A Screen Previewer for VM/CMS

Don Hosek

A good previewer is a useful tool for working with TEX, but unfortunately, there are very few available. For users of TEX under the IBM VM/CMS system, the only choice available used to be DVI82, a Versatec driver that, as an added feature, allowed previewing on IBM 3279 and 3179-G terminals.

To deal with this situation, I wrote DVIview, a TEX previewer that displays its output on VT640-compatible displays connected to an IBM mainframe via either a 3705 controller or a Series-1/7171 protocol converter. In addition, the output routines are modularized enough that it should be a fairly simple task to modify the program to drive any graphics terminal connected to the mainframe. (I have plans to include support for GDDM-driven displays in the near future.)

DVIview is a lengthy `WEB` program that interprets the instructions in a DVI file and displays them on the user's screen as determined by commands typed at the keyboard. The entire page may be viewed with block outlines of the characters, or smaller portions of the page may be selected and viewed using the actual shapes of the TEX fonts. Font information is read from PK files. (I cannot recommend the PK format enough to people writing new device drivers; the fonts take roughly half the space of GF files and about a third the space of PXL files. And PK readers are easier to write!)

The DVIview distribution includes two manuals: "Previewing TEX Output With DVIview" is the users' guide and explains how to use the program from a user's standpoint. Also included is "Installing and Customizing DVIview", intended for the systems person who installs DVIview. Instructions are given for installing DVIview as is, as well as instructions on adding changes to the file and a "Hitchhikers' Guide to `WEB`" (for those who don't care how they get where they're going as long as they don't have to ride the bus).

Due to the size of the program, it cannot be distributed over the networks. To obtain a copy of DVIview and its documentation, send $30 (to defray duplication costs), a blank tape, and a return mailer to:

Don Hosek
Platt Campus Center
Harvey Mudd College
Claremont, CA 91711

This article first appeared in TEXmag, 1987#7.

The program is public domain, so feel free to give it away. However, since it is still a young program, I'd like to keep track of who has copies for purposes of distributing updates.

Why TEX Should NOT Output PostScript — Yet

Shane Dunne
University of Western Ontario

In a recent TUGboat issue [1], Leslie Lamport suggested that since PostScript is becoming accepted as a standard page description language, perhaps TEX could be modified to output PostScript instead of DVI code. This is a good idea, but it should *not* be done yet for the following reason: At the moment, the available PostScript literature does not state precisely how drawn objects are to be rendered on the output raster. As I will show in this article, such a specification of PostScript's semantics is urgently needed to allow precision application programs such as TEX to properly use the language. I have written to PostScript's developers, Adobe Systems Inc. of Palo Alto, California, to draw their attention to this problem, and suggested that it be resolved publicly using TUGboat as a forum for discussion.

For readers unfamiliar with the PostScript language, a few words of explanation are in order. PostScript is a language designed specifically for specifying the output of raster printing devices. The language is interpreted, with the interpreter usually resident in the printer itself. It was intended to be human-readable, and hence uses only printable ASCII characters, but to simplify parsing it uses a rather cryptic postfix syntax. This is justified on the grounds that most PostScript programs will be written automatically as the output of other applications. PostScript incorporates a sophisticated device-independent drawing model in which a single transformation matrix (called the *current transformation matrix* or CTM) specifies the correspondence between the user and device coordinate systems. User coordinates are floating-point numbers with essentially infinite resolution; device coordinates are normally integers.

The incompleteness of the current PostScript semantic definition is apparent from the following example. Assume that the CTM of a PostScript device is set so that one unit in user space corresponds

to the distance between adjacent device pixels, and the point with coordinates (100, 100) is well within the visible part of the output page. (This is what our TEX driver does.) Now suppose the following code fragment is executed.

```
newpath
100 100 moveto
1 0 rlineto
0 1 rlineto
-1 0 rlineto
closepath fill
```

This draws an outline "path" which is a unit square with lower left-hand corner at (100, 100), and then fills it with black. It is reasonable to expect that a single device pixel will be blackened — after all, that *is* a black box one unit high by one wide, with the units we have chosen. However on our QMS PS800 laser printer, the result is a two-pixel by two-pixel box — four pixels are blackened. It turns out that whenever you ask for a box which is x units wide and y units high, you get one which is $x + 1$ pixels by $y + 1$ pixels. Similar things occur with the *stroke* command which draws lines — if you ask for a line width of one unit you get lines two pixels wide, two units becomes three pixels, and so on.

The practical upshot of this is that our DVI-to-PostScript driver, which outputs code according to what the PostScript reference manual says, always yields TEX rules which are one pixel too long and one pixel too high.

Attempting to second-guess the programmers of the PS800 PostScript implementation, I came up with the following scenario. We begin with the outline path with four vertices (100, 100), (101, 100), (101, 101), and (100, 101). Since we are working in one-to-one scale, multiplying these coordinate pairs by the CTM may add some translation factors, but should not make any multiplicative change to the values. The coordinates, which are real numbers, must next be converted to integers for the hardware, but they are already integers, and I have verified that the translation factors are also. Thus we can suppose without loss of generality that the coordinates are unchanged by the CTM. Now comes the strange part. The implementation seems to interpret integer-valued coordinate positions as *pixels*, and thus says that it must blacken all four different pixels identified by the four vertices (and, in this case, nothing else).

So apparently, each distinct coordinate *position* (a mathematical point in the plane with zero height and width) has been identified with a device *pixel* (something with very real height and width). Of course, if I am drawing a box 1 inch by 1 inch at 300 dots per inch, the error is only 1 part in 300, but if I am drawing *small* things (small with respect to the device resolution), the error can be quite serious, as shown by the above example. Unfortunately, typesetting and related applications involve *small* objects almost exclusively.

It would be more consistent with the PostScript philosophy to identify integer-valued coordinates (at 1 : 1 scale) with the lower left-hand corner of a pixel. This would require a small refinement to PostScript's fill algorithms.

The PostScript Language Reference Manual [2] says nothing definitive about the correspondence between coordinate positions and device pixels. It defines a virtual graphics machine separated from the real device by various mechanisms (such as the CTM) whose exact operation it does not define. Now in my experience, anything not defined in a software specification is usually defined by its implementation, which in turn means that I can expect different results from different printers, even at the same dot resolution.

It is of course tempting to say "Why worry about such details? If you want higher precision just go to a device with more dots per inch." There are two answers to this. The first is that 300 dpi laser printers, and lower-resolution mechanical dot-matrix printers, are probably going to be around for some time, and people will always want to use them at least for previewing. The second answer is that there really is no substitute for doing things right in the first place. If the sizes of drawn objects can be predicted with to-the-pixel accuracy, you can get the most out of whatever printer you have paid for. If not, you will always have to settle for less than what you know the machine can do.

As a developer of precision applications like TEX drivers, I need a formal definition of how PostScript's drawing operators (primarily `fill` and `stroke`) should be rendered on raster devices, relating the high-level virtual machine defined by the language to the low-level hardware. Such a definition could itself be device-independent, speaking in terms of a target device with x dots per inch resolution horizontally and y dots per inch vertically. It could take the form of a published article, perhaps here in the TUGboat.

Aside from the fill-outline problem I have already mentioned, at least two other aspects would have to be addressed (and now I must apologize for using terms which will be unfamiliar to some readers). First, are CTM-transformed coordinates rounded or truncated in order to be converted to

integers for the hardware? (I recommend truncation since it is fast, and the user may change it to rounding by adding .5 to the translation components in the CTM.) Second, what is the precise orientation of bitmap characters generated by `imagemask`, with respect to the *current point*. I suggest that the current point should coincide with the extreme lower left-hand corner of the rendered image. That is, when the CTM is as described in the earlier example, the current point should identify the lower left-hand corner of a pixel, and this pixel should be overlaid with the lower leftmost pixel in the bitmap image. (A related issue is that when the coordinate system is inverted vertically, the current point should coincide with the extreme upper left-hand corner of the image — this does not appear to happen with our printer.)

A description of how the PostScript software is structured, distributed, and implemented on specific devices would also help applications developers to understand its operation. My guess is that the basic PostScript interpreter is provided by Adobe Systems, and each device manufacturer writes their own driver, but this is only a guess. Perhaps device manufacturers tell Adobe how their machine works, and later receive a fully-configured interpreter in machine-code form. Just how much does the device manufacturer do, and by implication, how much can be expected from a given PostScript-compatible product?

The issues raised in this article came up in the course of research into musical score setting using TEX. I have been working with a modified DVI-to-PostScript driver which allows inclusion of arbitrary PostScript code into TEX source material using the `\special` primitive. The idea is to use the power of PostScript to draw all the variable elements of musical material (e.g. note stems of variable length but fixed width). The lack of a definitive explanation of how PostScript's graphic primitives work at the device level forced me to spend a great deal of time writing tiny PostScript programs and examining the results — with a microscope! — to figure out what the printer was doing. I needed the microscope only to measure the *extent* of various inaccuracies in size and position — at 300 dpi the *existence* of these inaccuracies is immediately obvious to the naked eye.

References:

[1] Lamport, L. *TEX Output for the Future*. TUGboat **8,1** (April 1987).

[2] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Massachusetts. 1985.

Index to Sample Output from Various Devices

Camera copy for the following items in this issue of TUGboat was prepared on the devices indicated, and can be taken as representative of the output produced by those devices. The bulk of this issue has been prepared at the American Mathematical Society, on a VAX 8600 (VMS) and output on an APS-μ5 using resident CM fonts and additional downloadable fonts for special purposes.

- Apple LaserWriter (300dpi):
  ArborText advertisement, p. 110.
- TEXnology, Inc., advertisement, p. 103.
- Canon CX (300 dpi): Georgia Tobin, *The ABC's of special effects*, p. 15.
- Compugraphic 8600 (1301.5 dpi):
  TEXt1 advertisement, p. 106.
- HP LaserJet (300dpi):
  Personal TEX advertisement, p. 99.
- Linotronic 100 (1270 dpi):
  Design Science advertisement, p. 105.
- Kellerman and Smith advertisement, cover 3.
- Micro Publishing advertisement, p. 101.
- Xerox 4500 (300 dpi): Greek sample text, in Silvio Levy, *Using Greek fonts with TEX*, p. 22, as indicated.