The fundamentals of NeWS

Tim Long

NeWS is a system developed by SUN Microsystems for driving bitmapped graphic displays. It uses an extended PostScript to implement its drawing primitives and is designed to work in a multi-process and multi-machine environment.

NeWS attempts to combine the machine and operating system independence of the canonical VDU with the interactive capabilities of the bitmapped graphics workstation.

In short, NeWS is a system a programmer uses to implement an interactive graphic interface, in the same sense that UNIX is a system a programmer uses to implement an application.

There are two other well known systems which attempt to address some of the same issues as NeWS. One is X-Windows, the other the Blit.

X-Windows attempts to achieve device independence by defining a canonical form for classic bitmapped display operations. It also defines an encapsulation and network transfer mechanism for these operations. By this means, X-Windows attempts to drive remote and varied displays with the same sort of library routines that have been used in systems where the display is local and tightly coupled to the driving application.

Another system which addresses the 'bitmapped graphics terminal' issue is the Blit.

In addition to other innovations the Blit takes a different approach to the application-to-graphics interface by allowing an application to program the terminal. This approach greatly relieves network bandwidth and allows a well written application to give a high fidelity of interactive response. But both the Blit and X-Windows fail in important respects.

The most important is device independence.

X-Windows attempts to achieve device independence through the adoption of a canonical bitmapped display to which all operations are mapped. In reality this does not allow for much variation. If the pixels are not just about 80 to the inch things begin to look very strange. If there is any unusual special purpose hardware in the display it often has to be ignored. And the handling of colour versus monochrome is awkward. X-Windows also suffers from requiring a substantial library on the application programming machine, and has major problems with network bandwidth.

The Blit takes a more extreme attitude. A Blit is a Blit is a Blit. And if you don't have a cross-compiler and the appropriate libraries, forget it.

The Blit, X-Windows and almost all other bitmapped graphics system suffer another major problem. They are difficult to program.

The primitives provided normally start at the bit manipulation level and, despite layers of protective libraries, still require a large investment in reading, programming and debugging to make an application operate in a basic fashion. When it comes to making it slick and attractive even more work is involved. As if this wasn't enough, all systems, except the Blit, impose a structure on the application program based around a main loop with some form of 'get next event' call and an enormous switch statement to decide what to do about it. This structure is often totally inapplicable to the task at hand.

NeWS, despite years of effort by SUN, and a somewhat over-engineered implementation, solves all of these problems and more. It has the programmability of the Blit, the network transparency of X-Windows, and a device independence and ease of programming all its own.

NeWS is based on PostScript. PostScript is a stack based interpretive programming language rich in primitives for rendering static two dimensional pictures.* It is device independent and well designed for the description of printed images. The extensions which turn PostScript into NeWS revolve around:

1. The introduction of multiple processes with private stacks but otherwise shared resources.
2. The introduction of an asynchronous message passing scheme for event handling and IPC.
3. The introduction of multiple drawing surfaces and the definition of a mechanism to keep the pictures drawn on them up to date.

In practice, a NeWS terminal (or server) implements an extended PostScript (which I will hereafter call NeWS-PostScript) and operates along the following lines:

The server initially starts with one process running. This process waits for a foreign connection request (via TCP/IP say). For each of these that it receives, it forks a new NeWS-PostScript process which establishes an environment similar to the standard PostScript starting environment and begins to execute operators from the connection. Meanwhile the main process is back waiting for new connections.

The application which requested the connection is now free to send NeWS-PostScript operations down the connection. Almost all applications would first send some bunch of function definitions and initialisation code. Once in operation, the terminal (server) side of the application will have several NeWS-PostScript processes running to implement various aspects of the user-interface. Any of these may also be involved in sending data back to the main application, while one process will still be executing operators from the connection to allow the main application to control the terminal side.

---

* For a short description of PostScript see the introduction to *The PostScript Language Reference Manual* by Adobe Systems.

A slightly more detailed examination of the extensions to PostScript which turn it into NeWS will make this mode of operation a little clearer.

Memory and processes

The memory model of pure PostScript divides all objects into two classes, simple and compound objects. Simple objects are pushed directly onto stacks by value and, naturally, the memory they occupy is relased when they are popped off. But compound objects (such as arrays) are pushed onto stacks by reference and have their value in 'virtual memory'. The memory used by these objects is only reclaimed by restoring the state of the interpreter back to some previously saved state.

While in most respects NeWS is identical to PostScript, the NeWS memory model is different. Compound objects in NeWS are retained while they are referenced and freed when they are no longer referenced. But the distinction between simple and compound objects is still important when considering multiple processes.

A new NeWS process is started by a *fork* primitive. The new process gets a copy of its parent's stacks, therefore simple objects are copied, but compound objects have a value shared between the two processes because they reside in the common 'virtual memory'. Likewise drawing surfaces (the screen included) and other terminal wide resources are shared.

NeWS processes are intended to be light-weight primitives. They are used for seemingly trivial tasks. Poping up a menu or just dragging a box around the screen would typically be done by forking a process to do it. This is one of the key elements which can make user interface programming in NeWS particularly easy.

Context switching is not pre-emptive. It happens only when a process performs blocking operations or explicitly allows context switching.

Events

Events are bundles of information about something which has happened. Some are generated by the NeWS system automatically to describe real world events (such as key transitions and mouse movements), and they are also used for inter-process communication.

An event, once generated, is 'distributed' to applicable processes. Which events are distributed to which processes depends on 'expressions of interest' processes have made for events matching some pattern. For example, suppose a process wished to act on left mouse button events over a particular drawing surface. It would construct an event which looked like such an event, setting to *null* those

fields which it did not care about and specifying those fields it did (such as the drawing surface and type of event). This would then be used as an argument to the *expressinterest* operator.

A process will typically receive events by executing the *awaitevent* operator which blocks until an event of interest is available, but once one is available, it returns it.

Canvases

Canvases are things you draw on. Canvases have been introduced to allow multiple drawing surfaces as opposed to the single page of pure PostScript.

Canvases are created in a hierarchy, that is, each canvas has a parent canvas on which it was made. A canvas also has a position relative to its parent and a clipping-path (further restricted by its parent's). Unlike most other windowing systems, NeWS's canvases are bounded by arbitrary paths.

Most other properties of canvases are variable. For instance a canvas may be transparent in which case all drawing primitives are subject to its position and boundary but then just flow through to the parent. Of more utility is the opaque canvas. Painting primitives on opaque canvases directly effect the bitmap that corresponds to them (subject to obscurement by other opaque canvases).

Canvases are intended to be 'cheap' graphics primitives and should not be confused with a window.

Picture maintenance

At the heart of any windowing system is some technique for picture maintenance in the presence of opaque drawing surfaces appearing, disappearing and moving. This normally, when all else fails, involves the application being asked by the system to regenerate some image which has been lost. NeWS is no exception to this.

When some previously hidden part of an opaque canvas becomes visible (and the NeWS server does not have the bitmap hidden away somewhere) a 'damaged' event is generated for that canvas. In a correctly constructed NeWS program every on-screen opaque canvas will have some process that is interested in damaged events on it and be prepared to repaint the image. There are a number of ways of optimising this but the principle remains.

Writing a NeWS program

Writing and running programs to use NeWS requires very little support on the client machine beyond the connection mechanism to the NeWS terminal. There is a small library, the source for which is available form SUN, which implements the few routines needed to establish a connection to a NeWS server (in terms of 4.2BSD networking primitives of course). The only important function is *ps_open_PostScript()* which sets two global variables to be standard I/O streams, one to, and one from, the NeWS server.

Once a connection is established almost all NeWS client programs will wish to transmit a bunch of PostScript functions to the terminal. The obvious way of doing this is to copy the source of your PostScript functions from some data file to the connection. This is very easy to do. But for some reason which remains a mystery to me, SUN have provided a much more difficult method. SUN's documentation uses this method exclusively so you could be forgiven for thinking this is the best approach.

In brief, SUN's method involves embedding fragments of your PostScript within pseudo-C function declarations. The syntax for this is poorly designed. The resulting file is parsed by a programs called *cps* which produces an include file with declarations of arrays which contain your PostScript and macros which transmit the various fragments.

Typically one of the fragments will be the bulk of your PostScript functions and initialisations. This will normally be invoked as soon as your connection is established.

After initialisation the difference in the two methods is small. For instance, using cps to draw a line would involve defining a function in the cps file which associates a PostScript fragment with a C function, called say ps_line(), and then invoking the function. So in the cps file we would have:

```
cdef ps_line(int tox, int toy)
    tox toy lineto
```

Then in the C program:

```
ps_line(tox, toy);
```

I think it is clearer to say:

```
fprintf(PostScript, "%d %d lineto\n", tox, toy);
```

because this is all the ps_line macro does anyway.

An example

The best way to get a feel for NeWS is to try and program it, but in the absence of that, looking at an example will do.

This example is somewhat artificial because I have fully expanded (or dodged altogether) a lot of code which would normally be provided by SUN libraries. I have also not used the SUN window system. This is deliberate.

SUN's libraries seem to be designed to make programmers used to the old way of doing things feel comfortable. But in doing this they largely defeat the benefits NeWS. It may be that SUN felt it was essential to do things 'the old way' in order to gain some form of market acceptance. Their plans for the next release of NeWS which contains an X-Windows immplementation certainly suggests they feel this is important. But in my opinion using SUN's windowing system would only obscure the purpose of this example, which is to demonstrate the fundamentals of NeWS.

This example maintains the time in ctime(3) format in a small white rectangle in the lower left of the screen. The point of this simple example is that the C program (the client) knowns how to find out the date, and the PostScript side (the terminal or server) known how it wants it displayed.

The client side of the example:

1. establishes a connection to the NeWS terminal;
2. copies its NeWS-PostScript code from a text file to the terminal;
3. responds to requests for the time from the terminal.

The terminal code:

1. makes the canvas on which it will print the time;
2. forks a process which maintains the time canvas and makes periodic time requests to the client;
3. does what the client tells it to do.

For more details see the programs.

The separation of the semantics of the application from the graphics is an essential part of good NeWS programming. This simple example is based on the hypothesis that the full breakdown of the current time is something only the client application is capable of calculating, so it has the job of supplying this. But how this is displayed is entirely up to the server code.

Given the well defined format of a ctime string the PostScript side could be pulling out the numbers and using them to draw hands on a picture of Big-Ben. Whatever is was doing would make no difference to the client code.

```
#include <stdio.h>

extern FILE *PostScript;
extern FILE *PostScriptInput;
extern char *ctime();

main()
{
    FILE    *stream;
    long    t;
    char    *s;
    int             c;

    /*
     * Establish a connection to the NeWS terminal.  (Sans error
     * messages for brevity.)  This sets PostScript and PostScriptInput
     * as a side effect.
     */
    if (ps_open_PostScript() == 0)
      return 1;

    /*
     * Send the initialisation code to the NeWS terminal.
     */
    if ((stream = fopen("clock.ps", "r")) == NULL)
      return 1;
    while ((c = getc(stream)) != EOF)
      putc(c, PostScript);
    fclose(stream);

    /*
     * Do what the server side asks of us.  This is a particularly simple
     * communications protocol from the server side, but in reallity
     * you hardly ever need anything more than a scanf.  It's not as if
     * there is a user out there sending this stuff.  You contol both
     * ends so make it easy on yourself.
     */
    while ((c = getc(PostScriptInput)) != EOF)
    {
      switch (c)
      {
      case 't':
          time(&t);
          s = ctime(&t);
          s[24] = '\0';
          fprintf(PostScript, "(%s) Update\n", s);
          fflush(PostScript);
          break;

      case 'q':
          flcose(PostScript);
          return 0;
```

```
%
% I have left it to here to mention a couple of points which will
% presumable only be of interest to people who wish to read the PostScript.
%
% Process IDs, canvases, and events are all immplemented in NeWS as
% dictionaries in the same way that fonts are in pure PostScript.
% That is: a canvas is manipulated as a dictionary with well known
% element names.  Likewise for events and processes.
%
% There is an example of this immediately below where a newly
% created canvas is 'begun' in order to set some of its attributes.
%

%
% Create a new canvas (call it 'ClockCanvas') whoes parent is the framebuffer.
% Reshape it to be at 0,0 and 145 long by 16 high (in points as usual).
% Make it opaque and mapped onto the screen, then finally make it the
% 'current canvas' (and therefore the implicit target of drawing operations).
%
/ClockCanvas framebuffer newcanvas def
0 0 145 16 rectpath ClockCanvas reshapecanvas
ClockCanvas begin
    /Transparent false def
    /Retained false def
    /Mapped true def
end
ClockCanvas setcanvas

%
% This function is run as a seperate process.  It handles a few things:
% it keeps the canvas up to date, it catches clock ticks, forwards
% them to the client side and re-primes the clock, and finally it
% forwards a left mouse button press on the canvas as a finish message
% to the client.
%
/ClockProc
{
    %
    % Express intrest in damage event on the clock canvas.
    %
    createevent begin
      /Name /Damage def
      /Canvas ClockCanvas def
      currentdict
    end
    expressinterest

    %
    % Express intrest in left mouse button down transitions over the canvas.
    %
    createevent begin
      /Name /LeftMouseButton def
      /Action /DownTransition def
      /Canvas ClockCanvas def
      currentdict
    end
    expressinterest
```

```
%
% Express intrest in 'Tick' events.  These are not NeWS systems events
% but something of out own fabrication.
%
createevent begin
  /Name /Tick def
  /Process currentprocess def
  currentdict
end
expressinterest

PrimeTick                       % Start the clock.
{
  awaitevent              % Wait for anything of interest.
  %
  % The event (which we just got or we wouldn't be here) is
  % a dictionary (as usual) so 'begin' it to get easy access
  % to the contents...
  %
  begin
      /Name /Damaged eq
      {
        %
        % A damaged event, must redraw canvas.  A small amount of
        % effort (and a reasonable sense of ascetics) can make a
        % big difference here.  By changing the shape of the
        % canvas, pulling out a few numbers from the ctime string
        % and drawing hands and things you get an analog clock.
        %
        damagepath clipcanvas   % Restrict drawing to damaged areas.
        1 fillcanvas            % Erase to white.
        0 setgray
        1 4 moveto Ctime show   % Redraw the current Ctime.
                          % (The default font is 12 point Times.)
      }
      if
      %
      % For 'Tick' events we just print a 't' to the client
      % and reprime the clock.  The client will give us a
      % new Ctime value and cause us to redraw in due course.
      %
      /Name /Tick eq {(t) print PrimeTick} if
      %
      % For left mouse down event we send a 'q' to the
      % client and break from the loop (which then falls off
      % the end of this processes code causeing it to exit).
      % The other process (the main one the client is
      % talking to) will exit when the client closes the
      % connection.  This will cause our userdict to be
      % discarded which will cause the last reference to the
      % ClockCanvas to disappear which will cause it to
      % vanish from the screen and any canvases it was
      % obscuring to get damaged events.  Simple really.
      %
      /Name /LeftMouseButton eq {(q) print end exit} if
  end
}
loop
}
def

%
% Generate a 'Tick' event to arrive at ClockProc in one second.
%
/PrimeTick
{
    createevent begin
      /Name /Tick def               % "Tick" is our invention.
      /Process ClockPID def         % Send straight to the given process.
      /TimeStamp currenttime 1.166 def % TimeStamp in future, so deliver then.
      currentdict             % Leave on stack for sendevent below.
    end
    sendevent
}
def

%
% (ctime-string) Update -
%
% Update Ctime and damage the ClockCanvas (which is assumed to be
% the current canvas) to cause it to be repainted.
%
/Update
{
    /Ctime exch def
    clipcanvaspath extenddamage
}
def

/Ctime () def

%
% Fork of the ClockProc function (above) and store the pid in ClockPID.
%
/ClockPID (ClockProc) fork def

%
% This is the end of the initialisation code sent down the connection.
% But this process will continue to read anything the client program
% sends down the connection until it gets closed.  In this case the client
% will just send things like:
%
%     (Tue Aug 23 01:33:08 EST 1988) Update
%
% See the Update function above and clock.c.
```

Conclusion

This description of NeWS is incomplete. There are some aspects of NeWS which I have failed to explain. Some because I hope they are replaced (such as SUN's windowing system). Others because they are simple and work (such as colour support). But mostly because I have tried to stick to the fundamentals of NeWS, for this is where its strength lies.

The programmability of the graphics terminal with a language appropriate to the task allows for great reductions in the bandwidth required between the display and application. At the same time it gives the NeWS programmer the opportunity to improve the responsiveness or sophistication of the interface using local CPU power.

The use of PostScript as a drawing primitive is a great advantages of NeWS. Although it is not often recognised, the malleability and availability of artwork is essential to the production of good graphic interfaces. PostScript is a superbly portable picture description language. Bitmapped based systems can never achieve the same degree of flexibility.

The use of light-weight processes is also an enormous advantage in interactive graphics. In many extant graphics programs the need for the application to be on-tap to perform display housekeeping at a moments notice totally perverts their structure. Being able to farm out conceptually different parts of picture maintenance to appropriately constructed processes greatly simplifies the programming task and improve the program's behaviour. But there is one point which overshadows all of the above.

The importance of device independence can not be over emphasised. Good software has a long natural lifetime, whereas hardware tends to come and go on an annual basis. Software tied to hardware will die. It may struggle for a while, supporting the hardware that dooms it, but die it will. It is in this matter that NeWS has it all over its peers. NeWS does not operate at the bitmap level, it is far more independent of changes in display technology. It is also far more capable of taking advantage of new features. It will take some radical redevelopment of X-Windows and all X-Windows applications to handle 300dpi screens. But 300dpi screens and the CPU power to drive them are almost here. Given the necessary CPU power NeWS will handle these without difficulty. Application programs will run unmodified, and take full advantage of the resolution.