A User-Interface Toolkit in Object-Oriented POSTSCRIPT

Owen M. Densmore
David S. H. Rosenthal

Sun Microsystems
2300 Geng Rd.
Palo Alto CA 94043

ABSTRACT

Only the die-hards want to develop applications using basic window system facilities. The preferred approach is a higher-level ‘‘toolkit’’ of user interface components, such as menus and scroll bars. Experience with current toolkits shows the need for an object-oriented interface to these components.

NeWS, the Network/extensible Window System, allows user interface components to be programmed in POSTSCRIPT. Fortunately, object-oriented interfaces are natural in POSTSCRIPT, and they have been used to create a toolkit with some novel properties. Among these are the use of concurrent processing and run-time inheritance of component methods.

‘‘PostScript is the future of words on paper.’’ *Arthur C. Clarke*

1. Introduction

The needs for portability, distribution, and standardisation are encouraging the implementation of window systems for UNIX as network window servers, rather than as extensions to the operating system kernel. These servers are user-level daemon processes; clients connect to them and make what are effectively remote procedure calls in order to create and destroy windows, and to draw in them. Examples include Carnegie-Mellon’s *Andrew*,[1] MIT’s X Window System,[2] and Sun Microsystems’ NeWS.[3]

Most network window servers base their client-server communication on a fixed protocol; applications express the operations they need in terms of the fixed set of operators supported by the protocol. The imaging model they support is pixel coordinates and RasterOp. In NeWS, by contrast, client-server communication uses a subset of Adobe’s POSTSCRIPT[4] page description language. POSTSCRIPT is a Forth-like programming language, and it supports the Warnock and Wyatt imaging model.[5] The fact that applications are talking to a programming environment with a high-level imaging model, rather than a fixed-function RPC server with pixel-level imaging, has radical effects in both the printing and window system worlds.

- It provides for resolution-independent and display technology-independent imaging.

Trademarks:

Mark	Owner
MacApp	Apple Computer Inc.
NeWS	Sun Microsystems Inc.
Object Pascal	Apple Computer Inc.
POSTSCRIPT	Adobe Systems Inc.
SunView	Sun Microsystems Inc.
UNIX	Bell Laboratories.
X Window System	Massachusetts Institute of Technology.

- It allows for extensibility in a controlled and portable way. Applications can customise the (display or printing) service to their needs by writing programs.
- It provides for a flexible distribution of function. Individual applications can determine which of their operations should be performed in the printer or display service.
- It provides an integration between printing and displays that has been lost since the days when both teminals and printers spoke ASCII. This integration naturally supports both text and graphics.

To support interactive displays, NeWS extends POSTSCRIPT with new primitives for:

- multiple overlapping drawing surfaces
- multiple threads of POSTSCRIPT execution
- multiple input devices

2. Toolkits & the Need for Object-Oriented Programming

Some window systems, Andrew and the Mac are examples, attempt to enforce a consistent style of user interface across all the applications that use them. Others, SunWindows and X are examples, attempt only to provide low-level mechanisms, and avoid specifying any details of the appearance or function of an application's user interface.

Nevertheless, systems like SunWindows and X do not expect every application to hand-craft its user interface from scratch. They normally provide a layer above the basic window system that implements common components of a user interface, such as menus, scroll bars, buttons and text panels. This upper layer provides a ''toolkit''; a user interface can be rapidly assembled by selecting and composing tools from the kit.

As experience has been gained with these toolkits, it has become obvious that the appropriate interface to the user interface components they supply is an object-oriented one. Applications want to be able to take generic objects from the toolkit, such as a menu, and customise them for their need without needing to understand their internals. They want to be able to create sub-classes of the generic object classes, to create, for example, the class ColorChoice from the class menu. In this way applications can inherit consistent behaviours for all the aspects of a menu they do not need to specialise, and specify only the details they are really interested in.

This evolution towards object-orientation is visible in the history of many existing toolkits, for example Sun's SunView toolkit. But the best example is MacApp,[6] because it was able to use an object-oriented language (Object Pascal) rather than to work, as SunView was forced to, in an object-oriented style in a language that doesn't support it (C).

Many of the essential ideas in object-oriented systems are similar to the more traditional ''package''- or ''module''-based systems. Briefly:

- Packages (modules) are replaced by *classes*.
- Procedures in packages are replaced by *methods* in classes.
- Creating package objects is replaced by creating new *instances* of a class.
- Package local and global variables are replaced by *class variables*.
- Object variables are replaced by *instance variables*.

New notions are:

- Classes are ordered into a hierarchy by *subclassing* a new class from a prior one, *inheriting* its methods, instance variables, and class variables.
- Methods are invoked by use of the `send` primitive. The term *message* is used for an invocation of a method with its arguments.
- There is a means of constructing classes on the fly. This is absent from most languages' module creation.
- Two new concepts, the *self* and *super* pseudo-variables, are introduced. They are used in methods to refer to the object that sent the message and the method's superclass, respectively.

3. Concurrency

A major problem in writing user interfaces in conventional sequential programming languages is concurrency.

> ''Providing a suitable graphical display is not especially difficult; what causes problems is the complicated flow of control required to deal with all the possible sequences of user actions with the input devices. One might consider a scrolling menu ... as a finite state automaton reading a token for each event ... however ... the presence of multiple input devices invalidates the notion of a single stream of tokens.''[7]

The problems are legion. The user may be using multiple physical devices simultaneously. The application may wish to update its image while the user is dragging a slider. An individual event may be of interest to a number of user interface components.

To cope with the multiplicity of independent input sources toolkits for conventional sequential languages have been forced to take the flow of control away from the application; instead of doing explicit I/O the application must register interest, describing the routines it wishes called when certain I/O events occur. The ''notifier'' of the SunView toolkit is an example of this approach.[8] The difficulties it causes when attempting to port pre-existing applications or to add window system support to other languages are well-known. Further, each routine registering interest in an I/O event must implement its own state machine, preserving its state in private storage. Each call is a separate invocation, preventing use of the stack for retaining state.

Cardelli and Pike[7] tackle this problem by inventing a new language (''squeak'') with explict support for concurrency, and compiling it into a sequential language (C). NeWS also supports concurrency explicitly. POSTSCRIPT has been extended with operators that fork new threads of execution, wait for threads to exit, create and use monitors to interlock between threads, and send and receive interprocess messages. These threads of execution are called ''lightweight processes'' because they exist in a single UNIX address space; they are artefacts of the POSTSCRIPT interpreter and need no special kernel support.

POSTSCRIPT lightweight processes are very cheap, and they can be used without inhibition to implement user interface behaviours. A typical menu implementation, for example, will use one process to track the mouse and highlight the current menu selection, and another to listen for the button up event that activates the selection. Moving into a pull-right will fork another pair of processes.

4. Object-Oriented POSTSCRIPT

The POSTSCRIPT implementation of classes uses dictionaries to represent the classes and instances. Instances contain all the instance variables of all their superclasses. Classes contain their methods as POSTSCRIPT procedures. Our current implementation of classes is entirely in POSTSCRIPT, it uses none of the NeWS extensions, and the details were described at the USENIX Monterey workshop.[9]

Briefly, the class implementation depends on POSTSCRIPT's use of a dictionary stack to resolve names. All names used in POSTSCRIPT are resolved at run-time by looking them up in each of the dictionaries in a process' dictionary stack in turn, starting from the most recently pushed dictionary. This allows for fine-grained control over the naming context of each operation and, since dictionaries can be shared, provides control over name space sharing between processes. When a POSTSCRIPT process forks, the child inherits a copy of the parent's dictionary stack, and thus operates in the same name space. Both the parent and the child can push new dictionaries, creating local names, or pop them, removing their access to shared names.

The class implementation provides the following operators:

- classbegin and classend. These operators create a new class (represented by a dictionary), given a name, a list of instance variables, and a superclass. They bracket a series of defs used to add methods and variables to the class. Here, for example, is the definition for the basic class Object:

```
/Object null [] classbegin
    /new {          % class => instance (make a new object)
        ...
    } def
    /doit {         % proc ins => - (compile & execute the proc)
        ...
    } def
classend def
```

It defines two methods, new to make a new object, and doit, a sort of meta-method. It takes a procedure as argument, and executes it in the context of the object. In this way, new methods can be added at run-time.

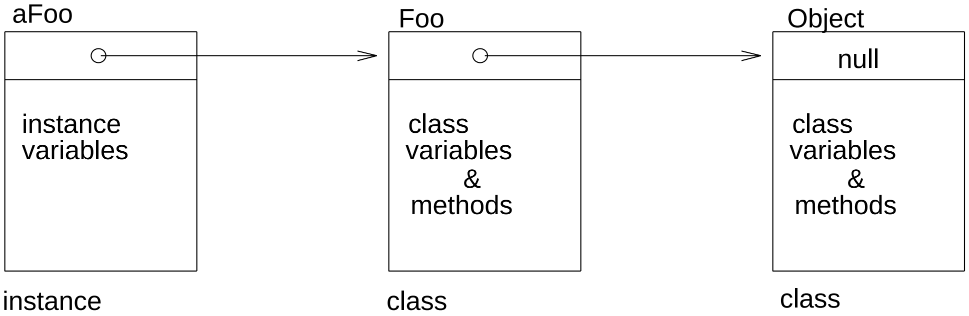
- send. The methods of an object are invoked by send. It takes some (optional) arguments, the name of the method, and the object and then:
  - establishes the object's context by putting it and its class hierarchy on the dictionary stack.
  - executes the method.
  - restores the context by removing the object and its class hierarchy from the dictionary stack.

  For example, we make a new Object by:

  ```
  /new Object send
  ```

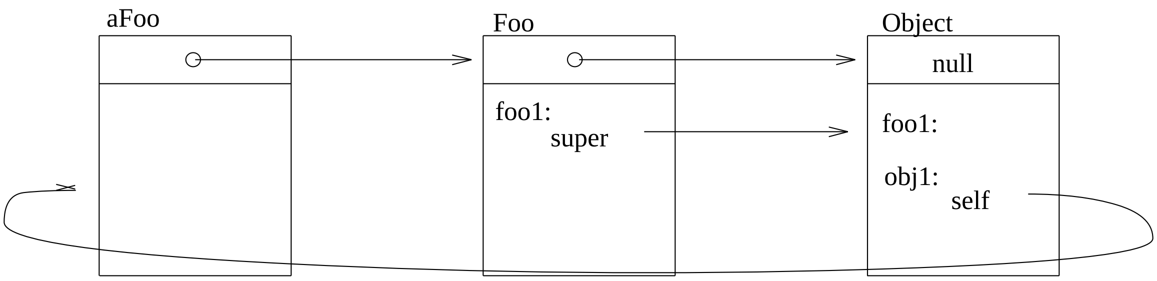
- self. When used as the object with a send, it refers to the instance causing the current method to be invoked.
- super. When used as the object with a send, it refers to the method being overridden by the current method.

The relationship between an instance and its class and superclass is shown in the figure below. We have made an instance, 'aFoo,' of class 'Foo,' which is a subclass of class **Object**. An instance has a copy of all instance variables of its superclasses, thus 'aFoo' has those required by both 'Foo' and **Object**. The methods known by an instance are stored in the classes in its superclass chain. Thus 'aFoo' can only respond to methods residing in 'Foo' and **Object**.



Relationship between Instances and Classes

Sending a message to an instance requires packaging the arguments to the method, finding the method in the class chain, invoking the method in the proper context, and possibly returning a result to the sender. If the pseudo-variable **self** is used for the object in sending a message, the search for the method starts at the beginning of the chain, while if **super** is used the search starts in the superclass.



Self and Super

In order to use **self** and **super**, the method must be ‘‘compiled’’. This is done automatically by **classend**, and by the **doit** method of class **Object**. This binds the names **self** and **super** at method compile time.

5. The ‘‘lite’’ Toolkit

NeWS supplies a toolkit, the ‘‘lite’’ toolkit, based on this POSTSCRIPT class mechanism. It provides a basic class **Object**, and three important subclasses:

- **Window** – the window object is simply a set of canvases and an event manager, a process listening for specified events and executing corresponding actions. The default window style manages a **FrameCanvas**, a **ClientCanvas**, and an **IconCanvas**. It provides two types of user interface management: menu interaction and direct mouse interaction with the window or icon.
- **Menu** – Menus associate a key, generally a string, with an action to be performed when that key is selected by the user. If the menu action is another menu, it is displayed in turn.
- **Item** – A common need in interactive systems is a simple, user-definable, graphic, interactive, input/output object. Examples are buttons, sliders, scrollbars, dials, text fields, message areas, and the like. The class **Item** defines a skeleton for such an object.

The item package currently implements the base class, **Item** (which is useless by itself), the subclass **LabeledItem** (which also is useless), and several practical subclasses of **LabeledItem** (which *are* useful).

An **item** has these major components:

- A canvas that depicts the item and is the target of the item’s input.
- A set of procedures that paint the canvas and handle activation and tracking events.
- A current value and a procedure that notifies the client when that value changes due to action of the tracking procedures.
- Methods for creating, moving and painting the item, and for returning the item’s location and bounding box.

The **LabeledItem** class adds to these:

- A polymorphic label-object pair, either of which may be a string, an icon, or a general POSTSCRIPT procedure.
- A ‘‘round rectangle’’ frame enclosing the item.
- Simple layout rules for automatic positioning of the label and object. The object position may be to the Right, Left, Top or Bottom of the label.

The (useful) subclasses of **LabeledItem** are:

- **ButtonItem**: provides a simple activation/confirmation item
- **CycleItem**: provides check boxes and choices
- **SliderItem**: provides a continuous range of values
- **TextItem**: provides a type-in area
- **MessageItem**: provides an output area

- **ArrayItem**: provides an array of choices

The toolkit simultaneously exploits the NeWS lightweight process mechanism and hides it from the application using the toolkit. Windows have processes listening for manipulation commands such as pop, move, and reshape, and executing them asynchronously with the application. Menus have processes painting their images and executing the associated commands. Items have processes listening for user actions and generating appropriate echoes.

6. An Example Program

We now present a complete, working, example NeWS program using the ‘‘lite’’ toolkit. It fills a window with a fan of lines (of varying colors on a color display). The complete text is shown in the appendix; we now examine the parts in detail.

6.1. Overall Structure

The overall structure of this, and many other simple programs, is:

```
#! /usr/NeWS/bin/psh
        PostScript program
```

In other words, its a Unix script. The POSTSCRIPT program is handed as input to psh, a program that simply establishes a connection to the NeWS server and sends its standard input across for the server to execute. One major breakthrough that NeWS provides is precisely this; it is a window system programmable at a shell-like level.

6.2. Paint Method

The program is going to create an object of whatever class is currently named by DefaultWindow. This window object needs a method to paint its window, and a method to paint its icon. In this case (and many others) they can be the same. The system arranges for the Window or the Icon canvas to be the current canvas when the method is invoked.

```
/fillcanvaswithlines {                                  % linesperside => -
        gsave
        1 fillcanvas                                    % paint the background
        0 setgray                                       % default color is black
        clippath pathbbox
        scale pop pop                                   % make coords 0 to 1
        0 1 3 -1 roll div 1 {                           % 0 delta 1 {..} for
                ColorDisplay? {dup 1 1 sethsbcolor} if  % change color if needed
                0 0 moveto 1 1 index lineto stroke      % draw line to top
                0 0 moveto 1 lineto stroke              % draw line to side
                pause                                   % let others run
        } for
        grestore
} def
```

This is just a loop that draws a pair of lines each time. It also does a pause, because the window object will fork a new lightweight process to execute the paint method. The pause ensures that, even if painting the window takes a long time it will not prevent other processes from running.

6.3. Menu

The user of this application needs a way of telling the program how many lines to draw. A menu that pops up over the window is the answer. We set this up by:

- Defining an initial value for the number of lines.
- Defining a procedure that gets invoked by the menu when an key is selected. It converts the currently selected key from a string to a number, updates the number of lines, and the updates the image by

sending a /paintclient message to the window object.

- Creating a new object of whatever class is named by DefaultMenu, giving it a list of keys, and the procedure we just defined as the procedure to be executed when one of the keys is selected.

```
/linesperside 10 def                          % start with 10 lines
/setlinesfromuser {                           % value => -
        /linesperside exch store              % set new value
        /paintclient win send                 % make window repaint
} def
.....................
/ClientMenu                                   % the menu sets linesperside
        [ (10) (20) (100) (250) (500) ]
        [ {currentkey cvi setlinesfromuser} ]
        /new DefaultMenu send def
```

When a menu is activated, the toolkit arranges for the menu code to run in a separate POSTSCRIPT process. Thus, other processes can run, even the process painting the lines display, while the menu is active. The application is not aware that this is being done.

6.4. Window

This application creates its own sub-class of whatever is the current default Window class. The only thing that is different is that objects of class LinesWindow have a sub-canvas of the ClientCanvas to draw in. The reason for doing this will become obvious when we consider Items below.

```
/LinesWindow DefaultWindow                    % new subclass of DefaultWindow
[/LinesCanvas]                                % instance var: the subwindow
classbegin                                    % override 2 methods
        /CreateClientCanvas {                 % this one creates the canvas
                /CreateClientCanvas super send    % do super's create
                /LinesCanvas ClientCanvas
                newcanvas store                   % create a subcanvas
                LinesCanvas /Mapped true put      % map it in
        } def
        /ShapeClientCanvas {                  % This one (re)-shapes it
                /ShapeClientCanvas super send     % do super's shape
                gsave
                ClientCanvas setcanvas clippath   % make path and
                LinesCanvas reshapecanvas         % reshape lines canvas
                grestore
        } def
classend def                                  % call new class LinesWindow
```

The application needs a window to run in. We create an object of class LinesWindow, and send it a procedure. The effect of this is to have the procedure executed in the context of the window object. The procedure re-defines those attributes and methods of the generic window object that are appropriate for this application, namely:

- FrameLabel – the string displayed in the window frame.
- PaintClient – the method for painting the window. This is the fillnameswithlines procedure we defined earlier.
- PaintIcon – we use the same procedure in the icon painting method, but only draw 10 lines. Drawing too many lines in a small icon doesn't look good.
- ClientMenu – as described above, we create a menu object and assign it as the menu to be displayed in the window.

Then, when we have customised the generic window object for this application, we get it some real estate on the screen by sending it the /reshapefromuser message. The window object has inherited the method for this from the environment; typically it involves dragging out a rectangle on the screen.

```
/win framebuffer /new LinesWindow send def          % make a new LinesWindow
{
        /FrameLabel (Lines) def                      % Label it Lines
        /PaintClient {                               % PaintClient method
                LinesCanvas setcanvas                % fills LinesCanvas
                linesperside fillcanvaswithlines     %  with linesperside lines
                FrameBorderColor strokecanvas        % and draws a box
        } def
        /PaintIcon {                                 % PaintIcon method
                10 fillcanvaswithlines 0 strokecanvas % uses 10 lines
        } def
        /ClientMenu                                  % the menu sets linesperside
        [ (10) (20) (100) (250) (500) ]
        [ {currentkey cvi setlinesfromuser} ]
        /new DefaultMenu send def
} win send                                           % override the methods
/reshapefromuser win send                            % shape the window
/map win send                                        % Map it in
```

Finally, we send the window a /map message to get it to display itself in the selected area. The window inserts itself in the window hierarchy, and the NeWS server generates a /Damaged event for its canvas. The window object has forked a process listening for these events, when they are detected the PaintClient or PaintIcon method is invoked as appropriate.

In general, window classes arrange to run their PaintClient methods in a separate process. If the PaintClient method is invoked again, it checks to see whether a preceding PaintClient invocation is still underway. If it is, it is aborted before a new process is forked for the re-paint. In this way, windows respond immediately to being re-shaped; there is no need to wait for completion of one re-shape before starting another. Note that this use of concurrency is invisible to the PaintClient method itself, it depends only on the occasional pause.

6.5. Item

As an example of the use of Item objects, we can add an alternative way of setting the number of lines to be displayed − a SliderItem across the bottom of the window. This needs the following changes:

- Add a new instance variable to the LinseWindow class to point to the SliderItem object.
- Change the CreateClientCanvas method to create a SliderItem and position it at the bottom of the window.
- Change the ShapeClientCanvas method to position the slider and restrict the LinesCanvas not to overlap it. (Now it is obvious why we used a separate canvas).
- Change the PaintClient method to send a paint message to the slider object.
- Change setlinesfromuser to update the value of the SliderItem, so that if the menu is used to change the number of lines, the slider will move to correspond.

Here is the new definition of the LinesWindow class:

```
/LinesWindow DefaultWindow                          % new subclass of DefaultWindow
[/LinesCanvas /LinesItem]                           % instance var: the subwindow
classbegin                                          % override 2 methods
/LinesItemHeight 30 def
    /LinesInset  10 def
        /CreateClientCanvas {                       % this one creates the canvas
                /CreateClientCanvas super send      % do super's create
                /LinesItem (Lines:) [1 500 linesperside]
                /Right {ItemValue 10 mul setlinesfromuser}
                ClientCanvas 500 0 /new SliderItem send% make a slider
                dup /ItemFrame 2 put
                10 2 /move 3 index send store       % position slider
                [LinesItem] forkitems pop           % fork process for slider
                /LinesCanvas ClientCanvas
                newcanvas store                     % create a subcanvas
                LinesCanvas /Mapped true put        % map it in
        } def
        /ShapeClientCanvas {                        % This one (re)-shapes it
                /ShapeClientCanvas super send       % do super's shape
                10 2 /move LinesItem send
                gsave
                ClientCanvas setcanvas clippath     % make path and
                pathbbox LinesInset LinesItemHeight translate
                LinesItemHeight sub LinesInset sub
                exch LinesInset 2 mul sub exch rectpath % leave room for slider
                LinesCanvas reshapecanvas           % reshape lines canvas
                grestore
        } def
classend def                                        % call new class LinesWindow
```

The setlinesfromuser procedure now looks like:

```
/setlinesfromuser {                                 % value => -
        /linesperside exch store                    % set new value
        {
                LinesItem /ItemValue
                linesperside put
        } win send                                  % update slider value
/paintclient win send                               % make window repaint
} def
```

The PaintClient method now looks like:

```
/PaintClient {                                      % PaintClient method
        /paint LinesItem send                       % paint the slider
        LinesCanvas setcanvas                       % fills LinesCanvas
        linesperside fillcanvaswithlines            %  with linesperside lines
        FrameBorderColor strokecanvas               % and draws a box
} def
```

Note once again the use of concurrency. The SliderItem is driven by a separate process, created in the forkitems call. This means that the user can drag the slider to a new value at any time, even while the lines display is painting itself. And, as soon as a new value is indicated by the slider or the menu, the lines display is re-painted, even if another drawing attempt has to be aborted to do so. And, best of all, the application code need not be aware of any of this. Simply providing objects with appropriate behaviours is enough, the toolkit takes care of providing concurrency where it is needed.

7. Review

The example program specifies exactly those attributes and methods that it requires. These include:

- The method for painting the contents of the window and icon.
- The keys and corresponding procedures for the menu.
- The label for the frame.
- The position, label, and scale of the slider.

Everything else is inherited from the generic Window, Menu and Slider objects. And, unlike library-based toolkits, it is inherited at run-time. This means that the default behaviour of any of these objects can be changed by the user, with no need to modify the application at all.

Further, note that the only way in which the example program needs to acknowledge the concurrency implicit in the NeWS lightweight process mechanism is in the disciplined use of pause to cede to other processes at intervals. Providing the objects with appropriate methods is enough, the toolkit deals with invoking these methods in separate processes when that is required. The result is an application that deals correctly with multiple things going on at one time, with almost no programming effort.

8. Conclusion

We have shown that, in addition to its role as a page description language, POSTSCRIPT is capable of supporting an object-oriented programming style. Despite being implemented entirely in standard interpreted POSTSCRIPT, the style is efficient enough to be used as the basis for a user interface toolkit.

The ‘‘lite’’ toolkit illustrates the use of the NeWS extensions to PostScript, primarily lightweight processes, to construct user interfaces that deal correctly with many concurrency issues without complex programming.

We are still at any early stage in the development of NeWS-based user interfaces. Much work remains to be done, and among the issues to be addressed are:

- Better performance for the class mechanism, through implementing send as a primitive.
- Exploration of alternate class hierarchies.
- Improved integration between the POSTSCRIPT running in the NeWS server and the C (or other language) programs running in the client.

References

1. J. H. Morris, M. Satyanarayanan, and others, ‘‘Andrew: A Distributed Personal Computing Environment,’’ *Commun. Assoc. Comput. Mach.*, vol. 29, no. 3, pp. 184-201, March, 1986.
2. R. W. Scheifler and J. Gettys, ‘‘The X Window System,’’ *ACM Trans. on Graphics*, vol. 6, no. 3, 1986.
3. Sun Microsystems, ‘‘NeWS Technical Overview,’’ 800-1498-05, April 1987.
4. Adobe Systems Inc., *PostScript Language Reference Manual,* Addison Wesley, July, 1985.
5. J. Warnock and D. K. Wyatt, ‘‘A Device Independent Graphics Imaging Model for Use with Raster Devices,’’ *Computer Graphics*, vol. 16, no. 3, pp. 313-320, July, 1982.
6. Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh,* Hayden Book Co., 1986.
7. L. Cardelli and R. Pike, ‘‘Squeak - A Language for Communicating with Mice,’’ *Computer Graphics*, vol. 19, no. 3, pp. 199-204, July 1985.
8. S. Evans, ‘‘The Notifier,’’ in *Proc. USENIX Summer ’86 Conf.*, Atlanta GA, 1986.
9. O. M. Densmore, ‘‘Object-Oriented Programming in NeWS,’’ in *Proc. 3rd USENIX Computer Graphics Workshop*, Monterey, CA, November, 1986.

Appendix: the example program

```
#! /usr/NeWS/bin/psh
/fillcanvaswithlines {                                  % linesperside => -
        gsave
        1 fillcanvas                                    % paint the background
        0 setgray                                       % default color is black
        clippath pathbbox
        scale pop pop                                   % make coords 0 to 1
        0 1 3 -1 roll div 1 {                           % 0 delta 1 {..} for
                ColorDisplay? {dup 1 1 sethsbcolor} if  % change color if needed
                0 0 moveto 1 1 index lineto stroke      % draw line to top
                0 0 moveto 1 lineto stroke              % draw line to side
                pause                                   % let others run
        } for
        grestore
} def
/main {
        /linesperside 10 def                            % start with 10 lines
        /setlinesfromuser {                             % value => -
                /linesperside exch store                % set new value
                /paintclient win send                   % make window repaint
        } def

        /LinesWindow DefaultWindow                      % new subclass of DefaultWindow
        [/LinesCanvas]                                  % instance var: the subwindow
        classbegin                                      % override 2 methods
                /CreateClientCanvas {                   % this one creates the canvas
                        /CreateClientCanvas super send  % do super's create
                        /LinesCanvas ClientCanvas
                        newcanvas store                 % create a subcanvas
                        LinesCanvas /Mapped true put    % map it in
                } def
                /ShapeClientCanvas {                    % This one (re)-shapes it
                        /ShapeClientCanvas super send   % do super's shape
                        gsave
                        ClientCanvas setcanvas clippath % make path and
                        LinesCanvas reshapecanvas       % reshape lines canvas
                        grestore
                } def
        classend def                                    % call new class LinesWindow

        /win framebuffer /new LinesWindow send def      % make a new LinesWindow
        {
                /FrameLabel (Lines) def                 % Label it Lines
                /PaintClient {                          % PaintClient method
                        LinesCanvas setcanvas           % fills LinesCanvas
                        linesperside fillcanvaswithlines %  with linesperside lines
                        FrameBorderColor strokecanvas   % and draws a box
                } def
                /PaintIcon {                            % PaintIcon method
                        10 fillcanvaswithlines 0 strokecanvas % uses 10 lines
                } def
                /ClientMenu                             % the menu sets linesperside
                [ (10) (20) (100) (250) (500) ]
                [ {currentkey cvi setlinesfromuser} ]
                /new DefaultMenu send def
        } win send                                      % override the methods
        /reshapefromuser win send                       % shape the window
        /map win send                                   % Map it in
} def

main
```