22

PostScript for archaeological drawings

Kelvin Goodson*

22.1 Introduction

Integrating text and graphics is of fundamental importance in documenting archaeological information. Books and catalogues conventionally provide the medium for recording, but computer databases offer a convenient alternative. There are disadvantages associated with each method. Books and catalogues must, to a great extent, be processed serially by a human reader and have limited explicit cross referencing. Databases impose a rigid structure onto the recording process, which may give rise to losing certain information that doesn't quite fit the predefined format, while many database user interfaces are so unfriendly that they deter people who are not computer literate from using them.

One significant failing of most database management systems is their inability to incorporate and retrieve graphical information, or to display the results of data retrievals by synthesising graphical representations. This paper describes the advantages of using the PostScript programming language for graphics programming in general, and how its integration with database systems can significantly improve the computer's user interface.

22.2 PostScript's philosophy

PostScript has a deep rooted and consistent philosophy that makes a fundamental difference to the way in which programming is carried out, as compared with most procedural high level languages.

PostScript is a programming language specifically designed for page description. Text can be integrated with graphics in the form of drawings or photographs. The language was originally developed in order that two processes, an application program generating graphical output and an interpreter running on a printer, could communicate by sending a *program* describing the required graphics (see Fig. 22.1). This was a significant deviation from the conventional communication with a printer where data, interspersed with a restricted and fixed set of control codes and sequences provided the graphical description. This mode of operation offers a number of advantages in that the information is highly compressed, new functions may be coded in PostScript

* Department of Electronics and Computer Science,
University of Southampton,
Southampton S09 5NH

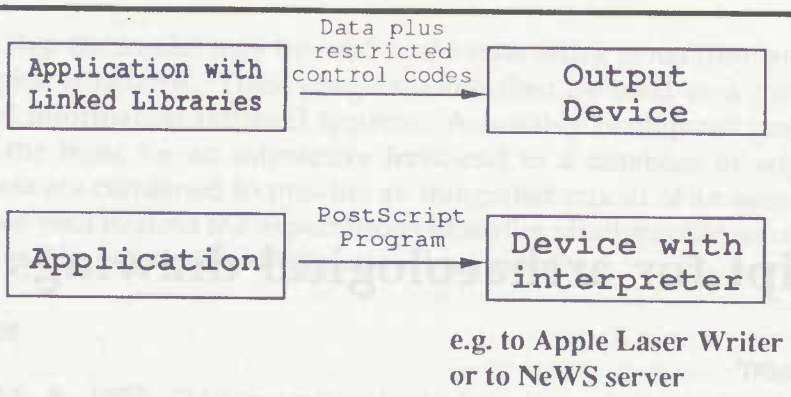

Application with Linked Libraries
Data plus restricted control codes
Output Device
Application
PostScript Program
Device with interpreter
e.g. to Apple Laser Writer or to NeWS server

Figure 22.1: Diagram to show the control of a PostScript interpreted graphics device by sending a PostScript program rather than a restricted device oriented set of geometrical data.

and the programmer need not consider the device characteristics when describing the page layout (this is left to the PostScript interpreter). This contributes to the *device independence* characteristic of the PostScript language.

22.3 The general user view of PostScript

In many cases a user will not be aware that it is PostScript that is generating his or her graphics. In general one uses an application which generates the PostScript program for rendering a picture or script. However, the great flexibility of systems that use PostScript output will be noticed. Many professional typesetting packages and most of the smaller ones generate PostScript output. It has become an industry standard for typesetting. It is usually possible, with little or no modification, to include the output of one application with that of another. For example, the illustrations in this paper were produced with Adobe Illustrator using an Apple Macintosh computer. The PostScript generated by this package was transferred to a Sun and previewed using NeWS. The illustrations were then included directly into the output from the LaTeX text processing package and printed on an Apple LaserWriter. This is just one example of the integration possible with the many different machines and application programs that understand and produce PostScript.

22.4 The PostScript imaging model

The model for imaging that the PostScript language adopts considers an image to be built up on a page by placing ink in selected areas in *stencil and paint* sequences (see Fig. 22.2). All ink is opaque, so placing white ink on top of black results in a white area. The ink can be any shade of gray or any colour. It can form text, outlines, filled shapes or half tone representations of images. PostScript achieves device independence through its imaging model. Three central concepts of the imaging model are the *current page*, the *current path* and the *clipping path*.

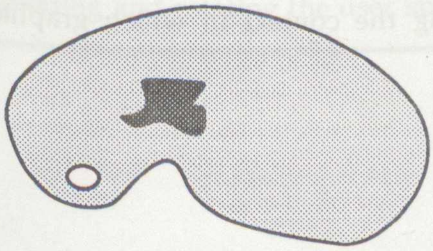


Figure 22.2: Stencilling and painting : the path of the large dark area has been painted as an outline and an area; ink used on subsequent paths has obscured that used on the large area.

The current page is independent of the output device. It is initially empty. PostScript painting operators place marks on the current page. Each mark may be black, white or a specified shade of grey or a colour. Whatever the nature of the mark it is created as though it is opaque, *i.e.* a mark removes any previous mark that it occludes.

The current path is a set of connected or disconnected paths that describe unrestricted shapes. The current path may be used to generate lines on a page, filled regions or used as a clipping boundary.

The clipping path is the boundary of the area which may be drawn upon. It will initially be set to correspond with the full dimensions of the device that the graphics are to be rendered on. The clipping path can be set to any user defined path, but can not be expanded. When altering the clipping path the only way to get back to the original is to save it and restore it.

22.4.1 The graphics state

The *graphics state* is a collection of data describing the context in which graphics operators execute. It includes things like :-

- the current grey level or colour,
- the current point,
- the current path,
- the clipping path,
- a *current transformation matrix* that maps user space to device space,
- the halftone screen for rendering grey levels,
- the transfer procedure that maps user gray levels to device gray levels,
- the current font,
- and a number of parameters that control how lines are drawn.

The users coordinate system, or *user space* (see Fig. 22.3), is independent of the device coordinate system and may be transformed by translation rotation and scaling. The current transformation matrix maps the user space to *device space*, (see Fig. 22.4). It is easy to modify the shape and position of a graphical object using any combination of these operations. The graphics state, which includes the current user space origin, orientation and scaling may be saved before making any changes and restored once the graphical object has been generated.

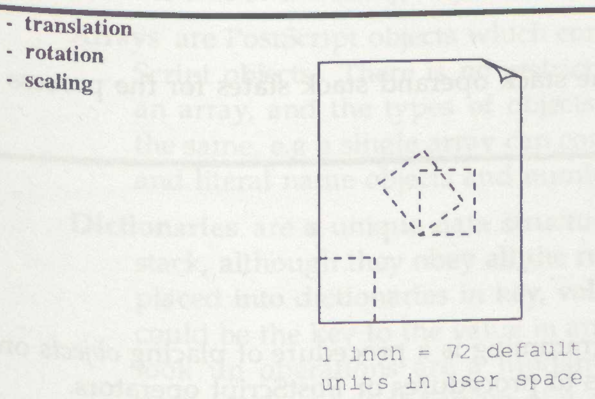


Figure 22.3: The default user space, and 3 other user space states resulting from scaling, translating and rotating the user space

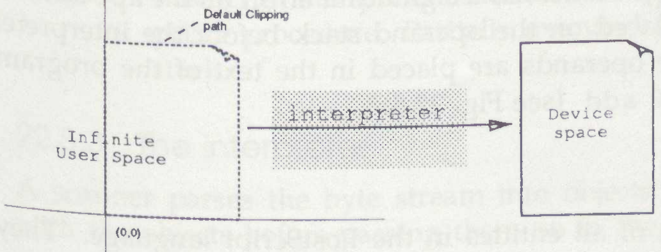


Figure 22.4: The interpreter maps the state of user space to the device space

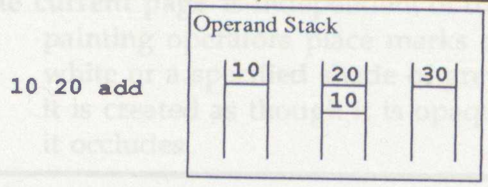


Figure 22.5: The PostScript code and the stack operand stack states for the process of adding two integers

22.5 The language model

22.5.1 Stacks

PostScript is a *stack based language.* Programming is a procedure of placing *objects* onto stacks for later retrieval or as arguments to procedures or PostScript operators.

There are four stacks :-

- the operand stack, which is used for storage of data objects that are to be used by PostScript operators;
- the dictionary stack, which contains only dictionary objects, and is used for name lookup operations;
- the execution stack, which holds objects that are currently being executed. Procedure bodies may be placed on the execution stack temporarily until they have been completely executed;
- and the graphics stack, which is used solely for saving and restoring the graphics state.

The stack based nature of the language model has a significant effect on the appearance of programs. Operands must be placed on the operand stack before the interpreter encounters the operator. Hence the operands are placed in the text of the program before the operator name, e.g. `10 20 add` (see Fig. 22.5).

22.5.2 Objects

Objects are a unified representation of all entities in the PostScript language. They are of fixed size, typically 8 bytes. This simplifies stack manipulation greatly. Objects have sets of attributes, e.g. an object will either be executable or literal. The distinction becomes important when the interpreter encounters a name object. If the object is literal then the interpreter will treat it as data and pop it on the operand stack. If however it is executable the interpreter will execute it immediately. A literal name is introduced by a slash '/' character, e.g. /PaintBox.

Simple Objects are objects which fit into the 8 bytes, e.g. integer, real, boolean, name etc.

Composite objects store pointers to their values, which are held elsewhere. These are strings, arrays and dictionaries.

Strings are simply sequences of characters which are usually created by placing the text of the string in parentheses, e.g. *(A string of text ...)*.

Arrays are PostScript objects which contain a one dimensional sequence of PostScript objects. There is no restriction on what type of objects are stored in an array, and the types of objects within an individual array need not be the same, e.g a single array can contain other arrays, dictionaries, executable and literal name objects and numbers.

Dictionaries are a unique data structure that generally reside on the dictionary stack, although they obey all the rules for other composite objects. Items are placed into dictionaries in key, value pairs, e.g. the name object *x_dimension* could be the key to the value in an integer object containing 100. Dictionary look up operations are a fundamental aspect of the PostScript language, particularly when the value of a pair is an executable array. When a look up operation results in placing an executable array on the operand stack, the contents of that array are executed. By default, two dictionaries exist (*systemdict* and *userdict*) that cannot be removed from the dictionary stack. The system dictionary which resides at the bottom of the dictionary stack contains all the name value pairs of PostScript operators. The user dictionary sits on top of the system dictionary and is available for the user to place key value pairs in it. Other dictionaries can be created and placed on top of the dictionary stack. The top dictionary is known as the current dictionary. Dictionary look ups are performed from the top of the dictionary stack downwards, thus allowing a simple mechanism for redefinition of PostScript operators.

Procedures within PostScript are simply array which have the executable attribute. Procedures are defined by placing executable arrays containing PostScript code into a dictionary. When a procedure name is encountered the PostScript interpreter searches a dictionary stack in order to find the appropriate procedure. Partially executed procedures are place on the execution stack.

22.5.3 The interpreter

A scanner parses the byte stream into objects and associates appropriate attributes with the objects before passing them on to the interpreter. For example the scanner recognises string objects because they are enclosed in brackets, e.g. `(Hello World)`. A point to note here is that the scanner is responsible for detecting syntax errors, and the only syntax error that can be generated in PostScript is unbalanced parentheses around a string object. All other errors are semantic.

Any token that consists entirely of regular characters and that cannot be interpreted as a number is treated as a *name object*. The name object is assigned the *executable attribute* unless it is preceded by a '/', when it is given the literal attribute. The interpreter takes each object and executes it immediately. Execution may simply mean placing the object on the operand stack, as is the case with say an integer. Most PostScript operators work by taking data from the operand stack and/or the current graphics state, and leaving a result on the operand stack or changing the graphics state.

22.6 Operators

22.6.1 Path operators

The operators which allow paths to be specified are `newpath`, `moveto`, `lineto`, `rlineto`, `curveto`, `arc`, `closepath` and many others. The `curveto` operator allows the specification of Bézier cubic splines. Many of the path extension operators use the current point as part of their arguments. Fig. 22.6 shows a simple example of a PostScript program which draws a box with some text in it. The text facilities of PostScript are based around families of fonts. Each font is a dictionary object for which the keys indirectly point to a path description that describes the outline of a given character or symbol. In general these paths are used for filling with the current grey level or colour, but the paths of text can be used as a clipping path, or stroked, just as with a path generated using the previous path creation operators. A character path is added to the current path with the `charpath` operator. Each character has metric information associated with it allowing the calculation of coordinates for placing a sequence of characters on a page.

22.6.2 Painting operators

The current path may be made visible with the `stroke` or `fill` operators. Strings may be made visible with the `show` operator. The show operator is effectively a path and painting operator. A path is generated from the string on the operand stack using the current font which has been previously scaled to the appropriate size. The path generated from the string is then painted by filling it with the current grey level.

To generate a bit map image the `image` operator is used. This operator also takes arguments that define the nature of the painting to be done before performing the painting operation. The device that the graphics are to be rendered on often consists of a rectangular array of picture elements or *pixels* which may be either black or white. The generation of grey level images must therefore be performed by using halftoning techniques. The user need not be concerned with the generation of the *halftone screen*, which defines the way in which grey levels are built up from dot patterns, although it is possible to specify the screen with the `setscreen` operator. There is a default halftone screen for each device which is optimum for the characteristics of the device.

The user must supply a procedure as an argument to the `image` operator which generates a string that contains the bit map image's data. This procedure is usually something like the PostScript `readhexstring` operator followed by ascii text representing hexadecimal numbers corresponding to the images bit map.

22.6.3 Graphics state operators

A number of operators exist for modifying or enquiring about the graphics state. For example, the operators `setgray` and `currentgray` set and retrieve the current gray level used for painting. Equivalently, `sethsbcolor` and `setrgbcolor` set the current colour for painting using either a hue, saturation, brightness or red, green, blue

```
%!PS-Adobe                       % standard PostScript program header

% define a procedure to create a rectangle path

/box {                           % width height x y box => rectangle path
   moveto                        % move to x y coordinates on top of stack
   /height exch def              % put height with top stack object in current dictionary
   /width exch def               % put width in current dictionary
   0 height rlineto              % create path of left side of box
   width height rlineto          % create path of top of box
   width 0 rlineto               % create path of right side of box
   closepath                     % close the path to finish the box
} bind def                       % place 'box' and the procedure in the current dictionary

% define a procedure to stroke a rectangle path at specified position and orientation

/stroke-box {                    % width height x y angle x_u y_u stroke-box => box stroked
   gsave                         % save the graphics state
      translate                  % translate user space to x_u y_u on stack
      rotate                     % rotate user space by angle on stack
      box                        % invoke box procedure
      stroke                     % stroke the current path
   grestore                      % restore the graphics state
} bind def                       % place 'stroke-box' procedure in dictionary

% Begin the main program

0.8 setgray                      % set the current gray level to 0.8
100 50 0 0                       % put width height x and y on stack
45 100 100                       % put angle and space origin on stack
stroke-box                       % create a box path with 'box' procedure
/Times-Roman findfont            % put the Times-Roman font on the operand stack
12 scalefont                     % scale the font on the stack to 12 point
setfont                          % set the current font to the one on the stack
20 20 moveto                     % move the current position to coordinates 20 20
(Hello)                          % put the string object 'Hello' on the stack
show                             % fill the boundary formed by the path
                                 % of the string on top of the stack
                                 % using the current font
                                 % with the current gray level
                                 % at the current position

showpage                         % put graphics onto device output
```

Figure 22.6: A simple PostScript program

colour space. Operators exist for setting and investigating all the parameters listed in section 22.4.1.

22.6.4 Other operators

There are over 250 PostScript operators built in of which a small set are used very frequently. Here are a few of the common ones.

- Program control operators such as `loop`, `repeat`, `for`, `forall`, `if`, `ifelse`, `exit`.
- Stack manipulation operators, such as `dup`, `pop`, `exch`.
- Maths operators such as `add`, `sub`, `neg`, `mul`, `div`, `abs`, `round`, `sqrt`.
- Dictionary operators such as `dict`, `begin`, `end`, `store`, `get`, `put`, `currentdict`.
- String Operators such as `string`, `forall`, `search`, `stringwidth`.
- Font operators such as `findfont`, `scalefont`, `setfont`.
- Optimisation operators such as `bind`.

22.7 NeWS: A Networked extensible Windowing System

NeWS is a PostScript based graphics environment, currently implemented on Sun computer workstations. The *NeWS server* supervises the execution of PostScript programs, many at a time, and *places ink* on the Sun's high resolution monitor. NeWS's PostScript contains a number of extensions in order to facilitate the creation of windows on a computer screen and interaction with those windows using the computer's keyboard and mouse.

Graphics are produced in NeWS by PostScript programs running under supervision of the NeWS server. The NeWS server itself knows no more about graphics than does the PostScript interpreter on a printer. However, a number of default PostScript files are executed by the NeWS server when the server is initialised. These files, amongst other things, define default characteristics for a windowing environment.

Client programs, written in some other high level language, communicate with the NeWS server using a serial byte stream, (see Fig. 22.7). The philosophy of the communication between the interpreter and the client program using only ASCII text is preserved in NeWS. However, the facility to use a compressed byte stream, or a mixture of compressed and uncompressed is offered, in order to enhance the speed of execution. Keyboard and mouse input is detected by the PostScript programs and can be passed on to the client program, or the PostScript program itself can react to the input.

The screen can be used as a single *canvas* and graphics can be generated on the screen in the same way as using a printer. Alternatively *offspring* canvases can be created. These can be built up into windows and can be made to react to events, such as the depression of a mouse button.

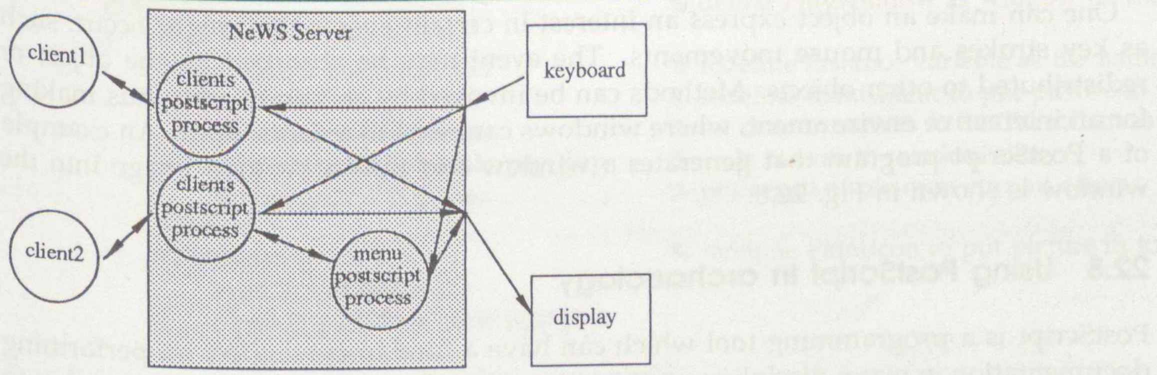


Figure 22.7: NeWS operates using a client / server model; Client programs send PostScript programs to the NeWS server and communicate with those programs using byte streams. User input from the keyboard and screen may be handled by the PostScript program or passed to the client program.

The windowing aspects of NeWS are implemented in an *object oriented* scheme. The types of objects that are usually dealt with in NeWS are windows, buttons and sliders. Buttons and sliders are used for simple purposes such as invoking operations or setting numerical values.

One can define classes of objects that have default *instance variables* and *methods*. The instance variables are static variables that describe attributes of the state of an object. Methods, are the procedures that an object can carry out when asked to do so. The class by itself can do nothing except be used to generate instances of itself. An instance of a class, *i.e.* an object, inherits the instance variables and methods of the class.

An instance of a class is generated in NeWS by sending the message */new* to the class, along with any appropriate arguments that the class might require to generate this instance. NeWS is supplied with a number of ready made classes for generating windows, panel buttons, sliders, switches, icons etc. All that has to be done by the user then is to redefine certain instance variables and methods in order to tailor a generic object to one that has a desired behaviour. These redefinitions can be sent to the object as a message.

One can make an object express an interest in certain events that might occur, such as key strokes and mouse movements. The event may be absorbed by the object or redistributed to other objects. Methods can be invoked by certain events, thus making for an interactive environment, where windows can react to a users input. An example of a PostScript program that generates a window and puts a bit map image into the window is shown in Fig. 22.8.

22.8 Using PostScript in archaeology

PostScript is a programming tool which can have a fundamental effect on performing documentation in many disciplines, not just in archaeology. However, our research into image and graphical databases is ideally suited to the use of PostScript in a number of different ways. PostScript provides for an integrated system which would otherwise not be achievable. Fig. 22.9 shows the computing environment for research into an archaeological graphical database at the University of Southampton.

Retrieval of information from a text based interface to an archaeological database is far less rewarding than an interface which can provide additional images to peruse. These databases are becoming more common, but are often constrained in their usage by the hardware used for image display. Often, only a single image can be displayed at a time, the image is displayed at fixed size and cannot be modified in any way. NeWS provides an environment in which the possibilities for interacting with multiple images are unlimited. A *snapshot* from a database enquiry session is shown in Fig. 22.10, depicting a window in which the database management system is running, images of pots in open and iconic states, a distribution map and a pie chart.

Once an image has been generated in a window it can be manipulated by the process that generated it (*i.e.* the database management system), or by the user. The interactions that the user can make are defined by the methods of that particular instance of the window class. Resizing, moving and iconifying are some of the default methods of the window class, thus allowing the user to put aside the results of a database retrieval for later comparison with other images. One could define methods that allow measurements to be taken from single images, or stereo pairs, or to perform

```
% create picture painting procedure
/pic {                                       % pic => canvas
    thepicture readcanvas
    pause
} def

% create main program procedure
/main {
    /mywindow                                % literal as key for window
        framebuffer                          % full screen canvas is argument to /new
        /new DefaultWindow send              % send message /new to class 'DefaultWindow'
    def                                      % define /mywindow as window on the stack
    {
        /FrameLabel thepicture def           % redefine instance variable as file name string
        /PaintClient {                       % redefine PaintClient to put picture in window
            ClientCanvas setcanvas           % set current canvas to be ClientCanvas
            clippath pathbbox scale pop pop  % scale canvas to window size
            pic imagecanvas pause            % put result of pic onto current canvas
        } def
        /PaintIcon {                         % redefine PaintIcon to put picture in icon
            IconCanvas setcanvas
            clippath pathbbox scale pop pop
            0 0 moveto
            pic imagecanvas IconCanvas setcanvas
            0 strokecanvas
        } def
    } mywindow send                          % send instance variables and
                                             % methods to mywindow

    /reshapefromuser mywindow send           % send reshape message to mywindow
                                             %    this invokes a method to stretch
                                             %    a rectangle on the screen
    /map mywindow send                       % make the window visible

} def                                        % place 'main' procedure in current dictionary

/thepicture (sunrise.im8) def                % define the file name string
main                                         % invoke the main procedure
```

Figure 22.8: A PostScript program which uses the NeWS extensions to PostScript in order to place an image in a window on the Sun's screen

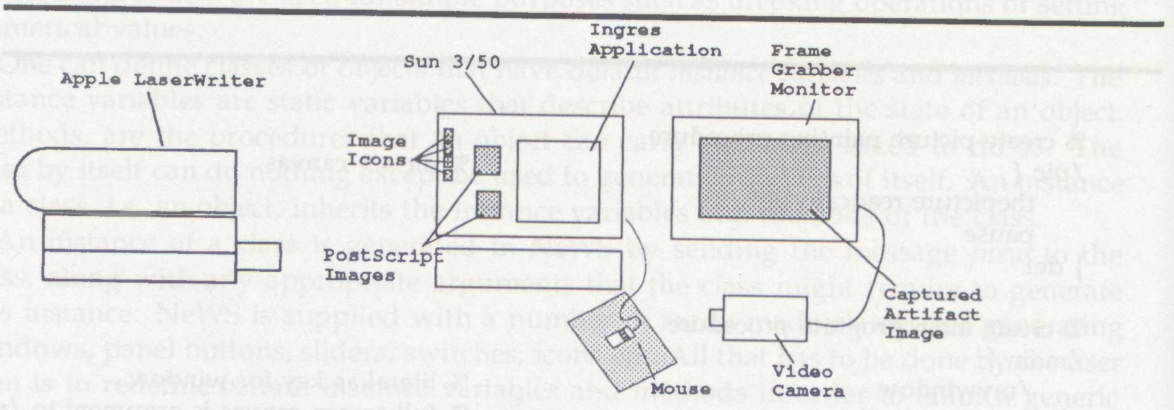
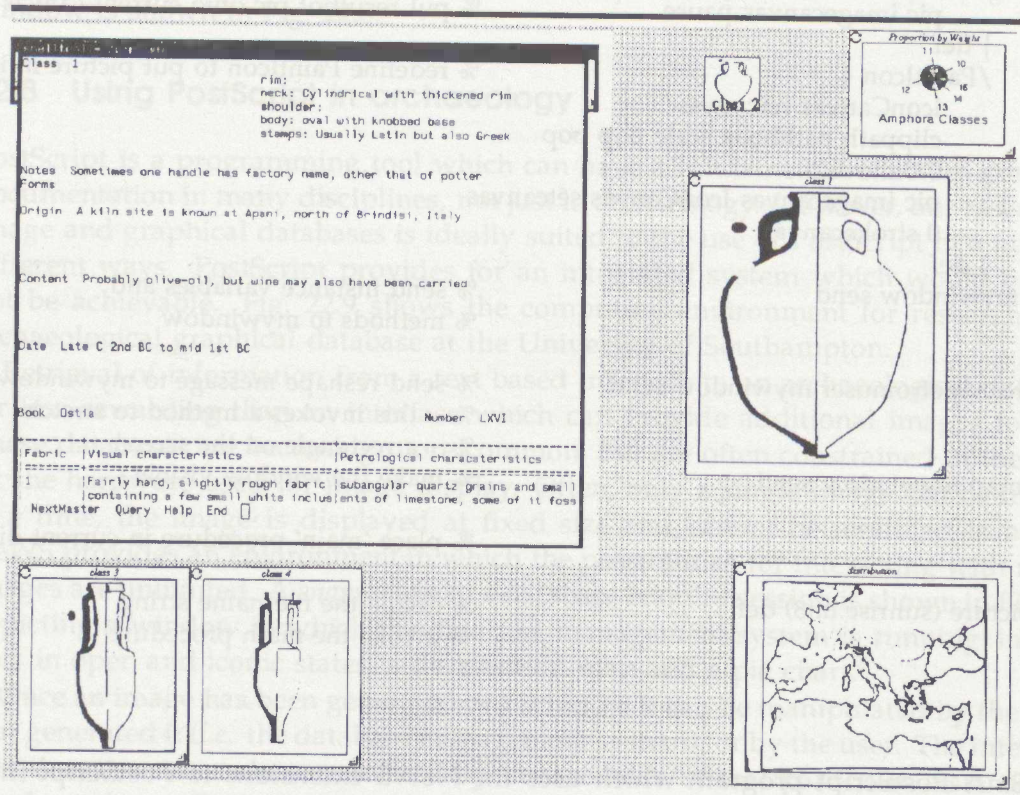


Figure 22.9: The central equipment for the archaeological database research



```
Class 1
                                  rim:
                                  neck: Cylindrical with thickened rim
                                  shoulder:
                                  body: oval with knobbed base
                                  stamps: Usually Latin but also Greek

Notes  Sometimes one handle has factory name, other that of potter
Forms

Origin  A kiln site is known at Apani, north of Brindisi, Italy


Content  Probably olive-oil, but wine may also have been carried


Date  Late C 2nd BC to mid 1st BC


Book  Ostia                                            Number  LXVI

+--------+----------------------------------+--------------------------------
|Fabric  |Visual characteristics            |Petrological characateristics
|========+==================================+================================
|1       |Fairly hard, slightly rough fabric |subangular quartz grains and small
|        |containing a few small white inclus|ents of limestone, some of it foss
 NextMaster  Query  Help  End  []
```

Figure 22.10: The contents of the Sun's screen during a database enquiry session

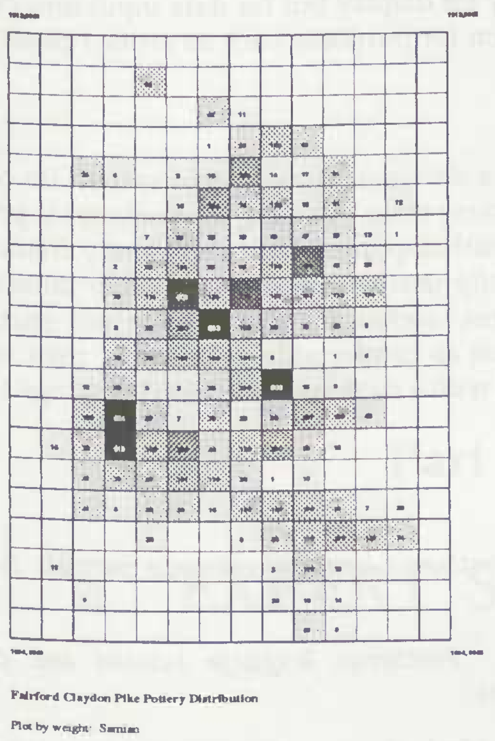


Figure 22.11: An plot of geographical dispersion automatically generated from a database retrieval.

image processing for producing shape descriptions, or with sensitive areas for revealing further information when the mouse is used to select the area, etc. etc.

Graphical display need not be limited to the database's graphical data. A simple PostScript program can filter numeric database retrievals to produce pie charts or bar graphs or whatever form of graphical output may be desired. Geographic information can be translated into picture form, perhaps with quantitative data superimposed (see Fig. 22.11).

These aspects of the use of PostScript all relate to the use of PostScript from within a database environment using a computer monitor as the device for display. The database environment effectively becomes all the books and catalogues that an archaeologist needs, with all the flexibility that books on a desk top offer, and more. However, there are times when it is not possible to consult the electronic database, and books are needed. This is the time when a paper copy of the database contents is needed. For this situation a database application program can be generated that uses PostScript, or a typesetting package that generates PostScript output, to produce a typeset version of the contents of the database. The paper version would include all the relevant graphical information, with suitable cross referencing and indexing.

The path description capabilities of PostScript may also offer a convenient method for storing artifact shape information not only for display but for data input into processes which use or analyse the shape information for purposes such as artifact classification.

22.9 Conclusions

In conclusion, PostScript, integrated with a database management system (in our case, Ingres) and an video frame grabber and frame store (Imaging Technology) is providing a powerful computing environment at Southampton, with a sufficiently friendly user interface that allows for an archaeologically useful tool. With suitably tailored front ends to the database management system, archaeologists that are not particularly interested in computers *per se* will feel just as comfortable performing their research, cataloguing, or publication as they would with a desk full of books, drawings and note pads.

References

ADOBE SYSTEMS INCORPORATED 1985. *PostScript language reference manual.* Addison-Wesley, Reading, Massachusetts.

ADOBE SYSTEMS INCORPORATED 1987. *PostScript language tutorial and cookbook.* Addison-Wesley, Reading, Massachusetts.

ADOBE SYSTEMS INCORPORATED 1988. *PostScript language program and design.* Addison-Wesley, Reading, Massachusetts.