A Comparison of DISPLAY POSTSCRIPT™ and NeWS™

DRAFT of January 5, 1989

Samuel J. Leffler

1612 Oxford St.
Berkeley, CA 94709

ABSTRACT

This document compares the DISPLAY POSTSCRIPT system from Adobe with Sun Microsystems' Network Extensible Window System (NeWS). Both systems provide display functionality based on the POSTSCRIPT language. However, while NeWS provides an integrated, window-oriented framework that includes support for overlapping drawing surfaces and input events; DISPLAY POSTSCRIPT limits its definition mainly to display capabilities. Nonetheless, both systems incorporate significant extensions to the POSTSCRIPT language for common purposes. These extensions are discussed, and the different approaches are compared.

The reader is presumed to be familiar with C, POSTSCRIPT, and NeWS.

1. Introduction

The POSTSCRIPT programming language is used to describe the makeup and presentation of 2-D images. The most common application of the POSTSCRIPT language is in page markup and printing; e.g. as the page description language for the Apple LaserWriter™ printer.

The POSTSCRIPT language, with extensions, is the basis for the Network Extensible Window System (NeWS™), a server-based window system developed by Sun Microsystems[1]. NeWS differs from other network-based window systems, in that it allows clients to transparently extend the services provided by the window server simply by downloading POSTSCRIPT code into the server. This extension mechanism, combined with additions to the POSTSCRIPT language to support multiple threads of execution and event-based input, make NeWS more a graphics-oriented programming environment than a simple window system. In fact, the basic NeWS server provides no support for the typical window management functions normally expected of a window system. Instead POSTSCRIPT code loaded into the server at the time it is started up defines a windowing environment in which applications execute. Client applications operate by establishing a network connection to the server and communicating POSTSCRIPT code that is to be

1 In this document NeWS refers to the 1.1 release unless explicitly stated.

executed. The execution of client supplied POSTSCRIPT code can be used to draw images on the display, alter the operating environment within the server, or provide auxiliary services such as debugging. NeWS consists of a server program, client library for C-based applications, and the **cps** program, a utility used in the development of C-based client applications.

The DISPLAY POSTSCRIPT™ system from Adobe provides "a standard level of display functionality that is fully compatible with POSTSCRIPT language printers."[2] DISPLAY POSTSCRIPT includes a POSTSCRIPT language interpreter, client library for communicating with the interpreter, and the **pswrap** program that can be used in the development of C-based client applications. The POSTSCRIPT interpreter that is part of the DISPLAY POSTSCRIPT system includes extensions to the POSTSCRIPT language that were added explicitly for raster-based display devices. These extensions, together with extensions to support multiple cooperating users of a display, are designed to be *embedded* in a window-oriented environment. That is, the DISPLAY POSTSCRIPT interpreter supports only the imaging requirements of a window system; the other facilities typically associated with a window system (input handling, communication services, overlapping drawing surfaces, etc.) are explicitly not defined within DISPLAY POSTSCRIPT. This is contrasted with NeWS which contains extensions that make it a fully-functional and self-sufficient window system. Because DISPLAY POSTSCRIPT limits its definition, a comparison of it with NeWS is necessarily limited to those areas where they overlap. A comparison of the window-system related extensions present in NeWS must be made with a system derived from DISPLAY POSTSCRIPT (such as the system being developed at NeXT).

The remainder of this document describes the important concepts and facilities present in DISPLAY POSTSCRIPT, and compares them with the similar facilities found in NeWS. Section 2 describes the POSTSCRIPT extensions present in DISPLAY POSTSCRIPT, while section 3 gives an overview of the tools provided for programming in C. Finally, section 4 summarizes our results and suggests areas where NeWS and DISPLAY POSTSCRIPT would benefit from a consolidation of facilities.

In this document, program names appear **emboldened**, C code appears in a `constant width font`, and POSTSCRIPT code appears in a Helvetica font, or when in-line, in **Helvetica Bold**.

2. POSTSCRIPT Language Extensions

The extensions to the POSTSCRIPT language that have been defined to create DISPLAY POSTSCRIPT are very similar to the extensions made to create NeWS. In particular the following additions exist in each, albeit in different forms:

- support for multiple threads of execution in the POSTSCRIPT interpreter,
- synchronization mechanisms for threads running in the interpreter,
- extensions to the virtual memory model to support garbage collection,
- an alternative binary encoding for the input stream,

[2] ''The DISPLAY POSTSCRIPT System Reference, Alpha Version.'' Adobe Systems Inc. October 10, 1988.

- an additional clipping facility above the *clip path*,
- support for bitmap fonts, and
- the ability to manage graphics contexts as objects.

In addition, Adobe has taken the opportunity to add some *optimization* operators. Some of these operators can be easily emulated in old interpreters, while others are rather difficult to completely emulate. The optimization operators fall into three main categories:

- rectangle operators,
- user paths, and
- additional text support.

It should be noted that NeWS also includes extensions to the POSTSCRIPT language that are not found in printers. Most of these extensions, however, were not added for optimization purposes, but rather to support new facilities such as canvases, processes, or input handling.

DISPLAY POSTSCRIPT also incorporates several additions to the POSTSCRIPT language that have been made in the past few years, but which have not been widely publicized. These facilities are intended to be a standard part of all POSTSCRIPT interpreters:

- packed arrays,
- immediately evaluated names, and
- better font cache control.

Finally, somewhere between these standard features and the definition of DISPLAY POSTSCRIPT, Adobe also added in support for color printers.

2.1. Binary Stream Encoding

The original POSTSCRIPT definition defines only an ASCII encoding of the input stream. Numbers are presented as ASCII strings that the interpreter converts to an internal machine representation.

NeWS extended the ASCII encoding to support a more compact encoding based on *binary tokens*. With this scheme, whenever a byte with the top-most bit set is encountered by the POSTSCRIPT scanner, a token is scanned by interpreting some number of subsequent bytes according to a binary encoding scheme. The encoding scheme directly supports binary transmission of numbers (integer, fixed point, and single and double precision floating point) and strings. In addition, POSTSCRIPT objects may be placed in a *system object table* or in a per-connection *user object table* and subsequently referenced in an efficient manner.

DISPLAY POSTSCRIPT defines two encoding forms, a *tokenized* form that is very similar to the encoding used by NeWS, and a less space efficient *binary object sequence* format that is designed to be very efficient to interpret. (The latter form is used by the **pswrap** program to encode executable arrays—see Section 3.) Like NeWS, the DISPLAY POSTSCRIPT encoding supports the mixing of ASCII and binary encoded data.

The important differences between the two encoding schemes are:

- DISPLAY POSTSCRIPT supports both little- and big-endian byte orderings in a single stream while NeWS ordains a single (big-endian) byte ordering,

- DISPLAY POSTSCRIPT supports both IEEE and *native* floating point formats while NeWS requires that all floating point data be transmitted in IEEE format,
- DISPLAY POSTSCRIPT supports a compact encoding of homogeneous arrays of numbers,
- DISPLAY POSTSCRIPT encodes the literal/executable attribute of an object fetched from a system or user name table in an encoded token,
- DISPLAY POSTSCRIPT does *not* support double precision floating point values, and
- DISPLAY POSTSCRIPT limits encoded strings to at most 64 kilobytes.

Having the server handle various byte orderings and floating point formats shifts a burden from the client to the server. That is, clients can usually encode data using their native machine formats and expect the server to handle the decoding. This is preferable to the scheme used by NeWS, particularly when a machine does not support IEEE floating point, or when the client and server machines use a little-endian byte ordering. It would be best to use a onetime negotiation phase to define the byte ordering and floating point format, but this would disallow the transparent concatenation of data generated on varying machine architectures.

Encoding the literal/executable attribute in an encoded token permits an application to reference an entry in a name table as either an executable name or a literal name. This avoids having to define two entries in a table, one for the executable form of the name and one for the literal form.

The lack of a double precision floating point encoding in DISPLAY POSTSCRIPT restricts the encoding's applicability to systems that want to add extensions, such as 3-D operators, that are embedded in the same stream.

In both DISPLAY POSTSCRIPT and NeWS, the encoding scheme can also be applied to output from the interpreter to clients. In DISPLAY POSTSCRIPT, two new operators, **printobject** and **writeobject**, write an object's value to a file. An accompanying *tag* (supplied as an argument) precedes the data to identify the intended recipient of the data. Several tags are reserved for transmitting errors from the interpreter to a client. NeWS has similar facilities in the **tagprint** and **typedprint** operators, although errors are not handled for a client.

2.2. Encoded Number Strings

DISPLAY POSTSCRIPT includes the notion of an *encoded number string*. This is not strictly related to the binary encoding of the input stream, although its addition is clearly based on similar requirements. An encoded number string is a compact way of specifying a homogeneous array of numbers; e.g. the following are equivalent in DISPLAY POSTSCRIPT:

```
[ASCII-encoded numbers] rectfill
homogeneous number array rectfill
string rectfill    % encoded number string
```

The representation is both syntactically compact and compact in terms of space. Furthermore, because its contents do not need to be processed by the POSTSCRIPT scanner, it can be processed very efficiently. The down side is that it is supported by only a handful of operators (all added in DISPLAY POSTSCRIPT): **rectfill**, **rectstroke**, **rectclip**,

rectviewclip, **xshow**, **yshow**, and **xyshow**. In addition, encoded user paths (see below) represent their numeric operands as encoded strings.

2.3. Multiplexing Execution

The original POSTSCRIPT definition specifies that the interpreter supports only a single execution context that is scheduled to users in a serial fashion. Associated with this single context is a global uniform virtual memory environment in which objects are stored.

DISPLAY POSTSCRIPT and NeWS support multiple execution contexts within the POSTSCRIPT interpreter. NeWS calls such an object a *lightweight process* (or just process), while DISPLAY POSTSCRIPT identifies this object as a *context*. Each execution context has:

- an independent thread of control,
- a set of stacks: operand, dictionary, execution, and graphics state stacks;
- standard input, output, and error files, and
- miscellaneous state variables.

In NeWS all contexts share a single virtual memory while in DISPLAY POSTSCRIPT each context has a private memory and access to a global shared memory (see below). Additionally, NeWS organizes process/contexts into groups for the purpose of process management. DISPLAY POSTSCRIPT has no similar notion. Table 1 shows the set of process-related operators defined in DISPLAY POSTSCRIPT and NeWS (we use the term *thread* to signify either a context or process).

DPS	NeWS	Description
fork	fork	create new thread of execution
join	waitprocess	wait until specific thread terminates
currentcontext	currentprocess	return handle for current thread
–	killprocess	terminate thread of execution
–	killprocessgroup	terminate group of threads
detach	newprocessgroup	detach current thread from group
yield	pause	suspend current thread momentarily
–	breakpoint	suspend current thread
–	continueprocess	resume suspended thread
–	suspendprocess	suspend specific thread
lock	createmonitor	create synchronization object
monitor	monitor	execute procedure with synchronization
condition	–	create condition object
wait	–	wait for condition to occur
notify	–	resume threads waiting for a condition

Table 1. Thread-related operators.

There are significant similarities between the two systems, as well as annoying differences. For example, the **fork** operator in DISPLAY POSTSCRIPT initializes the operand stack of the new context by popping a collection of objects off the operand stack of the parent context. In NeWS, on the other hand, the **fork** operator simply duplicates the existing operand stack. NeWS provides more control over processes. Processes can be suspended and killed, although in practice the **breakpoint**, **suspendprocess**, and **continueprocess** operators are not used for their intended purpose of debugging because too much of a process state is inaccessible outside the context of a process[3].

The DISPLAY POSTSCRIPT *condition* objects are used in conjunction with *lock* objects for intercontext synchronization and communication. NeWS has a *monitor* object that is virtually identical to a lock, and a more general event mechanism that can be used to emulate locks.

The DISPLAY POSTSCRIPT documentation does not specify how context management relates to connection management. This lack of specification is deliberate; DISPLAY POSTSCRIPT is designed to be used on systems that do not support multi-tasking, e.g. the Apple Macintosh™. In NeWS, the relationship between process management and connection management is defined by a POSTSCRIPT server process that is created at the time the NeWS server is started up. For each client connection to the server, a new POSTSCRIPT process in a new process group is created to service client requests. When a client-server connection terminates, the POSTSCRIPT code that accepted the connection kills off all the processes associated with the process group and then terminates[4]. With DISPLAY POSTSCRIPT however, this scheme is not possible because the notions of connection management, context groups, and terminating contexts do not exist. Instead, the lifetime of a context (whether or not it is terminated when a client-server session is terminated) is determined by the environment in which DISPLAY POSTSCRIPT is embedded. Since DISPLAY POSTSCRIPT does not support a mechanism for terminating a context, one must assume that contexts are automatically reclaimed when a client-server connection closes. This may have ramifications to applications that use shared resources that need to be reclaimed. For example, in NeWS a "watchdog" POSTSCRIPT process can be used to automatically cleanup global resources when a connection is closed. A less important effect of DISPLAY POSTSCRIPT's context/connection management strategy is that, unlike NeWS, it is not possible to write an application entirely in POSTSCRIPT. Without POSTSCRIPT-based support for input, however, applications of this sort would have less utility.

As an independent matter, DISPLAY POSTSCRIPT does not treat a context as a full fledged POSTSCRIPT object. That is, it is not possible access the state of a DISPLAY POSTSCRIPT context given a reference to the context. NeWS, on the other hand, treats a process as a full-fledged POSTSCRIPT object that is accessible in other processes. The NeWS approach permits the state of a POSTSCRIPT process to be interrogated by another POSTSCRIPT process. This is particularly useful when debugging.

2.4. Memory Management

The original POSTSCRIPT definition specifies a single uniform virtual memory. Two operators, **save** and **restore** are used for memory management. The **save** operator logically checkpoints the state of virtual memory, while the **restore** operator rolls back the state of virtual memory to the point at which a previous **save** was done. These operators

[3] Instead a process is debugged with the assistance of a debugging procedure that executes within the context of the stopped process.

[4] These actions are defined by a POSTSCRIPT procedure contained in a file that the NeWS server reads when it is started up. Other actions can be defined by rewriting this POSTSCRIPT procedure.

must be matched. They are intended mainly to be used in a bracketing fashion around each page of a multi-page printout, or to create an encapsulated environment, such as for including illustrations.

In NeWS the **save** and **restore** operators are ignored. Instead memory management is done entirely with a reference counting garbage collection scheme. A new operator **undef** was added to explicitly delete a dictionary entry. Ignoring **save** and **restore** makes it difficult to provide an encapsulated environment for subdocuments because storage for objects such as user-defined fonts is not reclaimed.

DISPLAY POSTSCRIPT also includes a garbage collection facility, but rather than ignore **save** and **restore** they have been redefined to work in as "compatible a way as possible." This can result in some confusion; for example, when multiple contexts share virtual memory and one executes a **restore** operation. Like NeWS, DISPLAY POSTSCRIPT includes **undef**; in addition, an **undefinefont** operator is available to delete read-only fonts that reside in the **FontDirectory**.

In DISPLAY POSTSCRIPT, the virtual memory model has also been extended to include multiple, potentially independent, virtual memories, each with different lifetimes and visibilities. Each execution context has a *private* virtual memory and access to a single *shared* virtual memory that is visible to all contexts and that can be updated by any context under "suitable conditions." Typically, a DISPLAY POSTSCRIPT application will make use of several execution contexts that all share a common private virtual memory. Separate applications however, typically do not share storage, except through the global shared memory.

The virtual memory scheme defined in DISPLAY POSTSCRIPT can be useful in isolating execution contexts from each other in the same way that hardware memory management facilities isolate executing processes. It can also be applied well in a shared memory multiprocessor environment where a distinction between shared and private virtual memory can make it possible to use local memory for private POSTSCRIPT virtual memory. However, the scheme complicates many common operations by requiring that applications be cognizant of the location of an object's storage. Furthermore, since the POSTSCRIPT operand and dictionary stacks are private to a context anyway, the advantages identified in typical programming languages are not as clear. Experience with NeWS suggests that collisions in storage between execution contexts are rare when language-based schemes, such as the class support provided in NeWS, are used. There are no reported experiences of NeWS being integrated into a shared memory multiprocessor environment. Thus, it is too early to draw any conclusions regarding the the cost of a single shared virtual memory in such an environment.

2.5. View Clipping

POSTSCRIPT defines a *clip path* against which all imaging operators are clipped. This is insufficient in a window-oriented environment where imaging must also be clipped to a window boundary and, at times, to a subregion of a window.

DISPLAY POSTSCRIPT introduces an additional level of clipping with a *view clip path*. This path is designed to be used during damage repair. Clipping to a window boundary is not explicitly supported in DISPLAY POSTSCRIPT; derived systems are expected to provide this. It is unclear from the documentation where the view clip path comes from; presumably it is provided by the window system as a result of damage to a window.

In NeWS clipping to the window boundary is part of the *canvas* object in which all imaging operations are defined to take place. Damage repair is effected with a *damage path* that is automatically provided to each damaged canvas along with an appropriate event. NeWS also allows applications to extend the damage path to force damage repair in regions the window system can not know about.

2.6. Font Related Extensions

The font machinery defined in POSTSCRIPT is probably the main reason for its success. Fonts are defined in terms of curve outlines that may be stroked or filled to image characters. Individual characters or entire fonts can be scaled and rotated as required. For users of interactive display technology, these capabilities are a significant advance over the usual fixed size, hand tuned, bitmap fonts that cannot be scaled or rotated.

NeWS (in its current incarnation) provides only a subset of the full font capabilities defined in POSTSCRIPT. All the standard operators are supported, including stroke fonts, but the fonts are not developed enough to make them useful. In particular, there are problems with rotating fonts, and most fonts do not scale well because they are represented internally not as outlines, but instead as bitmaps[5].

DISPLAY POSTSCRIPT builds on the basic font machinery available in POSTSCRIPT and so supports all the facilities defined in POSTSCRIPT. In addition, DISPLAY POSTSCRIPT defines a collection of new operators for explicitly positioning text and for more efficient font switching. Bitmap fonts have also been added to the the standard font machinery in a transparent fashion that is similar to NeWS.

The new text operators are **xshow**, **yshow**, and **xyshow**. These allow an application to specify explicit x-y positioning information on a character by character basis without using explicit combinations of **moveto** and **show**.

The font switching mechanism added in DISPLAY POSTSCRIPT combines the canonical **findfont**, **scalefont** (or **makefont**), and **setfont** operations into a single **selectfont** operator. This operator also takes advantage of information in the font cache in order to avoid doing a **findfont** whenever possible.

The DISPLAY POSTSCRIPT support for bitmap fonts consists mainly of two new entries in font dictionaries: **BitmapWidths** and **BitmapTransform**. **BitmapWidths** is a boolean value that indicates if bitmap widths are to be used when the device provides bitmaps for the font, or whether scalable widths are to be used. For backwards compatibility, if the entry is not present, the scalable widths are used. **BitmapTransform** is a boolean that specifies if a font's bitmaps should be rotated, or whether rotated characters should be derived from the scalable versions[6]. These entries are not present in NeWS bitmap fonts.

Noticeably missing from the font additions in DISPLAY POSTSCRIPT is support for returning an application font metric information. With the current system, an application can reliably obtain font metric information only by applying the **stringwidth** operator to each glyph in a font. NeWS addresses half the problem by defining a **WidthArray** that contains the width of each character in a font, but does not provide a similar **HeightArray**

[5] The X11/NeWS server appears to have significantly better font machinery.

[6] This is supposed to change in a future release to provide finer-grained control (Hourvitz, private communication; 1988).

for the character heights. DISPLAY POSTSCRIPT explicitly refrains from providing this information through the server; expecting instead that applications will get the information directly from Adobe Font Metric (AFM) files in the filesystem. Relying on AFM files, however, presumes that applications are able to access and parse these files.

2.7. Graphics States

A graphics state in POSTSCRIPT is made up of a large number of items that are accessible only through special purpose operators. 11 pairs of "get" and "set" operators exist, one for each accessible element in the graphics state, and there are additional operators for manipulating the current transformation matrix, current path, clipping path, font, etc. The **gsave** and **grestore** operators allow an application to save and restore the graphics state, but there is no mechanism for manipulating the graphics state in a non-stack-oriented fashion.

In NeWS, graphics states and paths were made POSTSCRIPT objects that could be manipulated. The current graphics state and paths can be saved in virtual memory to be restored at a later time. This facility is independent of the ordering imposed by the graphics state stack and the **gsave–grestore** operators. In addition, the **copy** operator was extended to operate on the new *graphicsstate* and *shape* types. Unfortunately, the paths that can be saved and restored are the *transformed* paths; there are many instances where an untransformed path is desired.

DISPLAY POSTSCRIPT also promotes the notion of the graphics state to an object, defines new operators to set and get the entire graphics state, and extends the **copy** operator to handle graphics states. Additionally, DISPLAY POSTSCRIPT provides an operator that overwrites an existing graphics state object with the current graphics state. A separate facility called *user paths* is defined to address the manipulation of paths; this is described later.

I personally wish that graphics state objects were treated as dictionary-like objects in the same way that NeWS treats other objects such as processes, canvases, and events. This would allow almost two dozen special purpose operators to be replaced by the normal **get** and **put** operators, or an equivalent (such as pushing the object on the dictionary stack). Making the graphics state stack accessible through the process dictionary would also improve the debugging facilities in NeWS. Currently it is necessary to have a surrogate procedure executing inside a process in order to access the graphics state (and even then it is rather painful).

2.8. Line Quality

The scan conversion process associated with stroking a path can produce lines of non-uniform thickness due to rasterization effects. This is caused by quantization effects on path coordinates and line widths. Producing high quality lines on a raster display can be rather expensive and, in an interactive setting, must be controllable by an application.

DISPLAY POSTSCRIPT defines the notion of *automatic stroke adjustment*. When this is enabled, the line width and coordinates of a stroke are automatically adjusted as necessary to produced lines of uniform thickness. Furthermore, the line thickness is made as near as possible to the requested line width (i.e. no more than a half a pixel different). When this is not enabled, lines are rendered without concern for uniformity or quantization effects. Automatic stroke adjustment is a new component of the graphics state that is manipulated with the new **setstrokadjust** and **currentstrokeadjust** operators.

NeWS is less precise in defining how it handles the rasterization process. There is no equivalent to DISPLAY POSTSCRIPT's stroke adjustment control. NeWS does define a graphics state item, *line quality*, which is a number between 0 and 1 that "controls the quality of lines rendered by the **stroke** primitive." Increasing this value increases the line quality at the expense of rendering time. A value of 0 renders lines as fast as possible with the least attention paid to quality. A value of 1 causes lines to be rendered with the highest possible quality: they will have the correct width, and all endcaps and joins will be correct. Intermediate values may give different quality/performance tradeoffs. DISPLAY POSTSCRIPT effectively operates with line quality always set to 1.

2.9. Rectangle Operators

DISPLAY POSTSCRIPT defines a number of new operators that work only with rectangles. Since rectangles are very common in window-oriented applications, their use can, according to the DISPLAY POSTSCRIPT documentation, "*significantly improve performance*". It is unclear, however, just how much better special-purpose rectangle operators are over simply using user paths (described below) and/or recognizing and optimizing the handling of rectangular paths. NeWS defines a similar set of operations as POSTSCRIPT procedures. Table 2 lists the new operators that work either on a single rectangle, or on an infinite series of rectangles.

Operator	Description
rectfill	fill a rectangular path
rectstroke	stroke a rectangular path
rectclip	set a new *clipping path*
rectviewclip	set a new *view clip path*

Table 2. DISPLAY POSTSCRIPT rectangle operators.

2.10. User Paths

DISPLAY POSTSCRIPT introduces the concept of a *user path*, a POSTSCRIPT language procedure consisting entirely of path construction operators and their coordinate operands expressed as literal numbers. A user path is a completely self-contained description of a path in user space. The first "real" component of user path must be a specification for a bounding box around the path[7]. The bounding box information permits the POSTSCRIPT interpreter, in most instances, to cache a bitmap representation of the path when it is stroked or filled. User paths are particularly useful for repetitive operations such as window refresh. A compact representation allows for an efficient implementation, and their restrictive semantics permits them to be interpreted without employing the usual POSTSCRIPT mechanisms. User paths are not part of the standard POSTSCRIPT language found in most printers, but they may be emulated (although a complete emulation may be difficult).

As mentioned above, NeWS provides no equivalent facility. The facilities to save and restore a path work only on the transformed representation of a path, making it far less

[7] A **ucache** request may actually appear first to control whether or not the path is cached.

flexible than a user path. A facility like a user path was not included in NeWS for compatibility reasons[8].

2.11. Color Support

The original POSTSCRIPT definition supports color only through the **setrgbcolor** and **sethsbcolor** operators. These allow an application to set the current drawing color as either a red-green-blue or hue-saturation-brightness triple. This was inadequate, however, in that color images could not be imported, and there was no mechanism for controlling color correction or color halftoning.

NeWS addressed color with several additions. Most importantly, a new POSTSCRIPT type for color was added. A new **currentcolor** operator returns the current drawing color from the graphics state, while the **setcolor** operator sets the current drawing color in the graphics state. These operators return consistent results whether the current drawing color is a full color value or a gray scale value. The **hsbcolor** and **rgbcolor** operators take h-s-b and r-g-b triples and return a color object matching the requested values. This permits the old **setrgbcolor** and **sethsbcolor** operators to be emulated as

```
/setrgbcolor {rgbcolor setcolor} def
/sethsbcolor {hsbcolor setcolor} def
```

Most importantly, since color is a full-fledged object, applications can manipulate color objects without concern for the type of output device (color, grey scale, or monochromatic). For applications that require knowledge of the type of display device being used, it is possible to query a drawing surface to find out if it is associated with a color display (through the boolean **/Color** entry in a canvas's pseudo-dictionary). The **contrastswithcurrent** operator is also available to compare a specific color with the current drawing color to find out if they differ due to characteristics of the output device (the normal POSTSCRIPT comparison operators can be used to compare color objects). Instead of extending the **image** operator to handle color, a new **buildimage** operator was added that creates a canvas object from an image-like description. This canvas can then be imaged on a display device with an **imagecanvas** or **imagemaskcanvas** operator.

DISPLAY POSTSCRIPT's extensions, in contrast, are more directed at handling color printers. Direct support for the CMYK color model was added and the halftoning and transfer components of the device state extended to fully handle the RGB and CMYK models. A new **deviceinfo** operator returns a dictionary containing a specification of: the number of color components, the number of bits per sample for red, green, blue, or gray, and the number of bits per pixel. Finally, a new **colorimage** operator was added that supports the functionality of the **image** operator and also handles multi-channel color images. The **colorimage** operator handles 1, 3 (RGB), or 4 (CMYK) channel images, and the data may be organized either as a merged data stream (e.g. RGBRGB...), or as separate streams of data (e.g. RRR...GGG...BBB..). Data may have no more than 8 bits per sample (as in NeWS).

The definition of a color type in NeWS is a worthwhile addition that permits the support of many different color models. To fully carry this out, however, it must be possible to define colors that are not just three components, but have multichannel spectral values[9].

---

[8] Gosling, private communication; 1988.

[9] See Version 3.0 of the RenderMan Interface Specification for an example of a system that supports

The addition of support for the CMYK color model in DISPLAY POSTSCRIPT, for example, could have been done more cleanly if such a facility existed.

While NeWS acknowledges the possibility that colors displayed on an output device may not be exactly those requested by an application, it provides no facility by which an application can *control* the mapping between a desired color and the actual color displayed. The DISPLAY POSTSCRIPT transfer and halftoning facilities are only partially successful in this respect. The support for *colormap* objects in the X11/NeWS merged server does not fully address these problems.

The **colorimage** operator in DISPLAY POSTSCRIPT is a reasonably clean extension to handle color images. It is not completely backwards compatible with the **image** operator. The NeWS facilities for image manipulation are less flexible, but nicely integrated with drawing surfaces. Importantly, they permit an application to redisplay an image without having to recalculate its contents (something not possible with DISPLAY POSTSCRIPT, although one would expect the window system that DISPLAY POSTSCRIPT is embedded in to support such a facility.)

The use of color causes the amount of data needed to specify an image to increase significantly. Unfortunately, neither DISPLAY POSTSCRIPT or NeWS provide direct support for decoding compressed image data. The procedural hooks provided in the **image** operator should be sufficient to implement decoding algorithms in POSTSCRIPT code, but it is unlikely that this is efficient enough for an interactive environment. The NeWS **readimage** operator that creates a canvas object by reading an image from a file would be more useful if the file format were something other than the crufty Sun *rasterfile* format[10].

2.12. Miscellaneous Additions

There are numerous other additions in DISPLAY POSTSCRIPT. Several are earmarked as specific to window system. These are listed in Table 3.

Operator	Description
currentdevice	return information about current display device
infill	test whether point would be painted by fill
ineofill	test whether point would be painted by eofill
inufill	test whether point would be painted by ufill
inueofill	test whether point would be painted by ueofill
instroke	test whether point would be painted by instrok
inustroke	test whether point would be painted by inustrok
wtranslation	return translation from window origin to device space origin

Table 3. DISPLAY POSTSCRIPT window system support operators.

NeWS provides a **pointinpath** operator that is equivalent to **infill**, but nothing similar to the other DISPLAY POSTSCRIPT operators.

---

many different color models.

[10] For example, Aldus/Microsoft's Tag Image File Format (TIFF).

2.13. What's Missing?

There are several areas that DISPLAY POSTSCRIPT does not address in the support of interactive systems. Some of these are required for virtually any window system that is built on top it. Other areas seem as if they should be addressed to ensure that applications that are built on top of DISPLAY POSTSCRIPT are portable. Portability, however, is not possible with any DISPLAY POSTSCRIPT-based application because there are too many facilities that are left unspecified (e.g. connection management, drawing surfaces, and input handling).

The first, most obvious facility, that is missing is a mechanism for copying an area of an image from one place on the display to another. This facility is necessary for the efficient implementation of scrolling. NeWS defines a **copyarea** operator that cleanly integrates this function into the standard POSTSCRIPT imaging model. The NeXT extensions to DISPLAY POSTSCRIPT include a **composite** operator that is based on the compositing formalisms presented by Porter and Duff[11].

A second facility that is missing is a way to draw images that are only present for a short period of time; for example, a cursor image or some form of highlighting. Such a mechanism is usually needed even if the system supports multiple overlapping drawing surfaces because these objects tend to be too expensive to use for short bursts of activity. NeWS has the notion of an *overlay canvas*; a canvas that is temporarily present on the display. The overlay canvas facility provides limited drawing functionality that can be implemented, for example, with overlay bitplanes. Overlay canvases are used for highlighting selections, for feedback during rubberbanding operations, etc. Cursors are supported directly as an additional object that is associated with a canvas. The NeXT window system defines a facility that is equivalent to NeWS's overlay canvas, an *instance drawing mode* wherein the contents of a window are saved while instance drawing takes place. Cursors are also directly supported as an attribute of a window.

DISPLAY POSTSCRIPT would benefit greatly by the addition of a drawing surface object. While this might make it more difficult to integrate DISPLAY POSTSCRIPT into an existing window system, it would provide an important intermediate layer between the graphics state and an underlying display device. In particular, a drawing surface object would permit a concrete specification for the handling of: damage repair, input multiplexing, and retained drawing images. Additionally, in the current architecture, a window must be modeled as a device. This can cause confusion when information exists both for a window and the *underlying* hardware device that the window resides on (for example information about color).

POSTSCRIPT is defined with a write-only model. That is, there is no way to retrieve an image's representation once it has been rendered. NeWS provides a limited facility to retrieve data in the **writecanvas** and **writescreen** operators, but these are blemishes on the general POSTSCRIPT imaging model. A better mechanism that provides more device independence is needed for both systems; possible the NeXT **sizeimage** and **readimage** operators.

---

[11] "Compositing Digital Images." Thomas Porter and Tom Duff. Computer Graphics (SIGGRAPH '84 Conference Proceedings), Vol. 18, No. 3, July 1984, pp. 253-259.

The way in which the DISPLAY POSTSCRIPT POSTSCRIPT interpreter "fits" into its operating environment is poorly defined. The POSTSCRIPT interpreter may be accessed through a communication channel or as a library-like facility—this is undefined. The client library does not even include an interface routine to establish contact with the interpreter. This makes the DISPLAY POSTSCRIPT system decidely nonportable. I believe that it is possible to define a communication-based interface to DISPLAY POSTSCRIPT that is portable except for the communication protocol used between the POSTSCRIPT interpreter and client applications.

3. DISPLAY POSTSCRIPT Programming

An application that uses DISPLAY POSTSCRIPT is typically a combination of C code and POSTSCRIPT code. The DISPLAY POSTSCRIPT client library provides a means of transmitting POSTSCRIPT code to the interpreter, and interprets information returned by the interpreter to the application. An application may draw on a display by making calls to client library procedures, or by defining stylized interfaces that are created with the **pswrap** utility. For example, to execute the POSTSCRIPT operator **translate**, the following call might be made:

```
PStranslate(1.1, 0.5);
```

this is equivalent to the POSTSCRIPT code:

```
1.1 0.5 translate
```

The client library includes similar procedures for each of the POSTSCRIPT operators.

Typically an application uses a limited set of drawing requests that may be easily encapsulated in procedure-like bodies of POSTSCRIPT code. In this case, rather than use the single operator routines defined in the library, a special purpose interface can be defined with the **pswrap** utility. This program takes POSTSCRIPT code bodies and C-like interface definitions, and creates C callable procedures that cause the associated POSTSCRIPT code to be transmitted to the interpreter where it is executed. This is the equivalent of combining one or more of the single operator procedures into a single procedure (with the expected performance gain). For example, the following C code draws a gray filled rectangle:

```
PSgsave();
PSsetgray(gray);
PSfillrect(rectNums);
PSgrestore();
```

while an equivalent, single procedure, **pswrap** definition is:

```
defineps grayRect(float gray, rectNums[4])
        gsave
        gray setgray
        rectNums fillrect
        grestore
endps
```

The resultant procedure, `grayRect()`, could then be called as in

```
grayRect(gray, rectNums);
```

The single operator procedures and the **pswrap** utility are the entire set of drawing tools provided to a programmer that wants to create a DISPLAY POSTSCRIPT-based application. In this sense, users of DISPLAY POSTSCRIPT are no better off than users of NeWS (where the only facilities provided are the **cps** program and the supporting client library). The remainder of this section provides a more detailed comparison of the programming tools: **pswrap** with **cps**, and the DISPLAY POSTSCRIPT and NeWS client libraries.

3.1. pswrap and cps

Pswrap is a more refined version of **cps**. In particular, **pswrap** provides certain facilities that are not present in **cps**, or which are provided in only a limited manner:

- **pswrap** can be used as a preprocessor, a la the UNIX† utilities **lex** and **yacc** (**cps** has an escape mechanism for passing C code through, but it is clumsy and is only suitable for simple declarations),
- **pswrap** can generate external references for ANSI C compilers, and
- **pswrap** can generate reentrant code for shared libraries and multi-threaded applications.

Most importantly, **pswrap** provides better support than **cps** for passing and returning aggregate data structures. Specifically, homogeneous arrays of numbers can be passed and returned (as shown above). The syntax for specifying parameters is also closer to that used to define C variables, making it more intuitive to the C programmer. Having return parameters defined in the function prologue is an improvement over the definition style used by **cps** (it makes it simpler to locate return results). Finally, having **pswrap** generate POSTSCRIPT code to return parameter values is an improvement over **cps**.

On the down side, **pswrap** has a confusing scheme for handling POSTSCRIPT literals, names and strings. With this scheme, parameters names that are declared as "`char *`" are interpreted in a context sensitive manner. If the parameter is preceded by a "/", the parameter is treated as a literal. If the parameter is enclosed in parenthesis, it is treated as a string. Otherwise, the parameter is treated as an executable name. This can easily lead to confusion. For example, in the following

```
defineps threeStrings(char *str)
        (str) ( str ) (a str)
endps
```

only the first instance of `str` is substituted for. Furthermore, because names and literals are distinguished at the time **pswrap** is run, instead of at runtime, the only way to define a parameter that is either a literal or a name is to bypass the facilities provided by **pswrap**[12]. While less C oriented, the `string` and `postscript` type definitions provided by **cps** appear to be a better way to define such parameters.

---

[11] † UNIX is a registered trademark of AT&T Bell Laboratories in the USA and other countries.

[12] I found several instances where I wanted to do this in my NeWS client library.

3.2. Client Libraries

The NeWS client library is very simplistic and designed solely to support code generated by the **cps** utility. The DISPLAY POSTSCRIPT client library is slightly more elaborate. Aside from a set of single operator POSTSCRIPT procedures, DISPLAY POSTSCRIPT supports four notable facilities not found in its NeWS counterpart:

- the ability to manage multiple parallel threads of execution in the interpreter through the notion of a *context*,
- an exception handling facility that is integrated with the context abstraction,
- support for user defined POSTSCRIPT objects and names, and
- the ability to dynamically change the format of the POSTSCRIPT transmitted to a context.

A context is a logically independent POSTSCRIPT interpreter session. Each client-based context is normally associated with a POSTSCRIPT *context* in the interpreter, but they may also, for example, be associated with a text file for the purpose of debugging or printing. An application can have multiple contexts active at a time and contexts can be linked so that output directed to one context is automatically directed to all linked contexts as well. (This seems to have application only for the purpose of debugging.) A *current context* is identified as the default destination for POSTSCRIPT requests that do not take an explicit context parameter.

The exception handling facility permits an application to define handler procedures for exceptional conditions in a nested fashion. The only exceptional conditions defined are for errors detected in the POSTSCRIPT interpreter or in the client library. It is unlikely that large applications will use these facilities because they are not well integrated with the UNIX runtime environment; i.e. signals.

The support for user defined POSTSCRIPT objects and names amounts to a set of tables that map strings into binary encoded tokens that are transmitted to the POSTSCRIPT interpreter. This a useful facility, particularly when used in conjunction with the last facility, the ability to dynamically change the format of the data transmitted to a context. That is, by maintaining a name table it is possible to record a a stream of POSTSCRIPT in a file simply by switching to an ASCII encoding and setting the current context to one that is associated with a file.

Noticeably missing from the DISPLAY POSTSCRIPT client library is any standard mechanism for establishing contact with the POSTSCRIPT interpreter. This facility is not part of DISPLAY POSTSCRIPT. Instead applications are expected to first gain access through the appropriate window system mechanism(s), with the resultant window then used by DISPLAY POSTSCRIPT. In a similar vein, there is no mention of how the client library multiplexes client-server communication when multiple contexts are used by an application. Adobe believes this is a property of the window system and operating system and should be left unspecified[13]. Whatever scheme is used to handle multiplexing and communication issues must also deal with input events.

[13] On the NeXT machine, under Mach, each context has an independent communication stream associated with a Mach IPC port. For other machines, Adobe uses some (undefined) protocol to multiplex a single TCP connection to a server process.

4. Summary and Conclusions

DISPLAY POSTSCRIPT provides a number of extensions to the basic POSTSCRIPT language explicitly for use in an interactive setting. These facilities have been added in a fashion that is consistent with the underlying concepts of the POSTSCRIPT language. Execution contexts address the requirement of multiple, simultaneous users of a single display. The binary stream encoding provides a clean mechanism for the efficient exchange of information in a client-server environment. View clipping, the new text operators, and graphics states satisfy various needs of non-printer-oriented applications. User paths appear to be a useful mechanism for improving the drawing performance of interactive applications.

The designers of NeWS have addressed the same problems as the designers of DISPLAY POSTSCRIPT; providing similar additions to the POSTSCRIPT language. In some instances, *e.g.* the **save** and **restore** operators, their efforts have not been completely successful. However, in comparison to DISPLAY POSTSCRIPT they have made fewer visible changes while still achieving good interactive performance. The addition to NeWS of the new operators defined in DISPLAY POSTSCRIPT should be a significant boon to interactive performance.

In general, I believe that DISPLAY POSTSCRIPT does not go far enough in addressing many of the requirements of interactive applications. In particular, DISPLAY POSTSCRIPT does not:

- provide integral support for the notion of a drawing surface,
- address input handling,
- define the interface between the operating environment and DISPLAY POSTSCRIPT contexts at a high enough level or in a complete enough manner to fully define their semantics,
- provide support for copying portions of a drawing surface,
- support cursors, and
- provide highlighting mechanisms.

In addition, I prefer the *object-oriented style* with which additions were made in NeWS to the *operator-oriented style* employed in DISPLAY POSTSCRIPT. The object-oriented style allows NeWS, for example, to provide a cleaner interface to color.

In a sense it is not clear how useful it is to compare DISPLAY POSTSCRIPT and NeWS because NeWS is a self-contained system, while DISPLAY POSTSCRIPT is a foundational layer on which a system such as NeWS might be constructed. A better comparison would be between NeWS and a system such as NeXTStep or “X with DISPLAY POSTSCRIPT.” If this comparison were to be done, however, I believe that the lack of integration of facilities such as drawing surfaces and input handling would be apparent. NeWS does a very nice job of melding these facilities into the POSTSCRIPT language, and DISPLAY POSTSCRIPT would be much better if it had either adopted the interfaces defined in NeWS, or defined equivalent notions. For example, there are difficult synchronization problems that must be addressed in integrating input handling. Having the facilities defined by the window system in which DISPLAY POSTSCRIPT is embedded may make the handling of such problems harder. At the very least, by not specifying these facilities in DISPLAY POSTSCRIPT Adobe has made it difficult to write portable DISPLAY POSTSCRIPT-based applications.