The NeWS window system: A look under the hood

James Gosling

Introduction

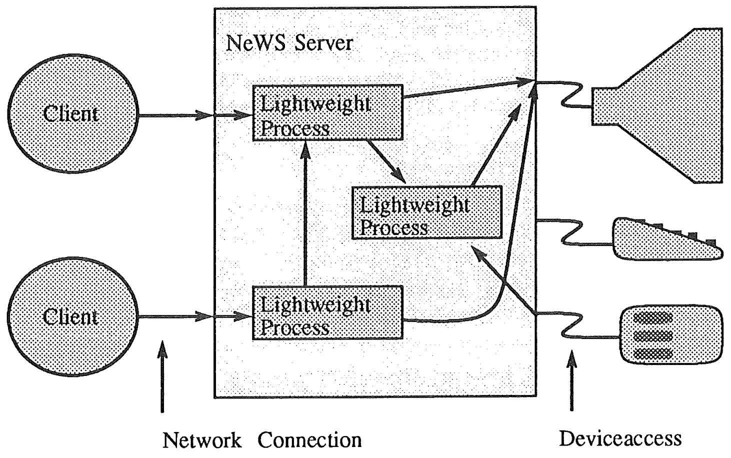
With the standards furor over the X11 window system, many people ask "why is Sun bothering with NeWS?". There are two answers: one is that PostScript† is an important standard too, although its importance is concentrated outside of the traditional workstation market. The other is that there are a number of important technical advantages to NeWS.

This paper starts with a brief overview of NeWS, so that those unfamiliar with it can understand the rest of the paper. This is followed by a section on living with PostScript that covers the use of object-oriented programming and rapid prototyping. Then there is a discussion of the performance implications of downloading programs, followed by a discussion of the advantages of having a close correspondence between the capabilities of a screen and printers. Finally, there is a discussion of one of the many ways that PostScript can be used to enhance X11: fonts.

1. The Structure of NeWS

The NeWS server resembles a self-contained operating system. It contains a set of concurrent processes that are independently and cooperatively executing. The server communicates with application programs through a variety of network connections, and with devices through the usual Unix device driver interface. The processes act as intermediaries between applications and devices, implementing the various tasks that go toward driving the user interface

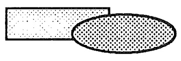
The code that is executing in these processes is dynamically loaded into the server either from the file system or from a client process through a network connection. It is all written in an extended version of the PostScript language. The server is missing many of the features found in other window systems (windows, for example). These are implemented in PostScript, on top of the mechanisms



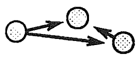
† PostScript is a registered trademark of Adobe Systems Inc.

provided by the server. This extensibility is not something that was added on, it is the central theme. The extension facilities are clean and protected, they provide mechanism, but not policy.

The PostScript language was originally designed for driving printers. For it to be useful in a window system, we needed to add a few extensions:



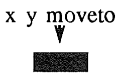
Multiple drawing surfaces (canvases.) In PostScript, there is only one drawing surface: the printed page. NeWS creates the illusion of multiple drawing surfaces. These can be composed in hierarchies, reshaped, and moved around. They are the basic building block out of which windows are built.



Lightweight processes. These are processes that are cheap to create, use relatively little storage, and share the same address space. The processes within NeWS handle requests from applications, deal with input translation, handle animations like menu highlighting, and perform an assortment of other tasks.

Garbage collection. NeWS has a true garbage collector, rather than the simple heap pointer reset in printer PostScript. PostScript code doesn't have to worry about freeing unused objects or placing gsave/grestore pairs appropriately: objects that are no longer referenced disappear automatically.

Input events. Printers only have to deal with output, while a window system has to deal with input as well. NeWS defines *events* which are messages that get sent from process to process. The keyboard an mouse appear as fictitious external processes that generate events when keys or buttons go up or down, and when the mouse moves.

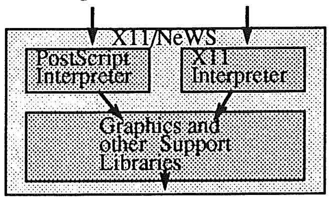


Encoded input stream. In the original PostScript specification, programs could only be written as strings of 7-bit ASCII text. This has significant negative performance implications. NeWS defines a compressed, pre-compiled, representation for PostScript code that is much more efficient to read.

One of the major attractions of PostScript is its approach to device independence. There is a lot of variability amongst displays: B/W bitmap, greyscale, color and an assortment of resolutions. We wanted to avoid a "lowest common denominator" approach. The PostScript imaging model doesn't say anything about pixels, it's a resolution independent description of the image. It is at a high enough level of abstraction to be able to make use of high performance displays.

There are two kinds of device independence: In the first, the server defines a high-level model that attempts to cover many different hardware characteristics. Under this kind of model, user programs are generally oblivious to the hardware characteristics, but they can inquire and adapt if they choose. In the second kind of device independence, the server defines many low-level models, providing different ones for different kinds of hardware. This is simple to implement, but it places a greater burden on application programs and it restricts the applicability of exotic hardware.

NeWS has been merged with X11, so that applications written for either window system can be run concurrently. This easiest way to understand how this works is to think of X11 as an interpreter for a simple language: the X11 protocol. The NeWS server was restructured to allow multiple interpreters to reside within it. They all sit on top of the same set of graphics and support libraries.



2. Classes and Object-Oriented Programming

PostScript is one of the world's worst programming languages. It was designed as a language in which programs are written by other programs, not by people. To a large extent, this problem has been dealt with in NeWS by adding facilities for object-oriented programming, which provide an easy way to define an manipulate objects in PostScript

The easiest way to understand this is to walk through a simple example:

```
/ConsoleWatchBag AbsoluteBag []
      classbegin
            /PaintCanvas { ... } def
      classend def
```

This defines a class called ConsoleWatchBag to be a subclass of a class called AbsoluteBag. This defines one method: /PaintCanvas (a method is a procedure that "belongs" to a class). ConsoleWatchBag behaves exactly like AbsoluteBag, except that it has this one different procedure. ConsoleWatchBag is a *subclass* of AbsoluteBag.

```
/win  [ConsoleWatchBag] framebuffer
      /newdefault ClassNoticeFrame
      send def
```

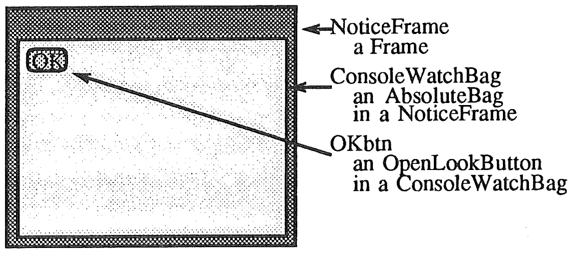
This rather opaque piece of code creates a new instance of ClassNoticeFrame (`/newdefault ClassNoticeFrame send`). This instance creation has a couple of parameters (`[ConsoleWatchBag] framebuffer`). A notice frame is a window that has a client pane within it: in this case, a ConsoleWatchBag.

```
/mybag /client win send def

/OKBtn [ 10 10
      (OK) { pop /unmap win send }
      OpenLookButton
] /addclient mybag send
```

Finally, we create an open-look button labeled OK that will unmap the window when it is pressed.

Here's what this looks like on the screen:



This example has defined a new kind of AbsoluteBag (a canvas that can be drawn on), that has a special method for painting its contents. Then we created a window that contained one of these, and we created and installed a button in it. This is essentially the complete implementation of a Macintosh-style confirmer.

The class paradigm followed in NeWS follows very closely that of Smalltalk. We have made a couple of extensions. Chief among them are multiple inheritance and method promotion. In multiple inheritance, a class can be a subclass of more than one class. This effectively lets you mix together the behaviors of several classes in one class. In method promotion, the definition of a method can be moved dynamically up and down the class hierarchy. For example, if we had a method that computed the area of instances of a class, for any specific instance of that class (an object) we could promote that procedure to be one that simply returned a constant, rather than performing a computation based on constant values.

PostScript turns out to be a very nice language in which to implement object oriented programming. All of the facilities present in NeWS can be implemented in pure PostScript. An instance is just a dictionary, which contains a reference to its class. A class is just a dictionary that contains an array of its superclasses. Entering a class instance (sending a method) just pushes the object, it's class, and its super classes onto the dictionary stack.

3. Classes and Rapid Prototyping

Since PostScript is an interpreted language code can be developed interactively and incrementally.

There is a utility called *psh* (the PostScript shell) that allows developers to interact directly with the PostScript interpreter, having code executed as it's typed, with the results visible immediately. When this is coupled with the class-based approach, for defining objects, new kinds of objects can be built easily based on others. This encourages an experimental approach that allows interfaces to be rapidly built up out of standard components.

There is a rich standard set of objects to start from. This includes blank drawing surfaces, menus, pop-ups, scrollbars, buttons, button stacks, dials, sliders, labels, and text items. There are many variations of each available.

Many facets of the application and its interface can be separated, allowing a range of changes to the interface which don't affect the application. By doing this outside of the application, applications written in any language can take advantage of it.

4. Printer Correspondence: making applications easier

For many applications, PostScript language compatibility isn't nearly as important as compatibility with the PostScript imaging model. They need to be able to draw a string on the screen and *know* that it will match the behavior of the printer. This is necessary for applications that want to implement WYSIWYG (what you see is what you get).

Applications written for window systems other than NeWS have to go through a rather laborious process of calculating what the printer would do, calculating what the window system will do, comparing the two, and compensating. NeWS removes the burden of ensuring that text on the screen matches text on the page. This lets application writers concentrate on the task at hand, rather than on how to reconcile the printer and the display.

Another source of problems is the power of the imaging model. Printers today, primarily PostScript printers, have a very sophisticated imaging model. They can scale and rotate arbitrary objects, including text and images. Many applications put a lot of effort into graphical rendering because the underlying system isn't powerful enough. By supporting the full PostScript model, NeWS applications avoid this.

Above and beyond the imaging model provided my most window systems, NeWS provides:

- Curves. In conventional systems, the boundaries of polygons are restricted to being straight lines. PostScript allows them to include arcs of ellipses and cubic Beziers.
- Image manipulation. PostScript can apply arbitrary transformations to images and can convert them from one form to another. For example, a full color image can be scaled by 75%, rotated 45 degrees, and dithered or halftoned to black and white.
- Arbitrary transformations. Transformations can be applied to *anything*. This includes simple graphics objects like vectors and circles, but also includes text and images.

As a side-effect of having PostScript in the window system it is easy to drive printers. A printed page is just a large monochrome image. It's a trivial matter to get NeWS to create such a large monochrome image, render a page to it, and write that page out. There are a number of printers on the market that can connect to a SCSI port and will print monochrome bitmap images sent to them. It is a "simple matter of programming" to write a program to transfer these images out to the SCSI port. This is made particularly easy by a new NeWS facility that allows an image to be in memory shared between an application and the server. Using this technique for printing guarantees WYSIWYG.

5. NeWS helping X11: fonts

NeWS has a very large standard font library:

AvantGarde-Book	Lucida-Bright
AvantGarde-BookOblique	Lucida-BrightDemiBold
AvantGarde-Demi	Lucida-BrightDemiBoldItalic
AvantGarde-DemiOblique	Lucida-BrightItalic
Bembo-Bold	LucidaSans-Bold
Bembo-BoldItalic	LucidaSans-BoldItalic
Bembo-Italic	LucidaSans-Italic
Bembo	LucidaSans
Bookman-Demi	LucidaSansTypewriter-Bold

Bookman-DemiItalic
Bookman-Light
Bookman-LightItalic
Courier-Bold
Courier-BoldOblique
Courier-Oblique
Courier
GillSans-Bold
GillSans-BoldItalic
GillSans-Italic
GillSans
Helvetica-Bold
Helvetica-BoldOblique
Helvetica-Narrow-Bold
Helvetica-Narrow-BoldOblique
Helvetica-Narrow-Oblique
Helvetica-Narrow
Helvetica-Oblique
Helvetica
LucidaSansTypewriter
NewCenturySchlbk-Bold
NewCenturySchlbk-BoldItalic
NewCenturySchlbk-Italic
NewCenturySchlbk-Roman
Palatino-Bold
Palatino-BoldItalic
Palatino-Italic
Palatino-Roman
Rockwell-Bold
Rockwell-BoldItalic
Rockwell-Italic
Rockwell
Symbol
Times-Bold
Times-BoldItalic
Times-Italic
Times-Roman
ZapfChancery-MediumItalic
ZapfDingbats

From PostScript, these fonts can all be scaled and rotated arbitrarily. In the X11/NeWS merge, support for these sophisticated fonts "leaks over" into X11. All NeWS fonts, including Folio and user-defined PostScript fonts are available to X applications as ordinary X fonts under X11/NeWS. We didn't have to extend X in any way, although we did have to take a few liberties with the definition of font names.

There is an "underground" X font name standard that we follow: "name*size*", where the size is a string of digits that trail the name. For example, "screen10" has a name of "screen" and a size of "10". When we execute the X open font request, we effectively execute:

```
/name findfont size scalefont
```

The font is looked up using the standard PostScript findfont operator and is scaled using the PostScript operator.

We also support the Logical Font Description (LFD) proposed X11 extension. According to this proposal, font names are long strings that can be decomposed into many fields. Here is an example of one:

```
/-adobe-helvetica-medium-o-normal--14-140-75-75-p-78-iso8859-1
```

We parse this name and calculate the font family, what encoding to use, and how it should be scaled. It effectively becomes the PostScript sequence:

```
/Helvetica-Oblique findfont
[14*75/72 0 0 14*75/72 0 0] makefont
/iso8850-1 encodefont
```

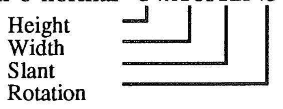
These fonts aggravate a number of generic X11 problems that applications run into.

- Ascent/Descent. X11 fonts have a pair of fields called their ascent and descent. According to the protocol specification, it is legal for characters to be taller than the ascent, or to extend lower than the decent. With many fonts, this actually happens: Applications want the ascent and descent fields to be tight, so that lines aren't too far apart. But the accents on upper case characters can extend rather high (as in the Icelandic spelling of Iceland: Ísland). The X ImageText primitive, which most terminal emulators use, depends on characters fitting within the ascent/descent of the font. This makes internationalization awkward.
- X11 requires that the width of a character be an integer number of pixels, and that it only has an *x* component. This makes it impossible to match the printer exactly, since when character widths are scaled, they are unlikely to be exact integers. We round character widths to integers for X, but not for NeWS, unless a NeWS application explicitly asks for it. This also makes rotated text impossible

- Another problem is with XListFonts. This X request is supposed to return to the application a list of all the fonts available. With scalable fonts, this list is infinitely long. We compromise by replying with a list of the fonts that have and-tuned bitmaps, and with LFD names with the size fields set to zero. This use of zero to indicate scalability is currently a "proposed extension to the proposed extension".

There are a number of extensions to the X text model that we would like to propose and implement. Based on the mechanisms available through PostScript, these are trivial to implement. We could overload the *size* field of an LFD name to indicate both x and y scaling, skew, and rotation. With the exception of rotation, these fit into X11's existing text model. To implement rotation we'd either have to return useless width values, or define a new request that returned *x* and *y* values with subpixel precision.

/-adobe-helvetica-medium-o-normal--14x16s12r45-140-75-75-p-78-iso8859-1



It turns out that we can actually do all of these in X11/NeWS today by defining new fonts from the PostScript side. For example, if we wanted to create a version of Times-Roman that is rotated 90 degrees, we could execute the following PostScript code:

```
FontDirectory
  /Times-Roman findfont
  [ 0 1 -1 0 0 0 ] makefont
/Times-Rotated put
```

This looks up the Times-Roman font, transforms it by using the makefont operator, and then installs it into the font directory. This new font, Times-Rotated, now appears like any other X font. Text drawn with it will be rotated 90 degrees. If the X application inquires the widths of the characters, it will get 0 for all of them, since their *x* widths are 0. The *y* components are non-zero, but there's no way to report that to an X application.