3= s.lookup( "cou20",db )
let font4= s.lookup( "small6x9",db )
let font5= s.lookup( "fix09",db )
let prev.font := nil
write "'n fix13 read in by the poms code'
prev.font := set.font(font2 )
write "'n met22'n"
prev.font := set.font(prev.font)
write "'n back to fix13 'n"
prev.font := set.font(font3)
write "'n cou20'n"
prev.font := set.font(font2)
write "'n met22'n"
prev.font := set.font(font4 )
write "'n small6x9'n"
prev.font := set.font(prev.font)
write "'n met22'n"
prev.font := set.font(font5)
write "'n fix09'n"
prev.font := set.font(prev.font)
write "'n met22'n"
let wait := read.a.line()
91)PS-islay: sstf >cf.scr
PS-algol
fix13 read in by the poms code
met22
back to fix13
cou20
met22
small6x9
met22
fix09
met22

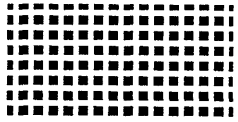

console
PS-algol
write "OK type a line: "
let a.string := input.line()
for i = 1 to 5 do
write "'n", a.string
write "'n"
let db = open.database( "FONTS",
"friend",
"read" )
let a.font = s.lookup( "cou20", db )
let prev.font = set.font(a.font)
write "OK : "
let second.string := input.line()
for i = 1 to 5 do
write "'n", second.string
write "'n"
let a.wait := read.a.line()
PS-muck: psr input.line.out
^Z
Stopped
PS-muck: sstf >il.scr
PS-algol
OK type a line: John Livingstone.
John Livingstone.
John Livingstone.
John Livingstone.
John Livingstone.
John Livingstone.
OK : second string in cou20.
second string in cou20.
second string in cou20.
second string in cou20.
second string in cou20.
second string in cou20.

console | PS-algol

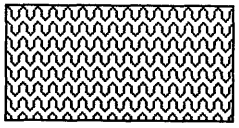
console (islay)

```
let patl := image 8 by 8 of off
let dot = image 1 by 1 of on
copy dot onto limit patl to 1 by 1 at  0 , 2
copy dot onto limit patl to 1 by 1 at  0 , 3
copy dot onto limit patl to 1 by 1 at  0 , 4
copy dot onto limit patl to 1 by 1 at  0 , 5
copy dot onto limit patl to 1 by 1 at  1 , 6
copy dot onto limit patl to 1 by 1 at  2 , 7
copy dot onto limit patl to 1 by 1 at  3 , 6
copy dot onto limit patl to 1 by 1 at  4 , 5
copy dot onto limit patl to 1 by 1 at  4 , 4
copy dot onto limit patl to 1 by 1 at  4 , 3
copy dot onto limit patl to 1 by 1 at  4 , 2
copy dot onto limit patl to 1 by 1 at  5 , 1
copy dot onto limit patl to 1 by 1 at  6 , 0
copy dot onto limit patl to 1 by 1 at  7 , 1

let internal.image = image 120 by 60 of off
texture( patl , internal.image )
copy internal.image onto limit screen at 10, 10

not limit screen to 122 by 62 at 9, 79 onto
      limit screen to 122 by 62 at 9, 79
let a.limit = limit screen to 120 by 60 at 10 , 80
texture( patl , a.limit )

let pat2 = image 8 by 8 of off
not limit pat2 to 5 by 5 at 0, 3 onto
      limit pat2 to 5 by 5 at 0, 3
texture( pat2 , limit screen to 120 by 60 at 10 , 150

write "textured internal image" at 140, 40
write "a screen limit" at 140, 110
write "a second pattern" at 140, 180
72)PS-islay: sstf > t.scr
```

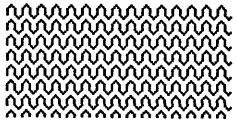
PS-algol



a second pattern



a screen limit



textured internal image