This paper should be referenced as:

Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. “An Integrated Graphics Programming Environment”. Computer Graphics Forum 5, 2 (1986) pp 147-157.

An Integrated Graphics Programming Environment

Ronald Morrison, Alfred L. Brown,
Alan Dearle and Malcolm P. Atkinson*

Department of Computational Science, University of St Andrews,
North Haugh, St Andrews KY16 9SX, Scotland

* Department of Computing Science, University of Glasgow
Lilybank Gdns, Glasgow G12 8QQ, Scotland

Abstract

The facilities of the PS-algol programming language are described in this paper to show how they may be used to provide an integrated graphics programming environment. The persistent store mechanism and the secure transaction facilities of the language provide the basic environment in which an integrated system may be implemented. This is augmented by data types and operations to support line drawings and raster graphics. The combination of these mechanisms may be used to provide the integrated graphics programming environment.

Keywords persistent store, line drawings, raster graphics, picture storage

1 Introduction

The inclusion of graphics facilities in a programming language that supports an integrated persistent environment yields an ideal base for building large and complex graphics systems [1]. The integrated persistent store allows data to be stored and retrieved automatically from user named databases in a type secure manner. When one of the supported data objects in the language is a picture then these may be stored in the databases along with any of the other legal data objects such as integers, vectors or procedures.

Picture libraries of complete or partial pictures may be built in the persistent store in the same manner as any other collection of data or program libraries [3]. Thus the users have a well defined and well understood mechanism in which to access and store pictures. In turn this provides a well structured method of building complex picture systems out of parts of pictures in the same manner that might be used when constructing complex programs out of procedures in a library.

Fourth generation seamless systems such as the software for the Apple Macintosh with picture editors, menu systems and mixed picture, text and program documents may be built out of the integrated persistent environment. We expect to use the persistent store as an enabling technology for a graphics software tools exchange.

This paper describes the facilities available in the language PS-algol[2] for graphics, both for line drawing and for bitmap operations, in the persistent store. The system which is implemented on the ICL PERQ computer[9], supports the notion of transactions and thus transactions on pictures are allowed. Examples of how the various mechanisms might be used are included in the paper.

2 Persistent Store

In an increasing number of applications, it is necessary to manipulate data that must be able to outlive any program that may use it. The usual approach to this problem is to provide a file system or a database management system (DBMS). With such an approach, data is viewed as either short term data, to be manipulated by a programming language, or as long term data, to be manipulated by the file system or DBMS. As a result a large part of a programmer's effort is taken up controlling the mapping between the programming language and DBMS. Furthermore the structure and protection of data provided by the programming language is often lost across the mapping.

In contrast to this approach the programming language PS-algol provides a long term store where the techniques for manipulating data are independent of its lifetime, or its persistence. Hence PS-algol relieves the programmer of mapping the data between short and long term storage and also allows any data object, regardless of type, to have any persistence. The object of this paper is to demonstrate the utility of this approach and in particular how it has been used in a graphics system.

3 PS-algol

PS-algol was developed from the programming language S-algol[11] as an experiment in integrating a long term or persistent store with a programming language. The base types are integer, real, boolean, string, pixel and picture. An image is a two dimensional grid of pixels. These types are augmented by the recursive application of the following three rules. Firstly given any data type T, *T is the data type of a vector whose elements are of type T. Secondly the data type pointer comprises a structure with any number of fields, and any data type in each field. Finally given a series of data types T_i,..,T_n and a data type T, proc(T_1 .., T_n -> T) is the type of a procedure of n parameters with types T_1, .., T_n that returns a result of type T.

PS-algol's persistent store consists entirely of legal PS-algol data objects. The store is partitioned into databases to allow concurrency control and protection when sharing persistent data. Each database has a root data structure which, via pointers, allows access to the other data objects in the database. Therefore the interface to the persistent store need only provide a method of accessing the root data structure of a database. Since the pointer data type may point to any structure class and any data type can be a field of a structure there is no restriction on the data types that can be held in a database.

This ability to store any data type in the persistent store has several useful properties. For example it is possible to store picture descriptions thus allowing the easy implementation of graphics systems such as picture editors. The picture drawing facilities shown later also allow screen images to be kept in a database and manipulated.

Another very powerful property is derived from the fact that procedures are a first class data type. That is, they are higher order and may be stored and manipulated like other objects. Therefore using procedures that can return procedures as results it is possible to implement abstract data objects [3,10]. It is also possible to simulate modules or Ada packages[8] using this technique together with the persistent store. This is achieved by writing programs for each module that when run place their public procedures in a database. Other programs can then use these procedures by retrieving them from the database.

There are several benefits of this approach to providing modules. Firstly there is no need to provide an explicit method of separate compilation and type checked incremental linking is automatically provided. Secondly the delayed binding means that internal changes to a module or the inclusion of additional public procedures need only result in recompilation of that one module and not of every program that uses it.

4 The Persistent Store Interface

The interface to the PS-algol persistent store is implemented by two procedures. These are

let open.database = **proc**(**string** database.name,password,mode -> **pntr**)

This procedure attempts to open the database with the name *database.name* in the mode ("read" or "write") given by *mode*. Passwords are associated with each database to provide some security when sharing databases. The result of this procedure is a pointer to the root data structure of the database or if unsuccessful a pointer to an *error.record*. An *error.record* is a data structure containing information relating to why the open failed.

This is sufficient to provide access to any object in the persistent store. Automatic transfer of data from the long term persistent store is performed by the persistent object management system [4] when the data is accessed. The access of the data in the persistent store is performed in exactly the same manner as in the main store, the object manager knowing the difference so as to leave the transfer transparent to the user.

It is often desirable to ensure that updates to persistent data occur in total or not at all. For example in a banking system the transfer of funds between two accounts would need to be such an update. A mechanism that implements atomic transactions is therefore provided by the following procedure.

let commit = **proc**(-> **pntr**)

When the first database is opened a transaction is started. Ordinarily data objects are copied from the persistent store when they are first used and changes to them are made locally. If any of these data objects have been changed a commit will copy them back to their databases. Any newly created objects reachable from these changed objects will also be copied into the persistent store. They have space allocated for them in the database of an object pointing to them. If data objects from databases that were not opened in write mode have been changed a commit will fail. This ensures that the persistent store is always in a consistent state.

If for any reason a commit should fail then its effects will be removed before any other use is made of the databases it was updating and as in the case of open an error report returned. In this way PS-algol provides a secure transaction mechanism on its persistent store. Further reading on the PS-algol system can be found in [1] and [4].

5 Graphics Facilities of PS-algol

In order to make use of the persistent store facility for graphics, pictures must be a proper data type in the language. Indeed PS-algol supports two separate types for graphics and a method of relating them. These two types correspond to the two main types of graphics devices available, i.e. vector and raster. Line drawings are supported by the data type picture and bitmaps by the data type image.

As well as making the persistent store available, having the graphics facilities supported by data types has another main advantage. We can define infix operators on pictures and images and have syntactic support for the manipulation of these objects. This means that programs which use these objects will be considerably shorter, with all the attendant benefits, than programs constructed from subroutine libraries, for example.

5.1 Pictures in PS-algol

The picture drawing facilities in PS-algol are a particular implementation of the Outline system [12] which allows line drawing in an effectively infinite two dimensional real space. Outline is itself derived from GPL/1[15]. Altering the relationship between different parts of a picture is performed by mathematical transformations which means pictures are usually constructed from a number of sub-pictures. In the Outline system picture description and picture drawing are separated. Picture description is supported by the programming language and picture drawing by mapping the picture to an image. In this manner pictures are described in a device independent manner.

In PS-algol the picture descriptions are represented by the data type picture. The simplest picture is a point. For example,

let point = [0.1,2.0]

represents the point with x-coordinate 0.1 and y-coordinate 2.0 in two-space. All the operations on pictures provided return a picture as their result, so arbitrarily complex pictures may be described and operated on.

Points in pictures are implicitly ordered. The binary operators on pictures operate between the last point of the first picture and the first point of the second picture. In the resulting picture the first point is the first point of the first picture and the last point is the last point of the second picture.

There are two binary operators on pictures, join '^' and combine '&'. The effect of the join operator is to give a picture that is made up of its two operands with a line from the last point of the first operand to the first point of the second operand. Combine operates in a similar way without adding the joining line. For example,

let box = [1,1] ^ [2,1] ^ [2,2] ^ [1,2] ^ [1,1]

will produce a unit square with its bottom left hand corner positioned at the point [1,1]. This is illustrated in Figure 1.

(note that the axes are put in for illustration and are not part of the picture)

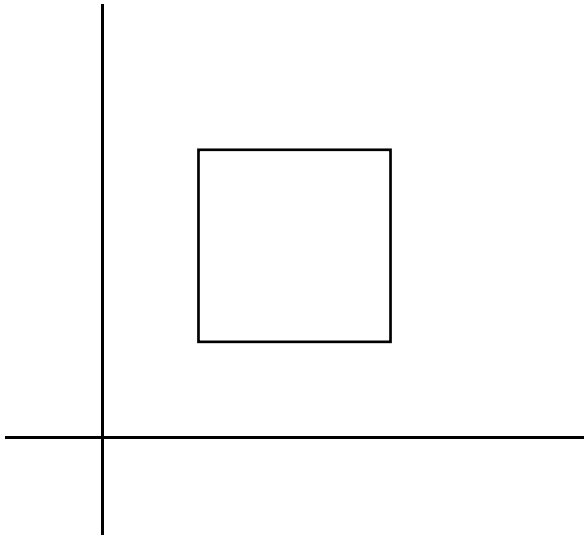


Figure 1: The box

In addition to the binary operators pictures may also be transformed by shifting, rotating and scaling.

shift p **by** x.shift,y.shift

will produce a new picture by adding x.shift to every x-coordinate and y.shift to every y-coordinate in the picture p. For example,

let new = **shift** box **by** -1.5,-1.5

Will initialise the value of the identifier new to be the picture shown in Figure 2.

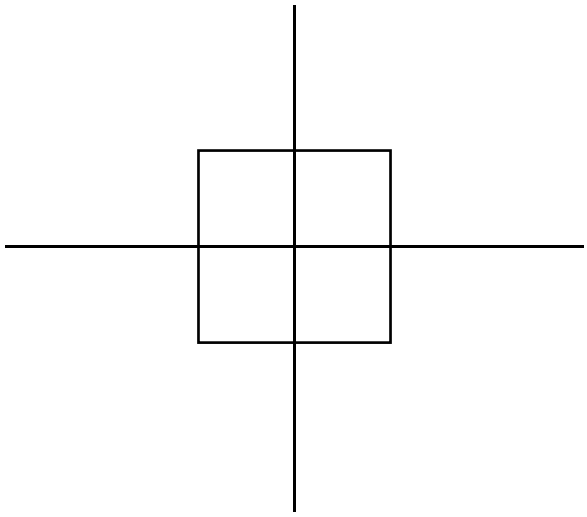


Figure 2: The shifted box

Rotation can be achieved by

rotate p **by** no.of.degrees

which will produce a new picture by rotating the picture p no.of.degrees degrees clockwise about the origin For example,

rotate new **by** 45

will produce the picture given in Figure 3

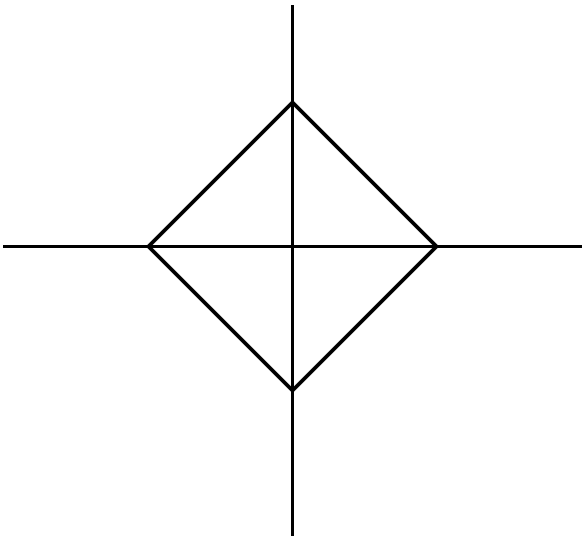


Figure 3: The rotated box

Finally scaling can be obtained by

scale p **by** x.scaling,y.scaling

which will produce a new picture by multiplying the x and y-coordinates of every point in the picture p by x.scaling and y.scaling respectively. For example,

scale box **by** 2,1

yields Figure 4

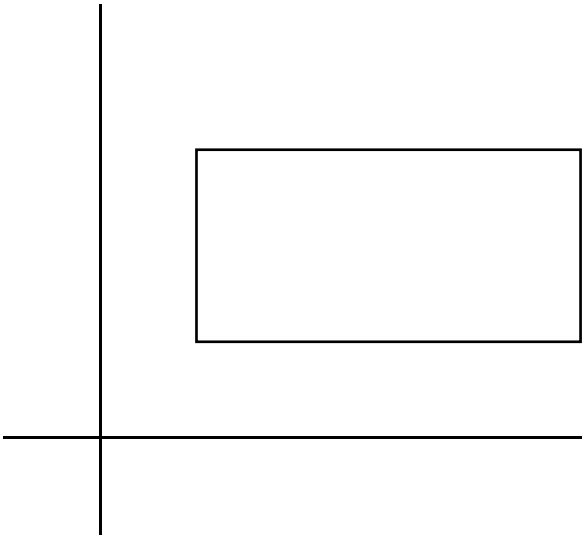


Figure 4: The scaled box

Text can be included in pictures using the text statement. This takes a string of characters and a base line and constructs the picture of those characters along the base line. For example,

text "hello !" **from** 1,1 **to** 2,1

yields Figure 5.

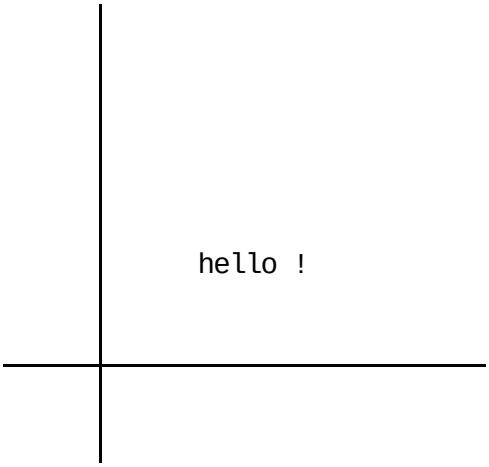


Figure 5: Some text

The characters will always be drawn from the first to last point of the base line. As a consequence text can be inverted by ending the base line on the left of its starting position.

Colour can also be specified in a picture but, unlike the other picture operations, the effect of this will depend on the physical output device used.

5.2 Storing a Picture in a Database

As an example of how pictures may be stored and retrieved from the persistent store we give an example of a program to calculate the unit circle at the origin and store it in the database. In this example we assume that the database root is a pointer to a data structure for associative storage and retrieval, supported by PS-algol, called a table. Entries are placed in the table using the procedure s.enter which takes the associative key, the table, and the value to be stored. The procedure s.lookup retrieves a value from the given table using the given key.

```
structure pic.container( pic a.pic )
let db = open.database( "a pic","pass","write" )
if db is error.record do
begin ! if db points to an error.record the open failed
        write "Unable to open database because: ",
        db( error.explain ),"'n"
        abort
end
let circle =
begin ! this block is an expression describing a unit circle
        let no.of.sectors = 10
        let angle = 90 / no.of.sectors
        let quadrant := [0,1]
        let segment := [0,1] ^ rotate [0,1] by angle
        for i = 1 to no.of.sectors do
        begin
              quadrant := quadrant & segment
              segment := rotate segment by angle
        end
        let semi = quadrant & scale quadrant by -1,1
        ! below is the value of this block expression
        semi & scale semi by 1,-1
end
! a structure containing the circle picture is
! associated with the key "circle"
s.enter( "circle",db,pic.container( circle ) )
! the database "a pic" is now updated
if commit() = nil do write "Circle entered in the data base'n"
```

Figure 6: A program to store a picture of a unit circle in a database.

The database "a pic" now contains a table with a key "circle" which has an associated value of a structure that contains the description of the circle picture. This is shown pictorially below.

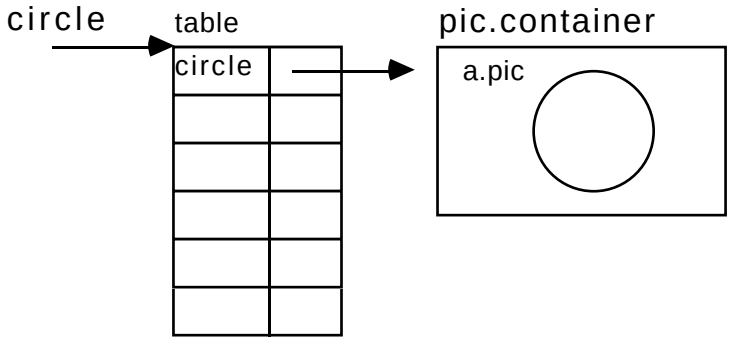


Figure 7: Pictorial representation of the database "a pic" after the transaction is committed

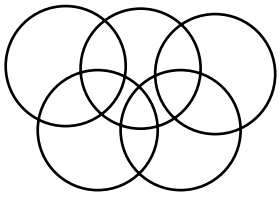
5.3 Retrieving a Picture From a Database

The next example retrieves the picture description from the database and uses it to define another picture which is the Olympic games logo.

```
! this structure will be used to hold pictures kept in this database

structure pic.container( pic a.pic )

let db = open.database( "a pic","pass","read" )
if db is error.record do
begin ! if db points to an error.record the open failed
        write "Unable to open database because: ",
        db( error.explain ),'"n"
        abort
end

let circle = s.lookup( "circle",db )( a.pic )

let olympics = circle &
        shift circle by 1.7,0 &
        shift circle by -1.7,0 &
        shift circle by  0.85,-1.5 &
        shift circle by -0.85,-1.5 &
        text "OLYMPICS" from -2.5,-3.5 to 2.5,-3.5
```

Figure 8: A program to retrieve the circle from the database and define an olympic games logo.

The picture olympics now contains the following:



OLYMPICS

Figure 9: The Olympic games logo

These are the basic support facilities for line drawing. Particular applications packages built on these facilities, for example curve fitting or 3-D modelling, may be stored in and retrieved from the persistent store as pictures themselves or procedures that produce or manipulate pictures. The choice is made according to the requirements of the application.

5.4 Images

An image is a three dimensional object made up of a rectangular grid of pixels. Pixels have depth to reflect the number of planes in them. Images also have an X and Y dimension to reflect their size. In their most degenerate form a pixel is one spot which is either **on** or **off**. Thus

let a = **on**

creates a pixel a with a depth of 1. To form a pixel of depth 4 say we could write

let b = **on** & **off** & **off** & **on**

which creates a pixel b with depth 4. To form an image we could write

let c = **image** 5 **by** 10 **of on**

which creates c with 5 pixels in the X direction and 10 in the Y direction all initially on. The origin of all images, which is at the bottom left hand corner of the image, is 0,0 and in this case the depth is 1.

Full 3 dimensional images may be formed by expressions like the following,

let d = **image** 64 **by** 32 **of on** & **off** & **on** & **off**

which would create an image d of depth 4 with 64 pixels in the X direction and 32 in the Y direction all initialised to the pixel value on & off & on & off. This is illustrated in figure 10.

Figure 10: An image with 4 planes

In order to introduce the concept of and operations on images gently we will restrict ourselves for the present to images with a pixel depth of 1 which is the case on the ICL PERQ. Everything that we say will be true for images of greater depth.

Images are first class data objects and may be assigned, passed as parameters or returned as results. e.

l**et** b = a

will assign the image a to the new one b. In order to map the operations usual on bitmapped screens, the assignment does not make a new copy of a but merely copies the pointer to it. This as we will see later is consistent with update in place raster operations on most raster devices.

There are 8 raster operations which may be used as described in the following BNF.

<void-clause> ::= <raster.op><image-clause>onto<image-clause>
<raster.op> ::= ror|rand|xor|copy|nand|nor|not|xnor

thus

xor b **onto** a

performs a raster operation of b onto a using xor. It should be mentioned that a is altered in situ as would be expected on a raster device. Both images have origin 0,0 and automatic clipping at the extremities of the destination image is performed.

The limit operation allows the user to set up windows in images. For example,

let c = **limit** a **to** 2 **by** 3 **at** 3,1

sets c to be that part of a which starts at 3,1 and has size 2 by 3 c has an origin of 0,0 in itself and is therefore a window on a. This is illustrated in Figure 11.

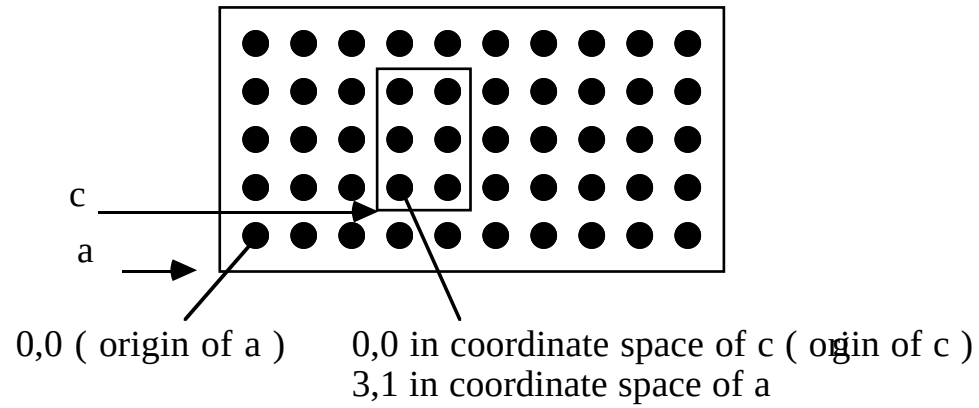


Figure 11: A limited region of an image

Rastering sections of images onto sections of other images can be performed by expressions like the following,

xor limit a **to** 1 **by** 4 **at** 6,5 **onto**
limit b **to** 3 **by** 4 **at** 9,10

Automatic clipping on the edges of the limited regions is performed. If the starting point of the limited region is omitted 0,0 is used and if the size of the region omitted then it is taken as the maximum possible. That is from the starting point to the edges of the host image. Limited regions of limited regions may also be defined The depth of the image may be restricted by the depth selection operation. For example,

let b = a(1|2)

yields b which is that part of a which has the two depth planes 1 and 2. b has depth origin 0 and dimensions 64 by 32.

The PS-algol system provides two functions for manipulating the colour map of the device. The first is,

colour.map(**pixel** p ; **int** i)

This functions sets the integer produced by the colour map when pixel p is displayed to be i.

The second function allows the user to interrogate the colour map

and is,

colour.of(**pixel** p -> **int**)

This function returns the integer corresponding to the pixel p in the colour map.

An example of the use of images may be seen from the program to draw a chess board and store it in a database in figure 13.

The pictorial representation of the database after the transaction has committed is given in Figure 12.

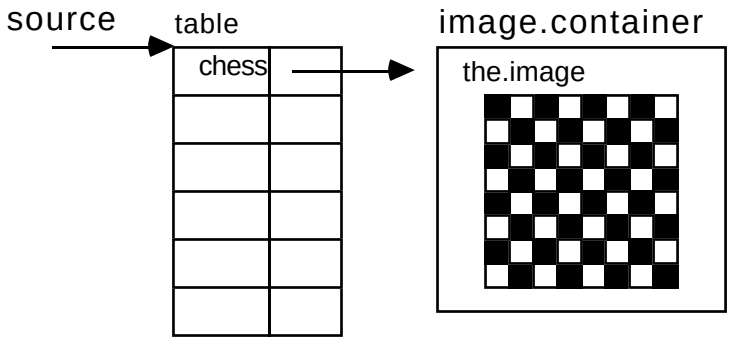


Figure 12: Pictorial representation of the chessboard in the database

```
! This structure will be used to hold images in this database
structure image.container( cimage the.image )

write "Please input the basic size of the squares " ; let size = readi()

let black = off ; let white = on
!define a black square
let black.square = image size by size of black

let size.8 = size * 8 ; let size.2 = size * 2
!define the chess board image
let chess.board = image size.8 by size.8 of white
for i = 0 to size.8 - 1 by size do
for j = 0 to size.8 - 1 by size do
        if i rem size.2 = 0 and j rem size.2 = 0 or
        i rem size.2 ~= 0 and j rem size.2 ~= 0 do
             copy black.square onto limit chess.board at i,j

let source = open.database( "raster.demo","ron","write" )
if source is error.record do
begin
        write "Error opening database : ",source( error.fault ),"'nbye'n"
        abort
end

!a structure containing the chess board image is associated with key "chess.board"
s.enter( "chess.board",source,image.container( chess.board ) )

let done = commit()
if done is error.record do write "Sorry - commit failed: ",done( error.fault ),"'n"
```

Figure 13: A program to store a chess board image in a database

Images may be stored in and retrieved from data bases in the same manner as pictures and thus we have the same facilities for providing libraries of images or procedures that manipulate images.

5.5 Mapping Pictures and Images to Output Devices

The standard identifier screen is an image representing the output screen. Performing a raster operation onto the image screen alters what is viewed by the user. For example,

xor a **onto limit** screen **to** 4 **by** 5 **at** 4,7

will raster a onto the defined section of the screen. This will be visible to the user.

The standard identifier cursor is also bound to an image which is mapped to the cursor. The cursor may be altered in the same manner as any other image. For example, we may say,

copy b **onto** cursor.

In the same way that images are mapped onto standard names, such as screen, pictures may also be drawn on vector devices with standard names. However if such a device is not available or we wish to map a line drawing on a raster device then we may map a picture on to an image using the standard function draw. For example,

draw(an.image,a.pic,0.0,3.2,1.5,3.9)

will draw the section of the picture a.pic bounded by the box specified by the points (0.0,1.5) and (3.2,3.9) on the image an.image. Automatic clipping of the line drawing is performed to make it fit the bounding box.

The picture may be drawn directly on to the image or any part of it. Once the line drawing has been mapped on to an image the image may be manipulated by any of the image operations. Notice also that we can now easily mix pictures and images on a screen choosing which ever one is appropriate for each section of the screen.

5.6 Miscellaneous facilities

We have now seen the basic graphics facilities provided by PS-algol. These facilities are the building blocks and many useful functions can be built with them.

One example of this added power is the ability of the language to manipulate fonts. Fonts are stored in a database which may be freely interrogated by the programmer. The layout of the font database is given in figure 14.

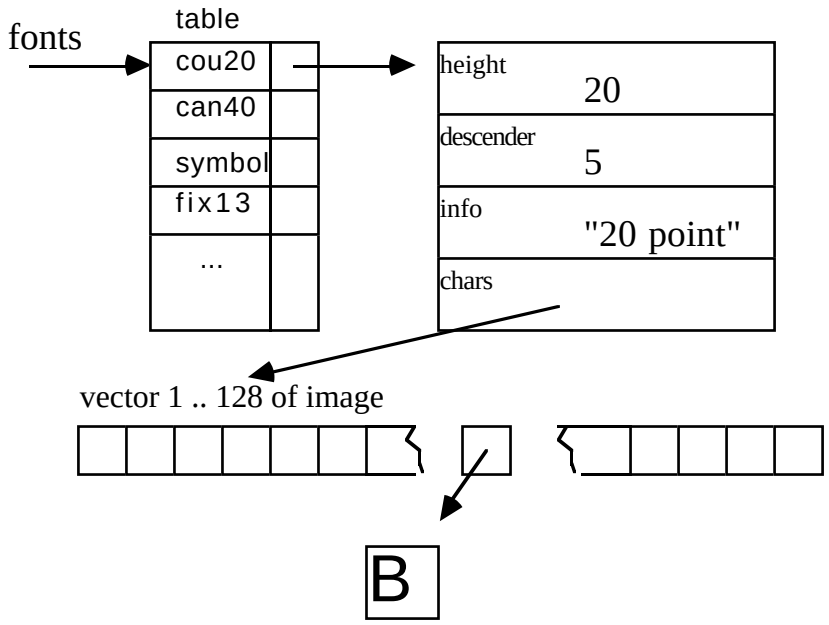


Figure 14: The font database

The programmer may not want to have to deal with the fonts database directly so a standard function written in the language is provided by the system. It is called string.to.tile and has the following type.

let string.to.tile = **proc**(**string** source,font -> **image**)

The procedure returns an image containing a representation of the string source in the font specified by the parameter font. These images can then be used for putting messages on the screen, on the cursor or as part of pictures being built up. They are often used in conjunction with the pop up menu mechanism which is also provided by the language. Like string.to.tile the menu function is not a primitive feature but is written using the features we have already seen and another function which allows the programmer to interrogate the pointing device. The menu function has the following definition,

```
let menu :=     proc(     image title ;
                *image entries,          ! vector of images
                bool vertical ;
                *proc( image,int ) actions,   ! vector of procedures
                -> proc( int,int -> bool ) )   ! returns a procedure
```

The menu which the user sees will have a title corresponding to the image title and will have entries corresponding to the vector of images entries. The function menu returns a function which when called will put a menu on the screen at the specified position and allow the user to make a selection from it. If an entry is selected the corresponding procedure from the vector actions is executed, the entry and position of the entry used to select it is passed to it as a parameter. If the user does make a selection the procedure will true otherwise it will return false. In this way many of the costly calculations that need to be made by the menu function need only be done once. This can be prior to the execution of the main program if the function returned by menu is stored in the persistent store.

5.7 User Interaction

In order to write a function like the menu function shown above it is necessary to interact with the pointing device and cursor.

As the pointing device (usually a mouse) is moved around the cursor follows it (unless the standard function which switches off tracking has been called). In order to find out where the cursor is the system provides a standard function called locator. locator has the following form,

let locator = **proc**(-> **pntr**)

it returns a structure of the following type,

structure mouse(**int** X.pos,Y.pos ; ***bool** the.buttons)

The fields X.pos and Y.pos give the position of the mouse relative to the standard identifier screen. The vector of booleans shows the current status of the mouse buttons. One of these structures is returned every time locator is called.

One problem in writing code which interacts with the user via a pointing device and the keyboard is that it is often necessary to know if the user has typed something or not. In order that this may be discovered the system provides a standard function called input.pending. This function returns a boolean which indicates if there is any input waiting to be read from the keyboard.

We have already seen that the cursor is an image in the system so it is possible to alter its appearance in order to provide visual feedback. We must also be able to specify which pixel is the cursor is the pointing tip so that it may be used as a pointer. In PS-algol the function cursor.tip provides this, it has the following type,

let cursor.tip = **proc**(**pntr** new.tip -> **pntr**)

In order that the old tip may be reinstated later this function returns the old cursor tip. In this way cursors which look like arrows and cross hairs may be used with the appropriate pointing tip.

Other standard functions provided include a seed fill operation, functions to find the size of images and extract pixels from images. These facilities are all stored in the persistent store and may be called by the user when required.

6 Implementation Issues

The main implementation issues in the graphics support in PS-algol centre around the ability of the host machine to support the fundamental picture and image operations in the language. The efficiency of the system may be greatly impaired by performing these operations on unsuitable hardware.

For example, raster operations on most mainframe computers are prohibitively expensive although alternative techniques have been found [14].

Normally we would not be performing graphics manipulation unless we eventually had some suitable device to draw on. With this in mind the language implementor can choose to make the evaluation of the graphics operations lazy rather than strict. That is instead of performing the operation immediately a data structure is built to represent the evaluation of the expression. When the object is eventually drawn then the data structure may be evaluated in the context of the output device thereby ensuring the

maximum efficiency for the hardware available. Of course this lazy evaluation would be hidden from the user for the most part.

Another advantage of this lazy evaluation is that the data structure (unevaluated) may be used to represent the object in the persistent store. It is unusual to find computers that support line drawing in the hardware and therefore the picture data type will nearly always be represented by a data structure. In fact the data structure for PS-algol pictures is a directed acyclic graph (DAG). It is a graph since there may be several paths to the same sub-structure, directed since the ordering of the points uniquely defines a traversal and acyclic since recursive referencing is not possible for pictures. A fuller description of how this may be done is given in [13].

With images the solution to the problem of strict or lazy evaluation is not so simple. A number of modern computer systems support raster operations directly in the microcode or hardware. The choice of which to use is left to the language implementor with a word of warning. With vector devices it is a simple enough matter to construct a DAG from a picture input by the user. It is not such a trivial matter to do the same for raster devices. Therefore, if lazy evaluation is chosen as the implementation technique then we are faced with the problem of trying to store screen bitmaps and image DAG's in the databases. The added complication of using DAG's in addition to quadtrees [7], space filling curves [5] or run length encoding for images seems unnecessary on hardware that supports raster operations but may be forced on the implementor for unsympathetic hardware.

The present system implemented on the ICL PERQ which has microcode assist for raster operations uses lazy evaluation for line drawings and strict evaluation for images. The implementation issues for persistence are described elsewhere [4].

7 Graphical data and the persistent store

Program libraries may be obtained in PS-algol by storing procedures in a database. Picture and image libraries may be similarly be obtained by storing pictures and images in a database. Since the system allows any mixture of data to be stored then the user has the choice of storing the graphical information in the form thought best for the particular application. This may mean storing pictures as pictures or as images or even as procedures which will produce pictures. Images may be stored as images or as procedures which will produce them. The flexibility of the system illustrates that it is ideal base for all Human Computer Interaction. The data, graphical in this case, could be digitised speech or even analogue data as long as we have a clean interface to it as epitomised by the graphics facilities of PS-algol. One of our main research areas in the future will be to use the persistent store to integrate the dispirit forms of data used in Human Computer Interfaces.

The transaction mechanism together with the concurrency protocols on databases ensure that the user has exclusive write access or shared read access to the data. This is a common requirement in large systems used for CAD. For example, a CAD system may be used for many projects with the data and programs for each project kept in separate databases. It is inevitable that some program and picture tools will be shared between projects. Indeed it is highly desirable. These facilities are provided for all data in PS-algol and therefore need not be specially provided for graphical systems.

Data components in the persistent store may be replaced individually. Other users see change after a transaction incorporating the change has been committed. Thus change to the data is controlled. Programs which saw the old data before a transaction will be able to view the new data after the transaction. If the data is itself a procedure, then the mechanism can be viewed as incrementally binding the data to the program. Large and complex data structures and their associated programs may evolve in a controlled

manner using this mechanism. It is no surprise to us that the main uses of our system so far has been in the provision of graphical tools and in the rapid prototyping of systems involving graphics and databases.

It would of course be possible to add the PS-algol graphical facilities to other languages. The strength of the system is not particularly in the graphics features themselves, although novel, but in the general mechanism of persistence to provide a consistent framework in which to build an integrated system. The SMALLTALK system [6] also subscribes to this philosophy. Without such a framework integrated systems become very expensive to build.

8. Conclusions

The PS-algol system provides a persistent store facility with a secure transaction mechanism. The facilities and richness of the environment depends on the facilities and richness of the programming language and is independent of the persistence concept. We have shown how the inclusion of line drawing and bitmap graphics facilities in a language that supports a persistent store provides an integrated graphics programming environment. We suggest that CAD systems or any large graphics applications are ideally suited to implementation using these techniques.

The graphics facilities of PS-algol do not and are not intended to form a graphics standard. However they would make an ideal base on which to build a system implementing a standard such as GKS. The persistent store allows for the storage of pictures and functions relating to pictures while the graphics facilities are sufficient to implement the requirements for both vector and raster devices. Some interactive computing facilities would have to be added to the language to model the machine independence of GKS but this could be achieved by a set of functions in a database relating to a particular device and employing the delayed binding mechanism of the persistent store to achieve specialisation.

9 Acknowledgements

This work is supported at the University of Glasgow by SERC grants GRC 21977 and GRC 21960 and at the University of St Andrews by SERC grant GRC 15907.

The work is also supported at both Universities by grants from International Computers Ltd.

Pete Bailey and Paul Cockshott must also be thanked for their part in providing the implementation of PS-algol.

10 References

[1] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. An approach to persistent programming. Computer Journal 26, 4 (1983), 360-365.

[2] Atkinson, M.P., Bailey, P.J., Cockshott, W.P. & Morrison, R. PS-algol reference manual. Universities of Glasgow and St Andrews PPRR-12 (1984).

[3] Atkinson, M.P. & Morrison, R. First Class Functions are Enough. Proc. 4th International Conference on Software Technology and Theoretical Computer Science. Bangalore, India(1984). In Lecture Notes in Computer Science, 181 (1984), 223-240. Springer-Verlag.

[4] Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. The persistent object management system. Software, Practice & Experience 14 (1984), 49-71.

[5] Cole, A.J. Compaction techniques for raster scan graphics using space filling curves. University of St Andrews. CS/84/2. (1984).

[6] Goldberg, A. & Robson, D. SMALLTALK-80 The language and its implementation. Addison-Wesley, London. (1983).

[7] Hunter, G.M. & Steiglitz, K. Operations on images using quadtrees. IEE trans. on Pattern Analysis and Machine Intell. 1,2 (1979), 145-153.

[8] Ichbiah et al. The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).

[9] I.C.L.Introduction to PERQ. International Computers Ltd. RP10103. (1983).

[10] Liskov, B. & Zilles, S.N. Programming with abstract data types. ACM Sigplan Notices 9, 4 (1974), 50-59.

[11] Morrison, R. S-algol language reference manual. University of St Andrews CS/79/1 (1979).

[12] Morrison, R. Low cost computer graphics for micro computers. Software, Practice & Experience 12, 8 (1982), 767-776.

[13] Morrison, R., Bailey, P.J., Podolski, Z. & Weatherill, M. High level language support for computer graphics. Eurographics 83, Zagreb, Yugoslavia. North-Holland. Amsterdam, (ed P.J.W.Hagen) (1983), 7-15.

[14] Pike, R., Locanthi, B. & Reiser, J. Hardware/Software trade-offs for bitmap graphics for the Blit. Software, Practice & Experience 15, 2 (1985), 131-151.

[15] Smith, D.N. GPL/1 - A PL/1 extension for computer graphics. AFIPS SJCC (1971), 511-528.