First Impressions of NeWS

W. Roberts, M. Slater, K. Drake, A. Simmins, A. Davison*
and
P. Williams†

Abstract

The SUN Microsystems Network Extensible Window System (NeWS) is a display system that uses the PostScript page description language to provide fully device independent screen and input manipulation as a network service. This system has been available in the UK as a beta test product since April 1987 and this paper reports the work done with NeWS at Queen Mary College. A general introduction to NeWS is given, together with some more detailed information and reports of what each author has been doing with the system. We will try to predict the future of NeWS and its main rival (the MIT X-Windows system) and to suggest changes to the NeWS beta product that will be necessary for it to realise its potential.

Keywords: NeWS, PostScript, workstation, window systems, device independence, graphics, X-windows

1. Introduction

The NeWS system is fully described in the NeWS Technical Overview document available from SUN Microsystems[1] so this paper will not attempt an in-depth description of NeWS. Section 1 will provide an outline of the system and the way it achieves its aims of device independence and the efficient provision of a network service (i.e. the application program can transparently drive the display on different machines somewhere on the network). Section 2 examines in more detail the PostScript programming style used in NeWS and some of the background necessary for Sections 3-7 which contain individual reports of the work done with NeWS by each author and their reactions to the system. Section 8 explains where we feel the NeWS system is missing its full potential and how we would like to see it modified. Finally, Section 9 measures NeWS up against X (see Scheifler and Gettys[2]) and tries to predict the future.

2. What is NeWS?

News is a Network Window System: it allows the display control aspects of an application to be isolated as a separate process, possibly on another machine elsewhere on the network. It is also a device-independent system: the separation of display control from application allows the application to talk in a standard way to the display control process and defers all the issues of keyboard mapping, screen resolution etc to the display control process. This means that implementing the display control process is all that is involved in porting the application to a new type of workstation (for example).

This is in many ways similar to earlier standardisation attempts such as GKS,[3] but with the network element added. This too is not new; the Andrew window manager[4] controls its display through a byte stream protocol that could work over a network connection. What IS new is full device independence and a fully programmable interface: both are achieved by using PostScript.

2.1. PostScript

PostScript is a page description language for laser printers that is fully independent of the printer resolution, output area etc. For this reason it is possible to print exactly the same PostScript file on a wide range of printers (from the Apple LaserWriter at 300dpi to the Linotronic 300 at 2540dpi) and get identical output, save that the higher resolution printers give higher quality rendition of lettering, curves etc. This property has made PostScript a de facto standard for page description languages and over 20 different printers are available using this language.

Device independence is achieved by describing the image to be printed as a series of mathematical curves in an arbitrary user coordinate space (with real coordinates) and then filling areas bounded by such curves, or *stroking* the curve with a line of a given thickness. PostScript describes its fonts in these terms and so text may be scaled arbitrarily and subjected to arbitrary

* Department of Computer Science
Queen Mary College
190 Mile End Road
LONDON E1 4NS

† QMC Interactive Systems Ltd
229 Mile End Road
LONDON E1 4NS

E-Mail enquiries should be addressed to
liam@cs.qmc.ac.uk

North-Holland
Computer Graphics Forum 7 (1988) 39-57

application when the interaction is completed.

2.2. NeWS Extensions to PostScript

As well as the addition of lightweight processes to the standard PostScript language,[5] NeWS adds a number of primitives necessary for managing an interactive display. The three major additions are processes, events and canvasses.

- **Processes:**
  The NeWS interpreter provides lightweight processes with non-preemptive scheduling. New processes are produced by using a fork primitive and the parent process receives an object of type process which can be used to collect an arbitrary PostScript object as exit value from the child. The child has a copy-on-write version of its parent's stacks, so they can communicate through any PostScript object to which they both have a reference. To allow for orderly access to shared objects, NeWS adds monitors for process synchronisation and protection. To create private memory, a process simply creates a new dictionary and pushes it onto its dictionary stack; there are no other references to this new dictionary so no-one else can access it. This allows an unusually high degree of flexibility in memory sharing policies, ranging from global shared memory to local private memory.

- **Events:**
  The most flexible way of handling a multi-window screen with mouse, keyboard etc. seems to be by describing all changes as *events* and leaving the window manager and application to decide on a course of action when the mouse leaves/enters a window, buttons are released etc. NeWS provides such an event mechanism and also allows arbitrary user-defined events, which can carry arbitrary PostScript objects. PostScript processes can provide event translation for the rest of the NeWS system by consuming events and issuing appropriate new events; this allows processes to listen for MenuButton events whilst someone else deals with which input(s) are to be so translated. Any process can opt to listen to the raw events and do its own translation.

- **Canvasses:**
  PostScript contains no undrawing facilities; the most you can do is draw over something in a different colour or shade of grey. To handle traditional window system features such as movable panels, NeWS extends PostScript from a single framebuffer to add multiple independent drawing surfaces called *canvasses*. All drawing activity is carried out in terms of a current canvas, which can be an arbitrary shape (even with holes in it) and each PostScript process has a separate graphics state. Canvasses exist in a hierarchy which determines the order in which they receive events and appear above the background canvas. Canvasses are also the general means of screen control; input events can be directed at specific canvasses and the NeWS interpreter generates events whenever the mouse enters or leaves the inside of a canvas. By using a *transparent* canvas, an application can arrange to receive events that happen within some arbitrarily complex region, but without having to use up memory holding display data or obscuring an active part of the display underneath. A further special form of canvas, called an *overlay canvas*, is provided specifically for fast interactive feedback: an overlay canvas is associated with a real canvas and anything drawn in the overlay obscures the underlying canvas. However, the overlay may be erased to reveal whatever was underneath it undamaged, and this is used to provide rubber banding and similar forms of interactive feedback. By using a suitable PostScript routine in the NeWS server, an application can start an interactive selection and leave the server to handle all the intermediate drawing etc. Final results need only be returned to the application when the interaction is complete, a considerable saving in network traffic.

There are 102 additional primitives in the NeWS version of PostScript (which has 253 operators), but the majority are concerned with the key extensions described above. NeWS also provides a library of useful routines and a demonstration window manager, which will be described in more detail in the next section.

The authors feel that NeWS is a more pleasant graphics environment than others with which they are familiar*, and that PostScript is inherently very usable, working with, rather than against, the graphics programmer.

2.2.1. Programming the NeWS Server

The libraries provided with NeWS include a number of convenient routines for producing menu-style selections, rubber-band style cursor-following and general programming support. The most important of these is the PostScript code which provides an object-oriented

*Including SunView, GKS, SunCore, SunCGI, Microsoft Windows, the Whitechapel Angel and Oriel systems, and Smalltalk-80.

programming interface with objects, methods, classes and inheritance. An object is represented by a PostScript dictionary containing its instance variables (and instance methods) and a reference to the dictionary defining its class (and class variables). The PostScript dictionary stack provides a natural way of implementing simple inheritance; the superclasses are pushed onto the dictionary stack before the subclass, followed by the instance dictionary, and the PostScript name lookup scheme ensures that names with multiple definitions refer to the definition lowest down the class hierarchy. Classes are defined using routines called classbegin and classend, and messages (consisting of the PostScript procedure name and any arguments required) are sent to an object using a routine called send. The message /new when sent to a class object will create a new instance of the class. This fits very naturally into a PostScript way of working and is provided by routines written completely in standard PostScript. This implementation of classes allows dynamic reprogramming of instances; the message can be any executable PostScript object and can therefore create new bindings of names in the instance. The context in which it is executed has as top level dictionary the one containing the instance variables, so the new names are normally keys in this instance dictionary. It would be possible to implement a browser-style interface to such classes, similar to that of SmallTalk.

Kieron Drake has carried out some work with the class routines, creating classes for objects such as points and splines, complete with methods for interaction. More details are given in Section 4.

2.3. The LiteWindow Window Class

The NeWS beta distribution includes PostScript code for a number of class definitions, of which the most important are the LiteMenu and LiteWindow classes. The prefix Lite- is intended as a strong hint that these are toy implementations and will soon be thrown away; however, the longer they exist the more likely they are to become the standard NeWS menu and the standard NeWS window respectively. The LiteWindow class will be discussed in some detail as it is referred to in later Sections.

As mentioned above, LiteWindow is the default window implementation in the NeWS distribution and as such is used by all the demonstration programs. To be completely accurate, these all refer to the DefaultWindow class, so it is easy to substitute something else, but LiteWindow is the default class associated with the name DefaultWindow.

2.3.1. Outline of LiteWindow Features

LiteWindow provides a straightforward window object similar in flavour to a basic SunView[6] *frame* with a *canvas sub-window*. It consists of a rectangular window with a mouse-sensitive frame, and a rectangular icon which is only visible when the window is closed. The area within the frame is called the *Client canvas* and is largely the responsibility of the application controlling the window. The cursor appearance changes according to its position in the window or window frame. Clicking in the open/close corner of the frame causes the window to flip from open to iconic and vice versa. Clicking the mouse in the rest of the frame produces a pop-up menu of options such as Reshape, Redisplay, Move, Top, Bottom and Zap. The application provides specific PaintClient and ClientMenu code to handle repainting the Client canvas and responding to mouse clicks within the Client canvas; these are stored in the instance dictionary of a LiteWindow object. All of this is highly adjustable, the simplest method being to define a subclass of LiteWindow which re-implements whatever you would like done differently.

William Roberts and Kieron Drake have done work on subclassing LiteWindow to provide windows with various properties such as fixed aspect ratio (See Section 3.1) and interaction with collections of client objects (see Section 4.3).

2.3.2. Details of Class LiteWindow

Adopting the Smalltalk convention of grouping methods into *categories* (the NeWS class implementation doesn't do this because it is not organised for interactive incremental modification), the public methods of LiteWindow are as shown in Figure 1.
There are also a number of private routines which are used internally and should not be sent by users of the object.

Figure 2 lists the LiteWindow class variables: These are common to all instances and define the characteristics of the windows which are unlikely to be changed on an individual basis*. The interpreted nature of NeWS means that changes to these class variables will have an immediate effect on all the instances of class LiteWindow, though these may not show up until the windows are repainted.

The instances of LiteWindow also have a number of private instance variables (Figure 3), the most important being PaintClient which is the application specific routine to draw the contents of the window.

*Note however that the ParentCanvas class variable is in fact one of the arguments to the /new message and so is altered every time an instance is created! This is fundamentally wrong.

Category	Method	Comment
Initialize-Release	new	Create window and record parent canvas
	destroy	Get rid of canvasses and child processes
Appearance	map	Make visible and enable frame input
	unmap	Make invisible and disable frame input
	reshape	Change window boundary
	flipiconic	Alternate between open/iconic
	move	Set window position relative to parent canvas
	totop	Push window/icon above its siblings
	tobottom	Push window/icon beneath its siblings
Painting	paint	Get the window/icon repainted
	painticon	Repaint icon
	paintframe	Repaint window frame
	paint framelabel	Repaint frame label
	paintclient	Setup and execute PaintClient routine
Interaction	reshapefromuser	Choose window boundary
	slide	Drag window
	slideconstrained	Drag window horizontally or vertically
	stretchcorner	Change boundary by dragging a corner
	stretchwindowedge	Change boundary by dragging an edge

Figure 1. Public Methods in Class LiteWindow

Category	Name	Comment
Frame	BorderLeft	Width of left edge of frame
	BorderRight	
	BorderTop	Width of top edge of frame
	BorderBottom	
	FrameBorderColor	Colour used to draw the frame
	FrameFillColor	Colour used to fill the frame
	FrameMenu	Menu object used with the frame
Icon	IconWidth	Dimensions of icon
	IconHeight	
	IconBorderColor	Colour for drawing the icon edge
	IconFillColour	Colour for filling the icon
	IconMenu	Menu object used with icon
Label	FrameFont	Font used for frame label
	FrameTextColor	Colour used for frame label
	IconFont	Font used for icon label
	IconTextColor	Colour used for icon label
Client	ClientMinWidth	Smallest allowed window dimensions
	ClientMinHeight	
	ClientFillColor	Fill colour used before repainting Client canvas
	ForkPaintClient?	Run PaintClient as a child process
Other	ParentCanvas	Background canvas for these windows
	KeyFocusColor	Colour used to indicate active window
	ZoomSteps	Controls window/icon transformation

Figure 2. LiteWindow class variables

Category	Name	Comment
Frame	FrameCanvas	Drawing surface for frame
	FrameX	Window position on parent canvas
	FrameY	
	FrameWidth	Window dimensions
	FrameHeight	
Icon	IconCanvas	Drawing surface for icon
	IconX	Position of icon on parent canvas
	IconY	
	IconImage	Optional contents of IconCanvas?
Text	FrameLabel	
	IconLabel	
Client	ClientCanvas	Drawing surface for contents of window
	PaintClient	Application routine for redrawing window
	ClientMenu	Optional pop-up menu active in ClientCanvas
	FixClient?	Contents of window partially damaged
Frame Events	FrameInterests	Events to which frame is sensitive
	FrameEventMgr	Process listening for frame events
	KeyFocus?	Window is focus for keyboard events
	KeyFocus=	(Working variable)
Other	PaintProcess	Child process doing window repainting
	Iconic?	Boolean indicating window is closed

Figure 3. Instance Variables of an object in class LiteWindow

2.3.3. Worked Example of Using LiteWindow

As an example of using the LiteWindow class, consider the problem of opening a window and drawing a black circle centred in the window and with diameter ¾ of the shortest side.

To tackle this in NeWS, we first create an instance of the window class:

```
currentcanvas /new LiteWindow send
/demowindow exch def
```

For convenience, we associated the new object with the name demowindow in the current dictionary. Next we equip the window with the procedure for drawing the desired image, split into two procedures for clarity; the procedure SetCoordinates will arrange a standard coordinate system and DrawCircle will carry out the drawing, assuming the desired coordinate system. This is a typical method of working in PostScript; it is often easier to change the coordinate system underneath than to change the object on top, especially where simple linear transformations are involved.

The SetCoordinates routine needs some explanation. The first two lines calculate the true size of the

```
{
    /CircleGray 0 def

    /DrawCircle {
        gsave
            0.75 0 moveto
            0 0 0.75 0 360 arc
            CircleGray setgray fill
        grestore
    } def

    /SetCoordinates {
        FrameWidth BorderLeft sub
        BorderRight sub
        FrameHeight BorderTop sub
        BorderBottom sub
        % The stack now holds
        % ClientWidth ClientHeight
        0.5 dup scale
        2 copy translate
        min        % A NeWS primitive
        dup scale
    } def

    /PaintClient {
        gsave
            SetCoordinates
            DrawCircle
        grestore
    } def
} demowindow send
```

ClientCanvas by compensating for the width of the frame. The origin is by default the bottom left corner of the canvas, and so changing the scale of the user coordinates by a factor of ½ will double the size (in user coordinates) of the canvas. The subsequent change of origin, moving right by the width and up by the height, therefore puts (0,0) at the centre of the window. The use of the copy primitive has saved additional copies of the height and width values (so they don't need to be recalculated) and the min primitive supplied by NeWS is used to find the smaller of the two values. This value is then used to change the scale once more, making the shorter side of the canvas 2 units long in the final user coordinate system. The DrawCircle routine can then simply draw a circle of radius ¾ centred on the origin; the other routine has carried out all the elaborate coordinate manipulation.* The shade of gray used to fill the circle is determined by a new instance variable called CircleGray, of which more later.

The instructions to define these three procedures are enclosed in {parentheses} to form an executable array, which is then sent as a method to the instance of LiteWindow created earlier.

Now the window is ready for use, but it is not yet displayed on the screen, nor has its size been chosen. These things can be accomplished by using the class methods for LiteWindow and setting the size explicitly, or the user can be prompted to give size and position by dragging a suitable rectangle. The latter is accomplished by executing the commands:

```
/map demowindow send % make window visible
/reshapefromuser demowindow send
```

All of the interaction will be handled by the NeWS server in the display; these commands can be coming from any machine on the network. The menus associated with the frame allow the window to be resized interactively in typical window-manager ways, and the PaintClient routine modifies the size and position of the circle accordingly.

To emphasise further the work that the server can do, we will extend the code so that clicking the Menu button in the client canvas will produce a pop-up menu which offers to make the circle darker or lighter. Unfortunately, the LiteWindow implementation requires the ClientMenu value to be setup before the FrameCanvas is created, so the following code must be executed BEFORE the /map and /reshapefromuser methods are executed.

```
{
   /ClientMenu [
      (Darker)
      { -0.1 /ChangeGray MyWindowInstance send }
      (Lighter)
      { 0.1 /ChangeGray MyWindowInstance send }
      ] /new DefaultMenu send
   def

   ClientMenu /MyWindowInstance self put

   /ChangeGray {
      CircleGray add
      0 max 1 min
      /CircleGray exch def
      /paint self send
   } def
} demowindow send
```

The first few lines create a menu with two option lines labelled Darker and Lighter along with the procedure to be executed in each case. The menu will be handled as a separate NeWS process and the environment in which it is executed will not have demowindow on the dictionary stack; if demowindow were left on the stack, the Menu code could mistakenly inherit methods from the LiteWindow class, so the NeWS class code correctly avoids this. To allow the menu to affect the window, we include a reference to the window as a new variable MyWindowInstance in the menu instance: activity such as changing the CircleGray variable in the instance is then done by using this reference to the demowindow object. The ChangeGray code, a new demowindow instance method, prevents the value of CircleGray from going outside the legal range, and in true object-oriented style, sends itself a */paint* message to update the window.

With the window so set-up, clicking the menu button within the window will pop-up the menu described and clicking either option will have the stated effect. The NeWS code provides a LiteItem class which can be used to add sliders and buttons, or the getanimated routine could be used to drag the intensity value, perhaps as a function of the distance between the cursor position and the centre of the circle.

Kieron Drake has worked on using the flexibility of NeWS for user-interface design and prototyping.

Reports of Work using NeWS at QMC

The authors can be contacted at QMC by electronic

*The combination of frequent changes of coordinate system and stack-based execution can seem very intimidating at first; with a little practice it becomes quite straightforward, though better machine support would be useful.

mail using either the ARPA or the UUCP networks. The electronic mail address for QMC is: **qmc-cs** (UUCP address) or **cs.qmc.ac.uk** (ARPA address). The usernames are

liam	William Roberts	mel	Mel Slater
kieron	Kieron Drake	tony	Antony Simmins
allan	Allan Davison	peterw	Peter Williams

3. William Roberts: NeWS as a PostScript Compiler

My job as a member of the programming and systems support staff includes supporting the Departmental printing facilities, both software and hardware, and that in turn involves PostScript because all our printers use PostScript (Apple LaserWriter+). I first started using NeWS to assist in my PostScript programming work (though I always intended to find out more about it for its own sake).

The main problem with developing PostScript programs in the absence of typechecking tools is that the only way to find out if it works is to run it; before NeWS that meant sending it to a printer and wasting lots of paper trying to get it right. One is also reduced to drawing things on graph paper and working out coordinates, which method is a poor input to the sophistication and precision of PostScript.

3.1. Preparatory Work

I first started using NeWS to help in the development of a PostScript program to print a board for the game Diplomacy, which requires an outline map of Europe divided schematically into geographic regions, each labelled in various ways. The attraction of NeWS was the prospect of placing sector boundaries, sector labels etc interactively with feedback directly on the screen and all positions in the working coordinate scheme. The Diplomacy board still isn't finished, but my subsequent work with NeWS has been on adjusting the SUN-supplied PostScript code for windows, interaction etc to facilitate the sort of things necessary for the Diplomacy board development and other similar tasks.

3.1.1. Constrained mouse following

The standard getanimated routine for tracking mouse movements until a mouse click, for example drawing a rubberband line or a framing rectangle, is not convenient for drawing rectangles with a fixed aspect ratio. I produced a version called getvalidated which takes two procedure arguments; the first is a validation/translation procedure which modifies the x,y if desired, for example to change x or y so that x0,y0 and x,y define the smallest A4 aspect rectangle containing the original position, and the second procedure performs any new drawing necessary. The second procedure is only called if the modified x,y values have changed, to avoid unnecessary redrawing. This getvalidated routine also returns an additional value when it exits; this can be calculated by either of the procedures. This routine has been used in the sizing of windows with fixed aspect ratio (see below) and as a demonstration part of a chessgame interface. The constraining routine converts the cursor position into the coordinates of the square on the board and determines whether x0,y0 to x,y is a valid knight move. The drawing procedure puts an edge around the x,y square and draws the knight in the x0,y0 or x,y square according to the validity of the move. This could be used to defer to the user-interface some of the simple checking involved in a chess program.

3.1.2. Window Shapes

The standard LiteWindow window definition allows the user to create and reshape windows as arbitrary rectangles: my work with the printer is more suited to windows with the aspect ratio of a piece of A4 paper, so I produced a subclass of LiteWindow which supports this: the getvalidated routine determines the smallest A4 portrait (taller than it is wide) rectangle that has (x0,y0) as a corner and contains the current cursor position, modifying whichever of x or y is too small.

3.1.3. Window Coordinate Systems

The LiteWindow class requires the PaintClient routine for the window to examine the window instance variables and organise its own coordinate system. This is awkward in two ways: it requires the application routines to have an intimate knowledge of the window implementation and makes them unnecessarily complicated, and secondly it limits the flexibility of the system by preventing (for example) application independent magnification of the window contents by manipulating the window coordinate system. To overcome this, my subclass of LiteWindow includes a separate method to set the bounding box of the desired coordinate system and private methods to implement this whenever the window is resized. In this way, PaintClient can simply assume the coordinate system and need know nothing of the window framing etc.

3.2. NeWS as a Development Environment for Printer Programs

Armed with these three modifications, I modified Kieron Drake's version of *psview* to provide a PostScript previewing window which has the correct

aspect ratio and coordinate system for an A4 page, and which prompts for a mouse click at each showpage or copypage primitive. I have used this in the other main piece of PostScript work I have carried out using NeWS; development work on a downloadable PostScript font.

3.2.1. Developing a PostScript Chess Font

NeWS does not implement the full PostScript font model, but this does not prevent it from being used to test the character definitions for a font. This is particularly useful when developing cacheable fonts that use the setcachedevice primitive because these fonts are not permitted to change the current colour. This means that the procedures defining the characters cannot use white to erase unwanted portions of black, but must use clipping to avoid making those marks in the first place. In a chess font of this nature there are characters for each (piece, piece colour, square colour) combination and the character shows both the piece and its background. The black squares are indicated by cross-hatching, which must not intrude into the white areas of a piece; for legibility all pieces have a narrow white halo surrounding them. This halo cannot be manufactured by enlarging the piece about some centre, but stroking the piece outline with a thick white line would produce the correct effect. As indicated above, white ink is not available to character routines in this type of font, so clipping must be used instead and this is where trial and error comes in.

By using the improved *psview* to display a full-page sized execution of the character, and editing the definition in another window, causing the *psview* window to be redrawn makes the PaintClient routine re-read the PostScript file* and the effect of any changes can be seen immediately. Using this setup, I experimented with using strokepath to get PostScript to calculate the boundary of the thick line and then incorporate that in the clipping path; unfortunately this doesn't work with either clipping rule when the path is taken as a whole (so presumably PostScript executes the stroke operator by filling the strokepath one subpath at a time). I wrote a procedure using pathforall to compile the strokepath into a list of procedures each of which redraws one subpath, and then added the paths to the clipping region one at a time. This did the trick, but caused our LaserWriters to run out of memory and produce a fatal system error.

*It uses the NeWS run routine to read from a named file, so cannot be used with standard input or from a remote machine.

Finally I returned to the original style I had intended for the Diplomacy board, and settled for using NeWS as a PostScript compiler. By displaying each part of the strokepath, indicating Yes or No by clicking mouse buttons, and writing the equivalent PostScript command for each Yes part, I selected interactively just the exterior parts of the halo and captured them in a file. The output was subsequently edited to make halo-drawing procedures for each piece. I intend to generalise this by converting to a join the dots input method, thus producing more directly the desired procedure and a better-structured user interface (endless Yes, No, No, Yes, Yes ... was very error prone and required you to anticipate what was coming next). An alternative approach to this particular problem would be to calculate the strokepath sub-procedures as above, but write those to a file: there is reason to believe that this would not be so memory intensive for the LaserWriter.

3.3. Conclusions

NeWS is a much better PostScript development environment than a bare PostScript printer. The immediate feedback on the screen, and being able to watch the printing as it happens, makes up for the reduced output resolution. The accuracy of NeWS was good, though I did discover a difference between the LaserWriter and the NeWS interpreter: they disagree about the position of the current point after a closepath operator.

The LiteWindow code is very messy and harder to use than it should be.

There is a lot of potential for using NeWS to simplify programs involving cursor oriented interaction: the key benefit is the fact that NeWS returns the cursor position in the coordinates of the current canvas of the process asking for it. This means that application programs can always receive cursor information pre-scaled to the appropriate coordinate system, for example a Printed Circuit Board editor can receive its mouse position information in the natural 0.05 inch grid, even if the user has elected to zoom in on part of the design. Use of procedures such as my getvalidated routine go even further; a chess program can receive integer board positions in the correct range, just as though the cursor was constrained within the board.

4. Kieron Drake: NeWS as a User-Interface Prototyping Tool

My major interest in NeWS was as a user-interface prototyping tool. I was attracted by PostScript's powerful viewing model, the support for sophisticated graphics operations and the possibility of device independence.

NeWS seemed to promise all of this in an interactive system with, additionally, multiple drawing surfaces, lightweight "concurrent" processes, event handling and, crucially in my view, support for object oriented programming by clever use of the dictionary stack name resolution mechanism. There was also a set of useful interactive bits and pieces provided (windows, menus, sliders, buttons, tick-boxes etc. etc.) as a sort of "class library". I was less concerned with the network aspects of NeWS and intended to write most of my programs in PostScript, in the NeWS environment, and run them on the same machine (the NeWS server).

4.1. Impressions

As part of the process of becoming familiar with the NeWS system I decided to work my way through the various examples described in the documentation, particularly those concerned with Object Oriented Programming (OOP). I also found that I was extensively using the NeWS system, and one of the supplied demonstration packages - *psview* - in particular, as a PostScript previewer for the laser printers. Various problems with the supplied *psview* convinced me to write my own - *myview*. This was later modified further by William Roberts to provide for fixed aspect ratios, multiple pages etc.

Once I was reasonably familiar with NeWS I decided to explore the use of the OOP facilities and the supplied LiteWindow and LiteMenu classes by implementing an interactive application. I settled on a curve drawing/editing system because this would exploit the high-level graphics primitives that PostScript makes available so that I could quickly produce something relatively complex whose user-interaction would be worth evaluating.

4.2. NeWS As A Program Development Environment

The NeWS system provides a multi-window PostScript workstation, complete with various terminal emulation windows, ways of connecting to the NeWS server and the ability to coexist with SunView windows. This allows the user to create and edit PostScript programs and then have them executed by the server. At this point various problems appear.

All type checking is done at run-time, because it's hard to perform static checking of languages like PostScript that support dynamic-binding and (limited) polymorphism. Unfortunately an error in a PostScript program will stop the program and either just break the connection to the server or, if the executive procedure has been invoked, provide a set of stack backtraces. Once this problem has been diagnosed and fixed then the program is re-run and the next error encountered etc. This is tedious. In addition the debugging package provided, though powerful, requires fairly detailed knowledge of the organisation of the interpreter's various stacks and is not really suitable for novice use.

It would also be nice to have a browser which one could use to navigate the class inheritance hierarchy and examine the various classes and the relationships between them. Although a browser is not essential, obviously, it would have been very useful given the knowledge of LiteWindow internals that are currently required to provide one's own PaintClient method. This requirement represents one of my major criticisms of the current LiteWindow class - see below.

Although NeWS isn't really designed to be a PostScript programming environment it would be nice if some such facilities were provided, given that it is necessary to program in PostScript to make use of much of the functionality available.

4.3. Implementing an Interactive Curve Drawer

In order to evaluate the NeWS system I decided to implement a simple curve drawing and editing package under NeWS. I decided to write it entirely in PostScript - mainly because there were suitable curve drawing primitives available that would have been fairly hard work to duplicate in C. In order to keep things simple I envisaged the user creating and interacting with curves by means of the control points which would have to be visible and selectable. Other commands like deleting the current curve and toggling control point visibility would be accessible via menu items.

Proceeding in an object oriented fashion I decided to create a number of object classes as follows:

- **Point**
  This had as instance variables the x and y values and was a subclass of Object. It understood the new message and would create an instance of this class with the x and y instance variables initialised accordingly. It also provided methods for extracting and setting the x and y values as well as a method, called hitPoint, that took an x and y value and a radius and checked to see if the Point is inside the circle so defined: if so, the Point's x and y values were returned, if not the PostScript null value.
- **Curve**
  This had as a class variable a value representing the number of control points it required and had as an instance variable an array of the corresponding size to hold the necessary Point objects. There were also the required new and hitPoint methods. The latter returned either one of the control points of the curve if it fell within the

hitPoint circle or else null. Additionally there were methods to draw, on the current canvas and using the current coordinate system, the curve itself and its control points, respectively. The latter were drawn as crosses within circles of a specified radius.

- **FashionWin**
  This was a subclass of LiteWindow. It was responsible for all of the display and interaction parts of the program. It had class variables FashionMenu and ClientMgrs to hold the required menu and event manager, respectively. These were provided by procedures defined within the FashionWin class and installed as a result of the new and map methods, respectively. The FashionMenu has then to be installed in the superclass' ClientMenu instance variable during instance creation. There were instance variables to hold the particular Curve being created/edited and some booleans NotClear? and ShowPoints? that were toggled by the menu routines and interrogated by the PaintClient procedure that was installed by the new method as a part of instance creation. There was also an instance variable to hold a suitable *Overlay canvas* - more on this later.

My major difficulty was in interacting with the LiteWindow class. In particular:

- **Coordinate Systems**
  It was necessary to set up my own coordinate systems within the PaintClient method and this required knowledge of the internals of my LiteWindow superclass. In particular I needed to know about the width and height of the surrounding frame.

- **Use of LiteWindow Variables and Methods**
  In order to exploit the interaction and menu handling provided by my superclass I had to install my menu into one of the instance variables (ClientMenu) of that superclass. This wasn't any particular problem (except that on stylistic grounds I prefer to do this by sending messages to super rather than having to know the names of class and instance variables within the superclass, but that's just my preference) except that in some cases it was necessary to know a lot about the order in which the superclass did things. Thus references to ClientCanvas make no sense until *after* map has been sent. Also the ClientMenu must have been set (hence the loading from FashionMenu) *before* map is sent. Untangling this network of dependencies requires detailed knowledge of the superclass' internals, and hence access to the PostScript source.

4.4. OOP Under NeWS

I found that I had no problems with the OOP mechanism provided, in fact I enjoyed using it. The only slight difficulty was in remembering that PostScript uses dynamic binding and to ensure that say, procedures defined within classes and that referred to class or instance variables would fail unless invoked as a result of sending messages or otherwise ensuring that the class and instance were on the dictionary stack: in the worked example, the SetCoordinates procedure refers to the BorderLeft class variable and so demowindow begin SetCoordinates will fail saying that BorderLeft is undefined. The bulk of the curve drawing program described above was bolted together in a day or two - mostly for the same sorts of reasons that it would have been easy using Smalltalk-80,[7] C++[8] or Objective-C[9] although the provision of high-level graphics primitives had a lot to do with the relative ease of implementation.

When it came to dealing with the internal complexities of the LiteWindow class a browser might have been useful - as long as it was associated with some classification mechanism (like Smalltalk categories) to bring order to the list of methods etc. As it was I struggled along reasonably well with multiple windows to view the source code.

4.5. Overlay Canvases

One minor complaint is to do with the implementation of Overlay Canvases. These are designed to allow rubber-banding etc. on top of the current canvas but without permanently overwriting its contents. The particular implementation on the Sun appears to involve turning the cursor off and then on before and after accessing it as well as, apparently, redrawing the overlay canvas whenever the cursor moves. This causes both the cursor and the contents of the overlay canvas to flicker as the mouse is moved. In addition, drawing thick lines on such a canvas causes interference between the line segments - they appear to exclusive-or with each other. I realise that these can be implemented in any way convenient for the hardware but it was a minor annoyance.

4.6. Summary

NeWS is nice, but don't expect a PostScript program development environment.

The Object-Oriented Programming code supplied with NeWS works well.

LiteWindow and other provided window management stuff is a mess - Clients have to know too much and yet too little. It would be better to have made more use of *subClassResponsibility* style of working,

with abstract superclasses that just give structure to the class hierarchy and are only used as superclasses to the actual working classes. There is a real danger of LiteWindow being regarded as part and parcel of NeWS and being relied upon because it is there. I believe this is a mistake as it masks much of the flexibility that makes NeWS attractive.

Having access to PostScript's high-level graphics operations enabled a curve drawer (the curve rendering part of which would have been a nightmare in say, C, let alone the rest of it) to be constructed in just a few days.

4.7. A Suggested Window Class Hierarchy

Below is a suggested class hierarchy to provide, and replace, the LiteWindow functionality as well as allowing more general windowing at various levels of complexity that can be selected according to the user's needs.

```
// described in (very) pseudo Objective-C:
// Class : SuperClass {
//         inst_vars,    // start with [A-Z]
//         methods,      // start with [a-z]
// } // comment describing class

ShapelessCanvas : Object {Parent, Position,
        Size, Menu, //may be NULL
          AlternateForm, // icon? may be NULL
          Children, //may be NULL - subwindows
          map, unmap, // etc. etc. etc.
          interactiveShaper = subClassResponsibility,
          repaint = userResponsibility // up to user
} // handles menu, if there.
// Shape is responsibility of subclasses

FrameShapedCanvas : ShapelessCanvas {
          interactiveShaper = getNestedRects
} // Frame shaped thing

RectShapedCanvas : ShapelessCanvas {
          interactiveShaper = getRect
} // Rectangular thing - probably forms
// middle of a window

A4ShapedCanvas : ShapelessCanvas {
          interactiveShaper = getA4AspectRatioRect
} // Preserves Aspect Ratio of A4 sheet

RectWindowFrame : RectShapedCanvas {
          Title, // perhaps NULL
          .... etc....
} // supports standard LiteWindow frame-menu,
  // has default AlternateForm (iconic form) of
  // small (titled) rectangle. Could have other
  // stuff in addition to title (KeyFocusColor...)
  // Generally would have one or more
  // RectShapedCanvas as Children

TiledRectWindowFrame : RectWindowFrame {
          TileChildren // lays out children,
                  // initially as tiles
} // This could be used to have a tiled set of
  // subwindows or, alternatively, each child
  // could ask its parent for its Children list
  // (container) and ask them about their positions,
  // sizes etc. and then lay itself out accordingly...
```

The essential point here is that the repaint method, if defined, is invoked within a coordinate and dimension system suitable for the child canvas. It is independent of the parent, which only functions as a demultiplexer to direct damage, move, go-to-AlternateForm etc. etc. messages to the children. Children can, in turn, consult their parent if required. All children have parents and any can have children.

Notice that a typical LiteWindow equivalent is in fact made up of three objects: a RectWindowFrame and two RectShapedCanvases (one is the child and forms the interior of the window while the other is the AlternateShape and forms the icon and also has the frame as parent).

This mechanism is not limited to rectangular windows. The user is free to define any subclass of ShapelessCanvas and provide any desired InteractiveShaper method. One suggested example above preserves an A4 Aspect Ratio. One of these, inside a RectShapedCanvas, could be interactively sized, via the frame, but would always retain the required aspect ratio. This would be useful for previewing PostScript destined for a laser printer that was loaded with A4 paper.

5. Mel Slater: Graphics Programming with NeWS

PostScript was developed from an imaging model suitable for describing printed pages and as such all of the PostScript graphics operations assume a 2D model. This is a very powerful model compared to traditional graphics systems, such as the one provided by the 2D standard Graphical Kernel System (GKS)[3] and the associated proposed standards such as the Computer Graphics Interface (CGI) and Computer Graphics Metafile (CGM). Moreover, taking into account the event input extensions provided by NeWS, the overall interactive graphics functionality provided by GKS is far behind that of the NeWS system. A full comparison between GKS and PostScript is given in Salmon and Slater.[10]

5.1. NeWS for 3D Graphics

NeWS clearly provides excellent tools for 2D interactive graphics. It has also been found to be suitable for the generation of 3D wire frame and shaded graphics

images, including, for example, Bezier bi-cubic patches. This statement is based on a small project which involved implementing a 3D viewing system, and generating a Bezier surface. The reason for this exercise was to generate some illustrations which could be printed on a LaserWriter. Screen dumps were not acceptable because of their relatively poor quality. Therefore a PostScript program had to be written. NeWS proved to be an excellent development environment, and the seemingly daunting task of writing a 3D graphics program proved to be remarkably straightforward. NeWS extensions such as dictbegin and dictend were helpful in development, and then the programs were converted to standard Adobe PostScript for use with the LaserWriter.

5.1.1. Structure of the PostScript Program

The PostScript program was divided into four logical modules.

- **Matrix Handling:**
  The first provides basic (4 by 4) vector and matrix handling facilities, such as matrix multiplication, vector cross products, and so on.
- **3D Viewing:**
  This involved the definition of a dictionary for storing the current view specification, which included the viewing parameters View Plane Normal, View Up Vector, View Reference Point, Centre of Projection, View Distance, and Front and Back Clipping Planes - based on the usual CORE model.[11] A series of viewing parameter access functions to this dictionary were provided, such as SetVPN, SetVUV, SetVPR and so on. Finally, the SetView function produced a view matrix from the information currently stored in the view specification dictionary.
- **3D Paths:**
  The third module provided a simple 3D graphics operator. Points were defined as an array of three numbers, for example [x y z]. A polygon was defined as an array of points and the operator path3D produced a new PostScript path according to the current view matrix. Hence,

```
/polygon [ [x1 y1 z1] [x2 y2 z2] ... [xn yn zn] ] def
polygon path3D stroke
```

  would generate the 2D screen representation of the 3D polygon with corners (x1,y1,z1)...(xn,yn,zn).
- **Bezier Surfaces:**
  The final module was a specific application for the generation of wire frame Bezier surfaces. Here a four by four control matrix of 3D points is supplied, and the standard recursive splitting algorithm used to generate the actual surface. Some results are illustrated in Figure 4.

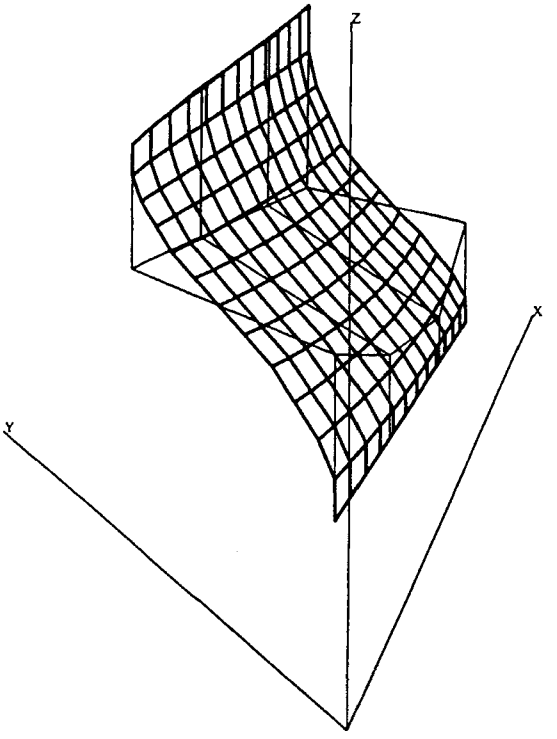

Figure 4. A Bezier Surface (showing the control points)

5.2. Using NeWS and PostScript from C Programs

A useful feature of NeWS is that higher level languages such as C or Pascal can be used in conjunction with PostScript. (At the moment this is only directly supported for C, though sufficient information is given in the NeWS documentation to allow the same set up for other languages).

With C the arrangement is that the programmer can create a file consisting of C stubs, with definitions in PostScript. For example, to define a simple line function, and a simple clipping function (for rectangular regions), the file (say, psheader.cps) would appear as follows:

```
cdef PSLine(x1,y1,x2,y2)
  x1 y1 moveto
  x2 y2 lineto
  stroke
```

```
cdef PSClip(x1,y1,x2,y2)
   newpath
      x1 y1 moveto
      x1 y2 lineto
      x2 y2 lineto
      x2 y1 lineto
   clip

cdef PSInitClip()
   initclip
```

This file can then be used as input to the *cps* preprocessor produces a new file psheader.h, which is a C header file containing macros. These macros contain definitions which when invoked send the appropriate PostScript code to the server (hence the name cps - C to PostScript).

A fragment of C code which uses the above definitions might be:

```
#include "psheader.h"

PSInitClip();
PSClip(0,400,0,400);
PSLine(0,0,400,400);
PSLine(0,400,400,0);
```

By default, parameters in the C stubs are ints. The NeWS documentation claims that other types should be possible (such as doubles) but this appears not to be implemented at present. It is also possible to pass information back from the server to the C application.

5.2.1. Comparison with Other Graphics Programming Libraries

It is interesting to note that this method of producing graphics from C programs actually has considerable performance advantages over SunView.[6] This is because the NeWS PostScript drawing and clipping operations are faster than those provided than under SunView, and of course rather more sophisticated. It can also be reasonably argued that NeWS PostScript provides a more tractable windowing and graphics environment than SunView.

Given existing graphics software written in C it is not difficult to port this to the NeWS environment, and experience so far suggests that the effort is worthwhile.

5.3. Conclusions

There was nothing in the 3D program which was particularly difficult to describe in PostScript terms (the full program is given as an Appendix in Salmon and Slater.[10] Better error messages would have been helpful, but the same complaint could be made about virtually any programming language with which the author is familiar. Speed of execution was acceptable, and as expected given the complexity of the problem. Future work in this area will concentrate on shading, and 3D interaction.

The facilities for using PostScript from applications programs are straightforward to use and offer some advantages over traditional graphics libraries, not least the reduced size of the compiled program (which doesn't need all the detailed code for handling the raster devices).

6. Peter Williams: Using NeWS in Commercial Application Development

Software Manufacturers have a wide choice of portable window manager products available for supporting their applications. A major topic of debate amongst software manufacturers of all sizes at the current time is which one to support for their product lines?

Let us consider and investigate the some of the properties of NeWS with respect to a large-scale high-quality multi-mode, multi-media interactive documentation system to run on a networked scientific or technical workstation. Such a system should exploit networks both for remote processing and imaging to provide the flexibility a User demands, yet provide adequate response times at the WIMP MMI such that the User's performance of a task is not degraded. These needs encourage us to research the requirements for both the graphics processing required for imaging information on the screen, and the ability to exploit graphics features in the application. In the case of a documentation system, we need to consider the facilities that graphics can offer, if any, in interactive document layout and formatting.

A very important facility of the graphics support offered by any window manager is its capacity to render information both accurately and in a manner which supports interactive screen processing, as directed by a wide-range of possible presentation modes and attributes. To investigate both the feasibility of using PostScript graphics to support a high-quality documentation application, and the architectural facilities of NeWS for supporting networkable, and separable, User interfacing to such a tool, the traditional problem of troff previewing was readdressed.

6.1. Previewing troff output

The standard batch mode processing techniques of tbl/eqn/pic/troff source documents were installed into an appropriate makefile, with appropriate centralised remote processing over the network to run the troff for-

matting software system. The Documenters Workbench software package is widely available, portable on most Unixs, and in not inconsiderable use. It is, however, a non-interactive mode of document layout and presentation and I recognise this glaring deficiency in the experiment.

Using ditroff coupled with the Adobe Transcript software, the raw postscript was presented onto a NeWS canvas, running inside a window. Using mechanisms discussed elsewhere in this paper, WIMP UI functionality was associated with the window to allow User control of some the presentation attributes. All programming was done in Postscript, with suitable massaging of the Transcript output to support the User interaction and reimaging of pages.

6.2. Conclusions

Working directly with Postscript graphics is pleasant and it is clearly possible, even from the rather trivial troff experiment, to build complex graphics-based applications. However the experience of the exercise suggests that to adopt purely this mechanism of building interactive software is to go right against the trend in manufacturing circles to adopt standards - both in graphics and networking. One can imagine formulating mappings between appropriate Postscript definitions and the functionalities defined in GKS, CGI, PHIGS, or the presentation sections of standards like ISO 8613 (ODA). But, it must be asked whether the various levels of graphics processing required are worth the advantages of PostScript and the novel architecture to the market-hardened software manufacturer with raster-based products.

Undoubtedly NeWS can independently support complex and demanding modern graphical applications - indeed its "networkable" and "extensible" properties are undoubtedly key features. But its very success in solving a number of intellectual and practical problems may be the source of its poor general acceptance! The decision to take the relatively staid X Window technology into consideration towards a truly portable window manager system for general release in the User community is undoubtedly wise, though less satisfying.

7. Allan Davison & Antony Simmins: A NeWS Terminal

The NeWS and X systems are likely to became de facto standards for distributed window systems. Both of these systems are based around a window or display server which manages the display hardware, ie screen, keyboard and mouse, on a machine. Applications programs, which may be running on a different machine on a network, communicate with the server to manipulate the display. This server/client structure enables an effective separation of an application's display and computation requirements.

A machine designed to run only the server software would provide a powerful windowed terminal without requiring all the functionality of a workstation; in particular, local discs, virtual memory and a general purpose operating system would not be required. This concept is not new; in the early 1980's AT&T Bell Laboratories developed a terminal called the Blit,[12] which directly supported multi-windowing onto multiple Unix processes. The Blit was successful, particularly within Bell Labs, but a major limitation of the design was its use of standard serial communications to the host system. The decision to use a serial interface was made because network protocols and hardware were not standardised at that time.

A project to design and build the prototype for an ethernet terminal to support NeWS has started at Queen Mary College. The prototype will be based on the INMOS T800 transputer with 4Mbytes of main memory and 1Mbyte of colour frame buffer to support a 1152x900 pixel resolution display with 8 bits per pixel. The only other hardware components required by the prototype are an ethernet connection, keyboard and mouse. We believe that the prototype will have twice the performance of a Sun3/160 when running NeWS and expect it to demonstrate that an ethernet terminal could be produced for approximately £2,000.

Our experience with Postscript and NeWS suggests that the speed of floating point arithmetic and graphics primitives, in particular masked block copy, are important to their overall performance. The transputer provides hardware support for these functions with a floating point unit that is five times faster than the 16MHz 68020/68881 combination and efficient 2D data movement instructions that support copying, overlaying and clipping of graphics images based on byte sized pixels. Finally, the choice of transputer technology will allow us to investigate the architecture of multiprocessor systems to support NeWS and similar environments.

8. Shortcomings of NeWS

As the NeWS system is composed of a PostScript interpreter and a collection of PostScript routines, we will consider these two parts separately (for completeness, we shall also indicate the parts of NeWS which have not been studied at QMC). The version we have been using calls itself Version 1.0, but there is some uncertainty about this.

8.1. The Interpreter

The NeWS PostScript interpreter is essentially a faithful implementation of the Adobe original, with the majority of its extensions completely in character with the original. We have not yet found any major omissions in function or errors in form, though there are a number of bugs in the release we are working with. Some small additions would be desirable:

- **imagecanvas:**
  Imaging the whole of the source canvas is often inconvenient and it would be useful to be able to use the interior of a path in the current canvas as the source instead.
- **writecanvas:**
  A facility to write the contents of a canvas in an implementation dependent form would allow easier interworking between NeWS and some existing pixel systems. This would be a weakening of principles however, as PostScript has no notion of reading what has previously been drawn.
- **undef:**
  More comprehensive storage management will be required for a windowing system where processes execute asynchronously; it will not be possible to make rational use of the PostScript save and restore primitives as the memory they manage is shared between all processes. A primitive for changing the size of a compound object would be very useful for managing dictionaries.
- **itransform:**
  In PostScript the correspondence between the working coordinate system and the underlying device coordinate system is controlled by the current transformation matrix (CTM). The default PostScript coordinate system has the origin in the bottom lefthand corner with 1 unit = 1/72nd of an inch. Instead of relying on this existing mechanism for device independence, the NeWS interpreter tries to pretend that the underlying hardware has a coordinate system with its origin in the bottom left corner of the display. On Sun hardware this is not true, leading to some bizarre behaviour when using NeWS matrix transformations for non-imaging purposes. This unnecessary extra layer of device independence should be removed from NeWS at the earliest opportunity.
- **currentpath:**
  The shape objects generated by this primitive can be used with setpath to set the current path. However, it does not seem to be possible to extend it by adding new subpaths: the commands setpath 0 0 moveto have exactly the same effect as newpath 0 0 moveto!

The standard PostScript primitives currently not supported in NeWS should all be implemented at the earliest opportunity, with priority given to dashed lines and the full PostScript font machinery.

The implementation of overlay canvases is rather weak: the definition given of the createoverlay primitive explains that it does not have to support the full PostScript imaging model, but what it does currently support seems rather too reduced for anything except simple thin as possible lines. Developers porting NeWS to other hardware should consider devoting special purpose hardware to support for overlay canvases; the stated intention of the vague definition is to allow exploitation of such hardware.

The use of non-preemptive scheduling is a major step backwards in operating system design; it makes the use of new NeWS code fairly hazardous because a rogue process can run indefinitely without relinquishing control to the other processes. It is not even possible to terminate NeWS cleanly, because the NeWS server will not respond to mouse events or attempted stream connections.

8.2. Additional PostScript Code

The additional PostScript routines supplied are of very variable quality. The code implementing classes and the LiteMenu class has been found to be robust and usable; the code for the LiteWindow class has not.

8.3. LiteWindow

The LiteWindow class is a workable implementation of windows, but it is completely at odds with the major PostScript style rule:

> "Do not compromise device independence or reusability by requiring PostScript code to have detailed knowledge of its environment"

For an application, in particular the PaintClient routine associated with an instance of a window, to use LiteWindow successfully, it must have a detailed knowledge of the class implementation: in order to calculate something as fundamental as the size of its canvas, the application must know to compensate for the size of the window frame. As a consequence, windows without frames would need to include dummy frame information in order to be usable with code written to run under LiteWindow.

This has wider reaching implications: it is not possible to exploit the arbitrary coordinate transformations available in PostScript to provide, for example, application independent-magnification facilities. What is needed is a way for the application to declare to the window the environment it needs, and defer to the win-

dow all the problems of creating this environment. William Roberts has experimented along these lines by constructing a subclass of LiteWindow with a method clientbbox with which the application declares the coordinate bounding box it requires and then assumes that the window does the rest: in the earlier worked example, the SetCoordinates routine would not be necessary and a single call of [-1 -1 1 1] clientbbox would set the window correctly regardless of how many times it is resized. To provide a magnification feature for the window class would then require determining an addition coordinate transformation to apply before the PaintClient routine is actually executed.

The LiteWindow ParentCanvas class variable must become an instance variable and care taken to use only overlays of the ParentCanvas instead of the framebuffer for interaction. Without such a change, it is not possible to use LiteWindow to make sub-windows (for example in the document formatting system described by Peter Williams) that behave properly; moving in the hierarchy along with their parent window, not being hidden below the parent window etc.

The LiteWindow class is also rather too big, and could usefully be reworked as a hierarchy of simpler classes with LiteWindow near the bottom: one possible hierarchy is suggested by Kieron Drake in Section 4.

8.4. Low-Level Animation

The routines provided for cursor following, getclick and getanimated, are not well written and rather tied into the window system; for example, getclick insists on drawing a rectangle which follows the cursor, just as when reshaping a window. The getanimated routine ignores the mouseposition associated with its termination event, returning instead the position of the previous event; with fast cursor motion this can result in a returned position different from the click position. Getanimated is further complicated by the inclusion of code for constraining motion to be either horizontal or vertical; this is used in the LiteWindow code but should not be part of this supposedly general-purpose routine.

William Roberts has implemented a generalisation of *getanimated* which separates processing the cursor position from any redrawing, allowing easier programming of routines requiring some constraint on the cursor position, e.g. integer coordinates on a chess board or determining the smallest rectangle of given aspect ratio that contains both x0,y0 and the current cursor position.

8.5. Miscellaneous

The dictbegin and dictend routines are supposed to permit the creation of arbitrary sized dictionaries, but in fact they alway create dictionaries with room for 200 entries.

The documentation is weak in some areas, particularly storage management and the lifetimes of PostScript objects. Does NeWS do garbage collection? Do canvasses need to be explicitly destroyed in some way or is their storage reclaimed when all references to them have been lost?

Most of the demonstration routines assume that the full PostScript line-drawing model has been turned off (by use of the setlinewidth primitive) and don't work properly when it is enabled.

A simple text output window would be useful for many purposes, particularly monitoring and debugging.

8.6. Areas of NeWS not Yet Studied

For completeness, we include a short list of other features of the NeWS interpreter and its related code (both PostScript and C programs) which have not been examined in any depth.

- **LiteItem**
- **Text Input**
- **Raster Fonts**
  NeWS does not implement the true PostScript font model, but instead supplies a collection of fonts and various programs for making new font bitmap masters. No one was at all interested in this.

9. Comparison between X and NeWS

There is another, more strongly supported, player in the device-independent network window system arena: the X system developed at MIT and now endorsed by DEC and IBM to name but two. NeWS is a proprietary standard and the newcomer, but hopes to become a *de facto* standard in the way that SUN NFS has become a *de facto* standard for network file systems. NeWS and X appear to offer superficially similar things; both take the form of a server running in the machine with the display, to which application programs running anywhere connect via a byte-stream protocol. Both achieve hardware device-independence by using the server to provide an abstract device; the display programmer programs that abstract device and the server does the rest. Both exist as carefully written "reference ports" which can be ported to a new machine in a matter of a few working days.

They are however quite different when looked at more closely.

9.1. The Abstraction Provided by X

The X system is an attempt to rationalise a lot of exist-

ing hardware: any computer supporting 70-90 dot per inch screens, each with associated mouse and keyboard. The abstraction supports overlapping rectangles of pixels, *pixmaps*, which are arranged in a hierarchy. The coordinate system for each pixmap has (0,0) in the top left corner, the y-coordinate increases downwards and the unit is one pixel (hence all coordinates are integers). It has event-based input handling and provides facilities for text, graphics with thick lines and curves. The abstract device is controlled through a procedure call interface.

The key concept is the pixel, which may consist of several bits, and pixel colours. The use of colour is controlled in a device independent way, but not all X servers need be capable of supporting it and so the application is provided with routines for finding out the capabilities of the actual device in use. Sophisticated applications need to use similar routines to establish the true size of the display, the pixel resolution etc.

9.2. The Abstraction Presented by NeWS

The NeWS abstraction is full colour device with arbitrarily high resolution. It supports arbitrarily shaped overlapping regions of drawing surface, *canvases*, arranged in a hierarchy. The coordinate system for each canvas is arbitrary (it can be subject to arbitrary linear transformations by the application) but the natural coordinate system has (0,0) in the bottom left corner, the y-coordinate increasing upwards and the unit is 1/72nd of an inch (all coordinates are given as real numbers). It has event-based input handling and supports text, lines with thickness and intensity maps. The abstract device is controlled through a program interface, i.e. the primitives of the device include traditional programming constructs such as procedure definition, iteration etc.

The fundamental concept is a path; a set of curves defined in the coordinate system and then rendered on the device by stroking or filling. The device supports arbitrarily high resolution colour. The application has little or no way of determining the true capabilities of the actual physical device.

9.3. Closer Comparison

The first key difference between X and NeWS is the attitude to pixels: X is completely based on pixels, in NeWS pixels do not exist. X provides the full 16 possibilities for combining a new pixel with what is already present; NeWS has no concept of reading back what is already there and all drawing is performed by overwriting*. X must have a font file for each size and orientation; by eschewing pixels, NeWS becomes free to scale its fonts and coordinate systems in arbitrary ways, with any orientation and even with non-perpendicular axes.

The second key difference is the interface provided to clients of the server: X provides a procedural interface, NeWS provides a programming language. The ability to send arbitrary programs to the server has a large impact on the communications bandwidth needed to operate the display remotely, especially when fast interactive feedback is needed or the display involves a large number of repeated designs.

The difference in facilities is highlighted by the observation that it is possible to program the NeWS abstract device to be an X server, with the exception of any X operations which read back the existing pixels: Sun supply a partial implementation (in PostScript) of X.10 with the NeWS system.

9.4. Crystal Ball Gazing

In the short term (1-3 years) the X standard will prevail. It bears fairly close resemblance to a lot of existing hardware, it is easy to port, and the emphasis on pixels corresponds to the accepted practice of fast, interactive computer graphics over the last five years. Nothing about X precludes the sort of performance seen of arcade games, including all the tricks that can be played with colour maps. During this time NeWS ports will become available from most major manufacturers, simply because NeWS is very easy to port and not particularly expensive to licence.

In the long term, NeWS (or something similar) will supercede X, as surely as bitmap graphics is superceding dumb terminals. The NeWS abstract device has a much higher functionality and its assumptions of arbitrarily high spatial and colour resolution are future-proof: NeWS can already span the small range of available display resolutions (from 50 dpi to about 90 dpi) and the size of the application graphics remains constant, though the quality varies. When 200dpi screens become available, a standard X application window will shrink to ¼ of its size on current displays.

NeWS is not slow; Mel Slater has applications where NeWS with clipping is faster than SunView, both with and without clipping. Furthermore the size of the compiled program to run under NeWS is reduced to 20% of its size of under SunView.

10. Conclusion

NeWS is a revolutionary combination of PostScript with the best of existing workstation and network tech-

*NeWS does have a setrasteropcode primitive which controls the way colour is combined with the existing drawing surface, but this is only to enable NeWS to emulate existing window systems; you still can't read back the pixels.

nology. It is undoubtedly the best windowing and graphics environment we have ever come across; we intend to use NeWS on all the machines in our network and to develop its ideas in future research work.

References

1. Sun Microsystems, *NeWS Preliminary Technical Overview*, October 1986.
2. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics* **5**(2) (April 1986).
3. ISO, "Information Processing - Graphical Kernel System (GKS) - Functional Description," ISO 7942 (1985).
4. J.H. Morris, M. Satyanarayana, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM* **29**(3), pp. 184-201 (March 1986).
5. Adobe Systems, *POSTSCRIPT Language Reference Manual*, Addison-Wesley (July 1985).
6. Sun Microsystems, *SunView Programmer's Guide*, September 1986.
7. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley (May 1983).
8. B. Stroustrup, *The C++ Programming Language*, Bell Laboratories (November 1984).
9. B. Cox, *Object Oriented Programming - An Evolutionary Approach*, Addison-Wesley (1986).
10. R. Salmon and M. Slater, *Computer Graphics: Systems and Concepts*, Addison-Wesley (August 1987).
11. "Status Report of the Graphics Standards Planning Committee," *ACM Computer Graphics* **13**(3) (July 1979).
12. R. Pike, "The Blit: A Mulitplexed Graphics Terminal," *A.T. & T. Bell Laboratories Technical Journal* **68**(8) (October 1984).