

This paper should be referenced as:

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A. "A Persistent Graphics Facility for the ICL PERQ Computer". *Software – Practice and Experience* 16, 4 (1986) pp 351-367.

A Persistent Graphics Facility for the ICL Perq

R. Morrison, A.L. Brown, P.J. Bailey, A.J.T. Davie, A. Dearle.

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

Summary

The facilities of the PS-algol programming language are described in this paper to show how they may be used to provide an integrated programming support environment. The persistent store mechanism and the secure transaction facilities provide the basic environment in which an integrated system may be implemented. In particular the paper makes use of the data type picture of PS-algol to show how such an environment may be built for a graphics system ideal for use with a medium range computer workstation. An implementation of a picture editor on the ICL PERQ workstation is described to show the utility of the system.

Keywords: Persistent store graphics algol transactions workstation ICL PERQ

Introduction

The inclusion of a graphics facility in a language that supports an integrated persistent environment yields an ideal programming environment for a medium range graphics workstation such as the ICL PERQ[7]. The integrated persistent store allows data to be stored and retrieved automatically from user named databases. If one of the legal data types in the language is a picture then pictures may be stored 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 program libraries. Thus the users have a well defined integrated mechanism in which to access and store pictures. This provides the user with 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.

Thus fourth generation seamless systems such as the proposed software for the Apple Lisa with picture editors, menu systems and mixed picture, text and program documents may be built out of the integrated persistent environment.

This paper describes the facilities available in the language PS-algol[2] on the ICL PERQ for the building and the use of persistent pictures. The system supports a general notion of transactions and therefore allows transactions on pictures. An example of how such a mechanism might be used is included in the paper.

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 programmers 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.

PS-algol

PS-algol was developed from the programming language S-algol[10] as an experiment in integrating a long term or persistent store with a programming language. The base types are integer, real, boolean, string and picture. These 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 T1,...,Tn and a data type T, proc(T1, ..., Tn -> T) is the type of a procedure of n parameters with types T1, ..., Tn that returns a result of type T.

The range of PS-algol data types includes several unusual features. Firstly string is a simple data type[11] resulting in a very powerful string handling capability. Secondly the data type pointer may point to any structure class but, when a field of a structure is referenced a check is made that the structure present is of the appropriate class. Using this polymorphism over the data type pointer, arbitrarily complex data types can be manipulated without reference to their actual structure and this is the basis of the persistent store implementation[3].

PS-algol's persistent store consists entirely of legal PS-algol data objects. The store is partitioned into databases to allow some 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 a picture editor. 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 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[9]. These abstract data objects are usually implemented by returning a pointer to a structure containing procedures and constants from a creating procedure or block. This new data object can then be manipulated in the same way as any other data object in PS-algol. For example:

```

structure a.stack( proc( -> int ) pop ; proc( int ) push )

let stack = proc( int size -> pntr )
begin
  let entries = vector 0 :: size of 0
  let stack.pntr := 0

  let pop.stack = proc( -> int )
  begin
    if stack.pntr > 0 then stack.pntr := stack.pntr - 1
    else write "Popping an empty stack'n"
    entries( stack.pntr ) ! This is the value being returned.
  end

  let push.stack = proc( int new.entry )
  if stack.pntr < size then
  begin
    entries( stack.pntr ) := new.entry
    stack.pntr := stack.pntr + 1
  end else write "Pushing a full stack'n"

  a.stack( pop.stack,push.stack ) ! This is the value being returned.
end

```

Figure 1: A procedure to implement a stack abstract data object.

This program segment creates a stack abstract data object and when called creates a new stack of a given size with two operations on it, pop and push. The following is an example of how the operators are accessed. When called the procedure stack returns a structure with two fields both of which are procedures. Notice that the representation of the stack is invisible to the rest of the program by the normal algol scope rules and only visible to the two procedures. In this case an integer is pushed onto a newly created stack of size 20 and then popped off.

```

let a.stack = stack( 20 )

a.stack( push )( an.integer )

an.integer := a.stack( pop )()

```

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. 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.

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 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()

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. 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[3].

Picture Description 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 infinite two dimensional real space. 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 a procedural interface for the desired output device. Therefore 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.

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

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.

rotate p by no.of.degrees

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

scale p by x.scaling,y.scaling

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. 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.

let p = text "hello !" from 1,1 to 2,1

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.

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 store and lookup, 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 from the given table using the given key.

```

!
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","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
commit()

```

Figure 2: 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.

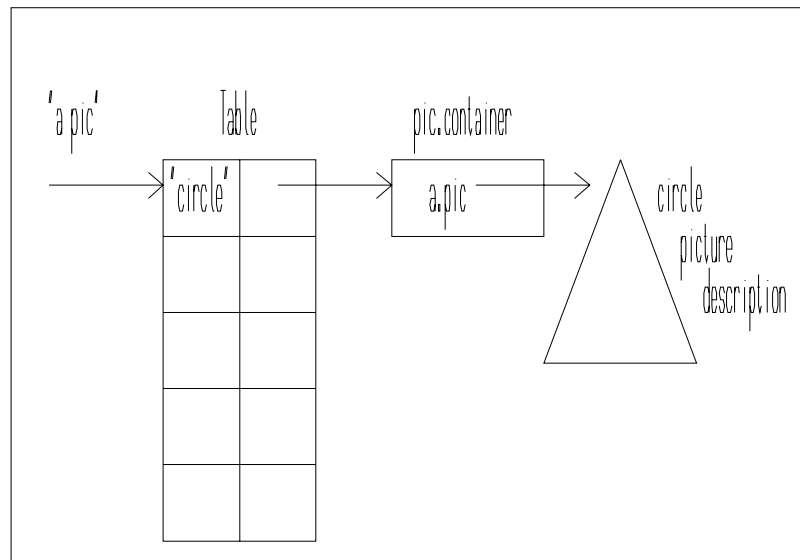


Figure 3: Pictorial representation of the database "a pic" after the transaction is committed

Retrieving a Picture From a Database

The second example retrieves the picture description and uses it to define a procedure to draw circles or ellipses of any size at any point.

```

structure pic.container( pic a.pic )

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

! the structure associated with the key "circle" is retrieved and
! its picture field extracted
let circle = s.lookup( "circle", db )( a.pic )

! the ellipse procedure can now use the circle to construct ellipses
let ellipse = proc( real x.scale, y.scale -> pic )
shift scale circle by x.scale, y.scale by x.scale, y.scale

```

Figure 4: A program to retrieve the circle from the database and define an ellipse procedure.

The ICL Perq

The ICL Perq is a powerful personal workstation designed to support bitmapped graphics. PS-algol is implemented on Perqs running the PNX[5], (Perq UNIX) operating system. Support for graphical output on the Perq under PNX includes a bit mapped screen, a window manager to control the screen and special hardware to support copying and line drawing on bit maps. Graphical input is supported by a tablet and mouse.

The Perq window manager provides the ability to create virtual devices. Each of these devices is a bitmap with possibly a border and title, that can be used for normal text output or as the destination of graphical output. Using the window manager it is possible to create, destroy, move or change the size of individual devices. It is also possible to associate input from the Perq keyboard or mouse with individual devices.

Interface to Graphics Devices

In PS-algol every graphical output device is accessed by an abstract data object whose operators are the Outline procedures used for drawing. These procedures are 'draw', 'erase.to', 'limit', 'g.input', 'print', 'activate' and 'shutdown'. The PS-algol structure used to hold the procedures for an output device also contain two constants that indicate the dimensions of the output device.

A predefined PS-algol procedure is available to create such structures. It is used to create a window implemented as a virtual device on PNX. This window can have several attributes specified, such as a border or title. In addition the dimensions of the window must be given as well as its screen position. For example to use a window 600 pixels square at screen location 200,200 with a border and the title "Hello" the following would be used.

```
let a.window = window.pac( "Hello",border,600,600,200,200 )
```

Since procedures are assignable objects in PS-algol it is now possible to get the individual procedures for operating on this window and rename them. e.g.

```
let drawer = a.window( draw )
```

```
let eraser = a.window( erase.to )
```

The system also has facilities (not given here) for window creation on other physical devices which may be attached to the Perq.

Picture Drawing Operations on the Perq

The procedures used to operate on windows will now be described. Each of these procedures will only operate on the window for which it was created.

'Draw' takes a picture and a bounding box in the infinite two dimensional space over which all pictures are defined. This box is scaled and shifted to fit the area of the window currently in use. Then any intersection between the picture described and this box is drawn on the window. The colours recognised by 'draw' are "background", "foreground" and "opposite". Black and white cannot be used because what appears as black and white can be dynamically altered by the user. Hence it is only sensible to use "foreground" and "background". The colour "opposite" is provided to invert the colour of the pixels it affects.

'Erase.to' is used to set the colour of pixels in the area of the window currently in use. It takes as a parameter a string representing one of the three colours recognised by 'draw'. This can be effectively used to draw a chess board, pop up menus or using the colour "opposite" to highlight areas of a window.

'Limit' is provided to redefine the area of the window currently in use. Its parameters are the ranges of x and y values, in pixels, of the new area required. These are checked to make sure the new area lies within the window.

Graphical input is achieved by use of the procedure 'g.input'. This provides a user with access to the PNX interface to the Perq mouse. Consequently the information returned includes the status of the mouse buttons, whether any keyboard input is available and three versions of the

mouse position. The mouse positions are relative to the tablet, screen and the window for which 'g.input' was called. All the mouse positions are inverted to correspond to Outline's screen addressing and are in pixels.

To obtain a hard copy of a drawn picture the 'print' procedure is included. The operation of 'print' can vary from using a dedicated printer to preparing a device independent version of the window's display for later processing and printing. What is available depends on the individual Perq.

When the packages of procedures is first created it is only a description of what the window will actually do. 'Activate' is provided to instantiate this description by creating a virtual device that implements the window. This newly created device appears on the Perq screen on top of any other displayed devices at the position given in the window's description. However once created the device may be moved around the screen at any time by the user. When the device is created 'activate' ensures that both keyboard and mouse input are connected to it.

'Shutdown' is used to destroy a virtual device created by 'activate' and so remove it from the screen. Both of these procedures perform some state preservation or restoration on a virtual device.

Saving Drawn Pictures

The package of procedures used to operate on windows can, like any other data object in PS-algol, be stored in the persistent store. In some cases it would also be desirable to store the drawn picture rather than the abstract representation of it. Such a facility is provided by the 'activate' and 'shutdown' procedures. The package of procedures used to operate a window keeps an internal copy of what is displayed on the window. When the 'activate' procedure is called this internal copy is displayed on the newly created virtual device. When the 'shutdown' procedure is called the internal copy is overwritten by whatever is currently displayed on the virtual device. Three steps are therefore required to store a complete window description, including what is currently displayed, in the persistent store. Firstly the window description must be placed in the database. Secondly 'shutdown' must be called to complete the window description. Finally commit should be used to update the copy of the database in the persistent store.

A Menu Driven Picture Editor.

To demonstrate how Outline may be used on the Perq to construct an interactive picture editor the following example program is presented. This program implements the functional geometry model of Henderson[4] as an interactive picture editor. Henderson's model consists of six basic operations namely, 'draw', 'beside', 'above', 'overlay', 'flip', and 'rotate'. We use Henderson's operations as some of the base commands of our interactive picture editor. In this implementation all the pictures created are kept in the picture editor's database so that previously created pictures are always available. Every picture is given a name which is used to identify it in the database's root table. These names are unique PS-algol strings except for "nilpic" and "screen" and "cache". "nilpic" is the name of an empty picture, "screen" and "cache" are the editor's names for its Perq window package and the names of pictures held in a picture cache.

The user interface is organised around the concept of a current picture that the user may modify or replace. This is supported by a Perq window that has its drawing area divided between a cache of recently used pictures, a large display of the current picture, a menu of commands available and a picture describing the current operation of the Perq mouse buttons. The cache includes an entry called "temporary" that is the current picture as it was when the last command was completed. This enables references to the current picture while constructing composite pictures. All of the window components are laid out to enable them

to be easily identified from the position of the Perq mouse. The resulting window layout is shown in Figure 5.


Geometric Picture Composition				
nilpic				
				temporary
select  abort	Title:			
quit				
rename				
restore				
save				
draw				
erase				
beside				
above				
overlay				
flip				
rotate				

Figure 5: An example of the Perq window used by the picture editor.

In addition to the operations of Henderson's model, several house keeping operations are provided. These include renaming a picture, placing the current picture in the database, selecting a new current picture, erasing the current picture, ending the editing session or committing the changes made so far to the editor's database. At the end of an editing session any uncommitted changes to the editor's database are discarded.

Initialising the Picture Editor's Database

```
! size and position of the editor's Perq window
let width = ...
let height = ...
let x.position = ...
let y.position = ...

! the picture of the screen background and menus of figure 5.
let background = [0,0] ^ [0,height] ^ [width,height] ^ [width,0] ^
    .
    .
    .
    .
    .
    ! the details of the background picture.
    .
    .
    .
    .
    .

let screen = window.pac( "Geometric Picture Composition",border,
    x.position,y.position,width,height )
screen( activate )()
screen( draw )( background,0,width,0,height )
screen( shutdown )() ! screen now has an internal copy of the
    ! drawn background picture.

! the empty picture.
structure pic.container( pic picture )
let nilpic = pic.container( text "" from 0,0 to 1,0 )

! the names of pictures held in the cache.
! **string is the type of a vector of vectors of strings.
structure cache.container( **string cache.names )
let cache = cache.container( vector 0 :: 4,0 :: 1 of "" )
! the temporary picture in the cache is at position 4,1.
! initially this is the empty picture nilpic.
cache( cache.names )( 4,1 ) := "nilpic"

! create and open the editor's database for writing.
let pic.db := open.database( "Pictures","Geometry","write" )
if pic.db is error.record do
begin
    write "Error creating the database: ",pic.db( error.explain ),"n"
    abort
end

! enter the screen, empty picture and cache names in the database table.
s.enter( "screen",pic.db,screen )
s.enter( "nilpic",pic.db,nilpic )
s.enter( "cache",pic.db,cache )

! commit the changes to the database to the persistent store.
commit()
```

Figure 6: A program to initialise the picture editor's database.

The picture editor is divided into two parts, an initialisation program and the editing program. The initialisation program shown in Figure 6 constructs a database for the picture editor that consists of the editor's Perq window, the empty picture "nilpic" and a two-dimensional vector holding the names of the pictures held in the cache. To simplify mapping the coordinates of the background to the Perq window, the background picture and Perq window were given the same dimensions.

The Picture Editing Program.

The main parts of the editing program will now be described. Figure 7 shows the initialisation of the picture editor from the persistent store. This segment of code together with that shown in Figure 8 and a single call to 'commit' form the total overhead in programming the persistent data. As will be seen this is only a small part of the complete program.

```

! open the database for writing.
let pic.db := open.database( "Pictures","Geometry","write" )
if pic.db is error.record do
begin
    write "Error finding stored pictures: ",pic.db( error.explain ),"n"
    abort
end

! retrieve and display the editor's Perq window.
let screen := s.lookup( "screen",pic.db )
screen( activate )()

! retrieve the names of the pictures held in the cache.
structure cache.container( **string cache.names )
let cache := s.lookup( "cache",pic.db )( cache.names )

```

Figure 7: A program segment to retrieve the editor's state from a database.

The database table performs a mapping from strings to pointers. Therefore pictures are held in individual structures, pointers to which are used in the database table. To abstract over this level of indirection the two procedures of Figure 8 are used to simulate a table of strings to pictures.

```

! the structure used by the database table to hold pictures.
structure pic.container( pic picture )

! define a procedure to lookup pictures in the database.
let lookup.pic = proc( string name -> pic )
    s.lookup( name,pic.db )( picture )

! define a procedure to enter pictures in the database.
let enter.pic = proc( string name ; pic apic )
    s.enter( name,pic.db,pic.container( apic ) )

```

Figure 8: Procedures to enter and lookup pictures held in the database table.

The maintenance of the editor's Perq window is performed using procedures similar to those in Figure 9. Each of these uses the 'limit' procedure to restrict its operation to the desired window area. Where necessary these procedures will also modify the data structures holding the abstract representation of what is currently displayed.

```
! define a procedure to invert the colour of a window area.
! this is used for highlighting during command selection.
let invert = proc( int x1,x2,y1,y2 )
begin
    screen( limit )( x1,x2,y1,y2 )
    screen( erase.to )( "opposite" )
end

! define a procedure to erase the current picture.
let erase.draw.area = proc()
begin
    screen( limit )( current.pic.left,current.pic.right,
                    current.pic.bottom,current.pic.top )
    screen( erase.to )( "background" )
    current.pic := nilpic
end

! define a procedure to draw a new current picture.
let fill.draw.area = proc( pic picture )
begin
    screen( limit )( current.pic.left,current.pic.right,
                    current.pic.bottom,current.pic.top )
    screen( draw )( picture,0,1,0,1 )
    current.pic := picture
end
```

Figure 9: Procedures to update the picture editor's Perq window.

The Perq mouse is used to select commands or pictures for commands to operate on. The selection procedures are based on a knowledge of the background picture and where the mouse is when its yellow button is pressed. During command selection the colour "opposite" is used to highlight a command when the mouse is moved over it. To prompt the user to select a command the whole command menu is highlighted. After command selection the selected command is left highlighted. The resulting selection procedure is similar to the menus provided by PNX 2[5] and POS[6].

```

! define a block whose value is a procedure to select commands.
let select.command =
begin
    ! variables to control highlighting of menu entries, these are not within
    ! the selection procedure since they must exist between calls.
    let old := -1 ; let new := -1

    ! invert the colour of the selected menu entry.
    let invert.command = proc( int num )
    begin
        let y1 = num * entry.height ; let y2 = y1 + entry.height
        invert( 0,menu.width,y1,y2 )
    end

    ! this procedure is the value of select.command.
    proc( -> int )
    begin
        ! invert the colour of the whole menu to prompt the user.
        invert( 0,menu.width,0,menu.height )
        let mouse := nil
        repeat
        begin
            ! g.input returns a pointer to a structure in which wrtabX and
            ! wrtabY are fields giving the window relative mouse position.
            mouse := screen( g.input )()
            ! calculate which menu entry is pointed at.
            new := if mouse( wrtabX ) < menu.width and
                mouse( wrtabY ) < menu.height
                then mouse( wrtabY ) div entry.height
                else -1
            ! if necessary highlight the menu entry pointed to.
            if old \(!= new do
            begin
                if old ~= -1 do invert.command( old )
                if new ~= -1 do invert.command( new )
                old := new
            end
        end
        while new = -1 and ~mouse( yellow.but )
        ! invert the colour of the whole menu to leave the
        ! selected command highlighted.
        invert( 0,menu.width,0,menu.height )
        new
    end
end

```

Figure 10: A procedure to select a command from the menu.

Since only a few pictures are held in the cache, picture selection using the mouse needs to be augmented by text input. This is made possible by using part of the mouse input that indicates if text input is available. This is demonstrated by the 'select.picture' procedure of Figure 11. In a similar way to command selection, highlighting is used in picture selection but this is omitted from Figure 11.

```

! define procedure to select a picture.
let select.picture = proc( -> pic )
begin
    ! prompt the user.
    write "select a picture or type its name:" ; flush( s.o )
    let picked.pic := ""
    while picked.pic = "" do
        begin
            ! read the Perq mouse.
            let mouse = screen( g.input )()
            ! if text input is available read a line of text from the keyboard.
            ! this line is the name of the picture to be selected.
            if mouse( keys ) then picked.pic := read.a.line() else
            ! no text available so see which cache picture is pointed at.
            if mouse( yellow.but ) and
                mouse( wrtabX ) < width and
                mouse( wrtabY ) \(>= cache.bottom and
                mouse( wrtabY ) < height do
            ! lookup the name of the picture pointed to.
            picked.pic := cache( mouse( wrtabX ) div cache.width,
                ( mouse( wrtabY ) - cache.bottom ) div
                cache.height )
        end
        ! look up the named picture in the database table.
        lookup.pic( picked.pic )
    end
end

```

Figure 11: A procedure to select a picture.

Each command is implemented by a separate procedure two examples of which are in Figure 12. Henderson's model is applicative in nature so commands such as 'above' create a new picture when they are used. The details of this are shown in Figure 12.


```

let above = proc() ! above.command
begin
    ! find out the name of the new picture.
    let name = get.string( "Enter the name of the new picture:" )

    ! find out what the two component pictures will be.
    write "Pick two pictures to be placed first above the second'n"
    let p1 := get.pic()( the.drawing )
    let p2 := get.pic()( the.drawing )

    ! find the ratio of heights between the two selected pictures.
    write "Enter the relative heights of the two pictures'n"
    let m = get.int( "first picture:" )
    let n = get.int( "second picture:" )

    ! construct the new picture scaling the result to fit a unit square.
    p1 := scale p1 by 1,m / ( m + n )
    p2 := scale p2 by 1,n / ( m + n )
    let new.pic = shift p1 by 0,n / ( m + n ) & p2

    ! change the current picture.
    erase.draw.area()
    fill.draw.area( new.pic )

    ! change the temporary entry in the picture cache.
    erase.cache.pic( 4,1 )
    fill.cache.pic( 4,1,name,new.pic )
end

! the variable continue is used to control the editor's command loop.
let continue := true
! the quit command terminates the picture editor by resetting continue.
let quit = proc() ; continue := false

```

Figure 12: Two of the procedures that execute the commands available.

The procedures that execute commands are held in a vector with the same ordering as the menu of commands. Hence the menu position of a selected command can be used directly to find the procedure that implements that command. This is shown in Figure 13.

```

! the vector of command procedures.
let menu.commands = @0 of proc()[ rotate,flip,overlay,above,beside,
                                erase,draw,save,restore,rename,quit ]

while continue do menu.commands( select.command() )()

```

Figure 13: The command loop of the picture editor program.

An example of the picture editor in use.

Figure 14 shows an example of the picture editor's Perq window during the construction of the Escher picture used in Henderson's paper[4]. On the window the cache contains the four starting pictures of fish, a combination of the four fish under an operation called 'quartet' and a side of the final Escher picture. The current picture is a construction using "fish q" and the operations 'rotate' and 'cycle'. The definition of 'quartet', 'cycle' and the side mentioned can be found in[4].

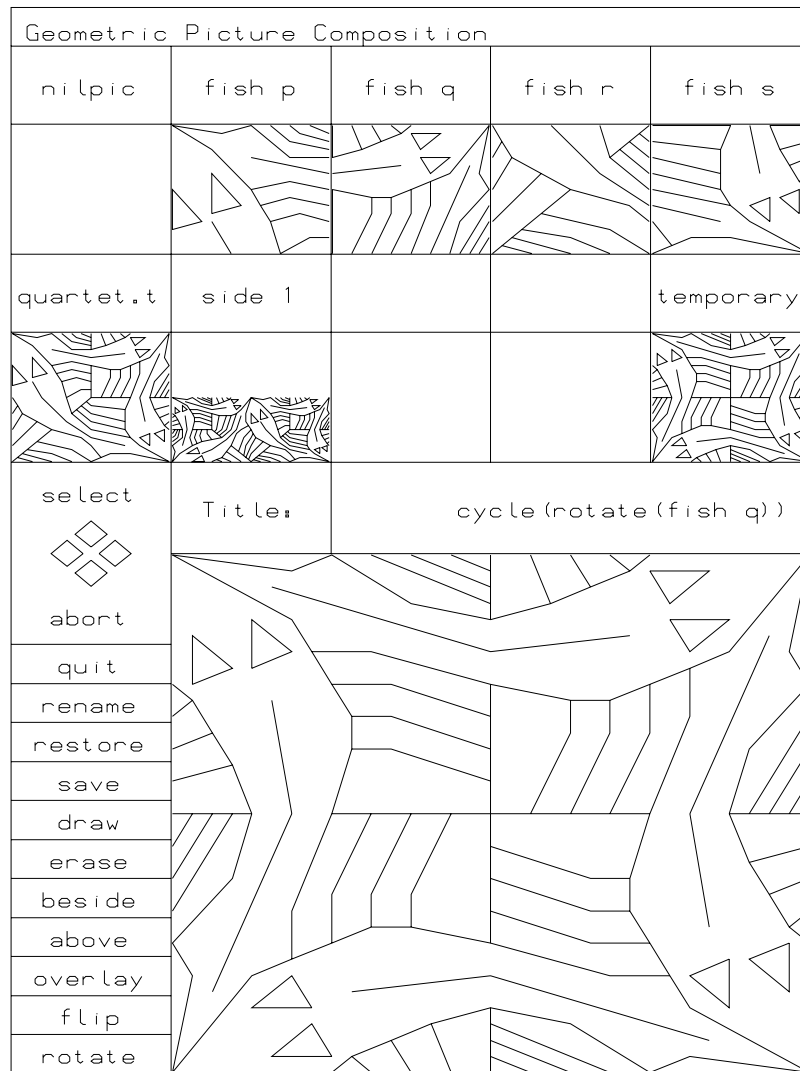


Figure 14: An example of the picture editor's Perq window in use.

Conclusions

The PS-algol system provides a persistent store facility with a secure transaction mechanism. We have demonstrated how such facilities may be used to implement an integrated programming support environment. 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 taken advantage of the data type picture in PS-algol to demonstrate one such environment, that of a picture editor implemented on the ICL PERQ. We conclude that the technique may be used for other environments such as the Ada APSE[13].

Acknowledgements

This work is supported by SERC grant GRC 15907 and a grant from ICL. We acknowledge the close collaboration with Professor Malcolm Atkinson's group at Glasgow University in this work and other aspects of language design.

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 Edinburgh and St Andrews PPR-8(1984).
3. Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. The persistent object management system. *Software, Practice & Experience* 14 (1984).
4. Henderson, P. Functional Geometry. *ACM Symposium on Lisp and Functional Programming, Pennsylvania* (1982), 179-187.
5. I.C.L., Introduction to PERQ PNX. International Computers Ltd. R10134100. (1984).
6. I.C.L., Introduction to PERQ POS. International Computers Ltd. RP10100. (1982).
7. I.C.L., Introduction to PERQ. International Computers Ltd. RP10103. (1983).
8. Ichbiah et al., The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).
9. Liskov, B. & Zilles, S.N. Programming with abstract data types. *ACM Sigplan Notices* 9, 4 (1974), 50-59.
10. Morrison, R. S-algol language reference manual. University of St Andrews CS/79/1 (1979).
11. Morrison, R. The string as a simple data type. *Sigplan Notices* 17, 3 (1982), 46-52.
12. Morrison, R. Low cost computer graphics for micro computers. *Software, Practice & Experience* 12, 8 (1982), 767-776.
13. Draft Specification of the Common APSE Interface Set (CAIS), Version 1.1; September 1983; KIT/KITIA CAIS Working Group for the Ada Joint Program Office; Report NTIS AD-A134825.