Cahiers GUTenberg
n° 2 – Mai 1989

TEX and Graphics: The State of the Problem

Nelson H.F. BEEBE

Center for Scientific Computing
Department of Mathematics, South Physics Building
University of Utah
Salt Lake City, UT 84112, USA
Internet: Beebe@science.utah.edu

Abstract

Inclusion of graphics in documents typeset by TEX is not yet a satisfactorily solved problem, and no final *general* solution is in sight.

This paper surveys alternatives for insertion of graphics in TEX documents. It summarizes graphics primitives of several modern software systems, and shows how TEX has seriously deficient support for their direct incorporation in TEX itself.

Several alternatives for the production of graphics at the pre-processing (METAFONT), input (TEX), and post-processing (DVI) stages are considered, and their advantages and disadvantages are analyzed.

1 Introduction

Donald E. Knuth, TEX's author, began work on TEX on May 4, 1977 [45]. The initial design of languages for typesetting and font generation were implemented in programs written in the Sail language, available only on DEC-10 and DEC-20 computers.

The typesetting program has since become known as TEX78. That effort was described in a book in 1979 [37], and favorable user experience led to a complete redesign in a more widely-available programming language, resulting in the release in 1982 of TEX and METAFONT, now written in Web, a language that combines documentation with Pascal program fragments. TEX82, or just TEX, is now frozen, except for extremely rare bug fixes, in order that it can form a stable base on which computer-based typesetting can grow. The culmination of the nine-year long TEX Project at Stanford was the publication of the five-volume *chef d'œvre*, *Computers and Typesetting*, [38, 39, 40, 41, 42].

A significant related effort is the LATEX Document Preparation System [46], which builds upon the low-level typesetting machinery of TEX to provide a system that permits authors to write documents using familiar concepts of chapters, sections, subsections, paragraphs, figures, tables, bibliographies, indexes, and so on, with almost all the stylistic considerations of layout hidden in a document style file prepared by, or with the assistance of, a professional document designer. Change of a single keyword in the initial `\documentstyle` command is all that is required to change the style.

The great virtue of LATEX is that the author can concentrate on the job of writing, without worrying about considerations of page layout which are best left to people skilled in document design, and that the same skills used to prepare a document in one style carry over *unchanged* to any other document style.

TEX is now supported by numerous commercial vendors, and has been implemented on almost every commercially-significant computing system, from small personal machines like the IBM PC, Apple Macintosh, Atari, and Amiga, up to the Cray 2 supercomputer. The source programs for TEX and METAFONT, while trademarked by the American Mathematical Society, remain freely and openly available to anyone to implement, and non-commercial implementations of TEXware can generally be freely exchanged with one significant, and desirable, restriction: the programs may not be called TEX and METAFONT unless they can correctly process two devious test files, *trip* and *trap*, prepared by Knuth.

TEX is now being ever more widely used, with several vendors offering workstation publishing environments built upon TEX. Large publishing organizations, like the American Mathematical Society and TV Guide[1] [47] are converting to TEX. The Maryland Bar Association has recommended that TEX be used

[1] TV Guide publishes 20 million issues weekly in the United States and Canada, or about 10^9 issues yearly, and renders about 1 million different pages, bound into 5000 different 200-page magazines.

for the preparation of legal documents in that state. The choice of multiple sources of TEX on almost any machine platform, the high quality of the code and the output, and the availability of complete source code were all critical issues that led these organizations to choose TEX over traditional typesetting systems, or desktop publishing personal computers and workstations.

The TEX Users Group (TUG) celebrates its tenth birthday at Stanford University in the summer of 1989, with over 3400 members in dozens of countries. The existence of strong TEX users groups in other countries, such as GUTenberg, le Groupe francophone des Utilisateurs de TEX, to which this paper is addressed, is evidence that TEX is not only for the English-speaking world.

However, there is a thorn in this rosy picture. TEX was developed in the late 1970's, when traditional typesetting was moving away from hot lead type to optical type, and before personal computers, workstations, and laser printers were generally available. If there is a weakness in TEX's design, it is the rather primitive facilities for graphics, and it is that subject which I wish to elaborate on in this paper.

While traditional methods of preparation of typeset galleys, followed by paste-up of graphics images, are certainly possible with TEX, it is clearly desirable to move to an entirely electronic document preparation system. For researchers in academia, it is significant that the intercontinental electronic mail connections that have been available to a small body of people at a few large universities for a decade, just in the last two years are becoming available to almost any academic personnel in the Western World. Just in the last two months, gateways have been installed that now make trans-Atlantic file transfer and login sessions possible for a few sites. The way is rapidly opening when investigators at great geographic distances can collaborate on research projects almost as if they were close neighbors. For such activities, it is essential that complete documents can be transferred in electronic form.

Similarly, in the commercial publishing industry, particularly for periodicals that have large circulations distributed over a wide area, there is a strong need to be able to transmit complete publications electronically to local publishing houses for printing and local redistribution.

For further background on the design of graphs for publication, I urge the reader to look at Tufte's outstanding book [66]. Bentley and Kernighan have important things to say as well [7]; I will discuss their work in the later section *Languages for Typesetting Graphics*.

2 Primitives of Modern Graphics Systems

To understand the inadequacies of TEX for graphics, it is necessary to first understand what facilities are available in modern graphics systems.

Until 1976, graphics was a terrible mess. Almost every vendor of graphics devices developed different protocols for representing and communicating

graphics requests between the computer and the graphics device. Well over one hundred formats existed, and while certain large graphics hardware vendors offered software packages that could use their hardware, that software was usually proprietary, non-portable, and expressly written to match the characteristics of the particular device. For example, plot coordinates for one vendor's plotter in Europe had to be sent in millimeters, while the same plotter model sold in the United States expected coordinates in inches.

The significance of 1976 is that in May of that year, a Workshop on Graphics Standards Methodology was held in Seillac, France. That effort led to a first draft of a proposal for a graphics software standard in 1977 [1], and a final proposal in 1979 [2]; it is known as the SIGGRAPH CORE system. My own graphics system, <PLOT79> [4], is based on that proposal, and the year in the name commemorates the CORE system.

Since the CORE proposal, there have been two graphics systems standardized, the Graphical Kernel System, GKS, [14, 26, 27, 73] and the Programmer's Hierarchical Interactive Graphics System, PHIGS [8, 75]. Work is in progress on an extended PHIGS, known as PHIGS+ [74], as well as the marriage of PHIGS+ with the X Window System [31, 48, 49, 50, 59] in a system called PEX [56]. The X Window System is a significant development, because like TEX, its source code is freely available, the quality is generally high, it has been implemented on almost every commercial workstation architecture and two major operating systems, and because it is a network window system that decouples the client (output generating) program from the server (display generating) program.

Besides the graphics standardization efforts, there has been one other significant event—the development of the PostScript language by Adobe Systems for the control of printers and screen displays. PostScript has its roots in Salt Lake City, Utah, where in 1976, John Warnock at Evans and Sutherland Corporation began the design of a powerful extensible language for page description. In 1978, he moved to Xerox PARC Laboratories, where an offshoot of his work gave rise to the Xerox Interpress page description language. In 1982, he and Charles Geschke founded Adobe Systems Incorporated, which produced the language specification and first implementation of PostScript.

The first commercial printing device to provide PostScript was the Apple LaserWriter, introduced in early 1985. In the four years since that printer was introduced, PostScript has been adopted by most major computer manufacturers, been implemented in about two dozen printers, evolved into Display PostScript in the Sun NeWS and NeXT workstation windowing systems, and recently, independent implementations of PostScript have been produced by Imagen (now QMS/Imagen), Compugraphic, Control-C Software, Custom Applications Inc. Tektronix, the Free Software Foundation, and others. At least five books on the language have now appeared [25, 30, 29, 53, 57], and there are also two periodicals [28, 70] devoted to PostScript.

While PostScript is not intended for direct programming, one can certainly use it that way, and even a simple word processor has been implemented directly

in the language [57, p. 315]. However, it most cases, PostScript will be generated by other programs, and in practice, it is more of a device control language than a programming language. Nevertheless, its primitives are comparable in power to those of CORE, GKS, and PHIGS, and its inclusion of support for a wide range of high-quality scalable typesetting fonts makes it even more useful.

In the graphics system surveys in the following subsections, I will completely ignore input primitives of CORE, GKS, PHIGS, and the X Window System, since they are irrelevant to us here.

2.1 Minimal System

The simplest graphics system imaginable requires only four primitives; others can be built up from these.

- initialize
- move (invisibly)
- draw (visibly)
- terminate

At this level, there is no support for color, but it is most simply provided by the concept of a drawing pen, which might offer in addition to color, options for line endpoint, line width, line style, and line intensity. Most pen plotters, and most graphics terminals and raster printers (dot-matrix, ink jet, electrostatic, laser), can be supported with such a simple device model.

2.2 CORE

The CORE system was the first to offer a rich well thought-out set of graphics primitives, and supported both 2-D and 3-D plotting. My <PLOT79> implementation has extended these to 4-D homogeneous coordinates, since that provides full generality for support of parallel and perspective projections, scaling, shearing, rotation, and translation, and representation of points at infinity.

- initialize
- current point
- move, absolute or relative, 2-D or 3-D
- line, absolute or relative, 2-D or 3-D
- marker, absolute or relative, 2-D or 3-D
- polyline, absolute or relative, 2-D or 3-D

- polymarker, absolute or relative, 2-D or 3-D
- polygon, absolute or relative, 2-D or 3-D
- text
- attributes (pen, line width, line intensity, line style, polygon edge style, polygon interior style, visibility, highlighting, ...)
- camera model of viewing specification (view up, view reference point, view distance, view plane normal)
- world coordinate window selection
- normalized device coordinate viewport selection
- clipping control (front, back, window)
- world and image modelling transformation matrices
- inquiry of *all* user-settable values
- single-level segments
- device-independent output, including a graphics metafile (a generic graphics file)
- terminate

The CORE concept of 'current point' makes it possible to support the use of relative coordinates, and importantly, to write 'black-box' procedures that draw objects relative to the current point at procedure entry.

One of its most significant contributions that distinguishes it from all previous efforts is the provision of functions that can inquire what the current state of any user-settable value is. This critical feature makes it possible to write closed routines that can save the graphics state, reset it to something else for convenience, draw some object, and then restore the graphics state before returning.

Another strong point of CORE is its requirement for device independence; all primitives are available on all output devices. In practice, this means that on most operating systems, compiled user code is linked with a particular device library to produce an executable program that produces output for one particular device. On systems that support sharable link libraries, it is possible to delay the choice of output device until run time, so that a single executable program suffices for all output devices. <PLOT79> supports about 45 different output device libraries, and some commercial CORE implementations support even more.

Weak points of the CORE system are single-level segments (analogous to having a programming language with a CALL statement that may be used only in the top-level main program), and the generally cluttered and disorganized specification [2].

The first edition of Harrington's book [18] used the CORE system; the second edition [19] uses an abstraction of graphics primitives common to several systems. Foley and van Dam's widely used book [15] employs a CORE subset.

2.3 GKS

The Graphical Kernel System in some ways built on the lessons of the CORE System, and in other ways, seriously failed to learn from them. It rejects the concept of 'current point', eliminating the significant benefits of relative coordinates for primitives, and offers only 2-D primitives. In this author's view, it was a tragedy that GKS became an international standard. It set graphics development back years, by forcing vendors to spend development resources on an inferior system, instead of improving the defects of the CORE system while retaining full support for 3-D. Efforts to retrofit 3-D graphics on GKS cannot, in my view, be done cleanly, and should be abandoned.

Because of the restriction to 2-D, GKS primitives are simpler than those of the CORE system:

- initialize
- polyline absolute 2-D
- polymarker absolute 2-D
- *fill area*
- cell array
- text
- generalized drawing primitive
- attributes (pen, line width, line intensity, line style, fill area pattern and style, visibility, highlighting, ...)
- inquiry of *all* user-settable values
- single-level segments
- device-independent output, including a graphics metafile (a generic graphics file)
- terminate

Further descriptions of GKS can be found in the standard [73] and in three textbooks [14, 26, 27].

2.4 PHIGS

The Programmer's Hierarchical Interactive Graphics System is a much more satisfactory descendant of CORE than GKS is. PHIGS offers full 3-D capability, output device independence, enhanced support for raster devices and interactive graphics, and unrestricted segment nesting. In addition, it provides a powerful display selection mechanism, wherein graphics segments can be given user-defined tags, and then display can be requested of all segments belonging, or not belonging, to a particular tag set. As an example, an architectural application might be to display a building plan with all the concrete parts drawn, and wooden and glass parts eliminated.

PHIGS [75] is relatively new, and evolution to PHIGS+ [74] and PEX [56] is underway. There is only one short textbook describing PHIGS [8].

The PHIGS primitives are available in both 2-D and 3-D forms, and include:

- initialize
- polyline
- polymarker
- text
- fill area
- fill area set
- cell array
- generalized drawing primitive
- attributes (line type, line width, polyline color, marker size, marker type, polymarker color, fill area interior and edge color, style and pattern, color model, visibility, highlighting, ...)
- local and global world and image modelling transformation matrices
- terminate

PHIGS actually has many more primitives than these; for a detailed list, see [8, Appendix II]. PHIGS+ [74] adds support for features of high performance graphics workstations, including lighting, shading, depth cueing, and curve and surface primitives.

The generalized drawing primitive of both GKS and PHIGS is provided to give the programmer access to graphics facilities (usually in output device hardware) not otherwise provided for by the standards; they include such things as circle, arc, and curve generators. However, in portable software, or software that must produce output for more than one device, such extensions cannot be used.

2.5 X Window System

The X Window System (X for short) is the result of research, begun in 1984, at MIT Project Athena carried out in conjunction with IBM and DEC. In the last five years, it has gone through 11 versions, with the final one, X 11 Release 3, supposed to be the one that is frozen to serve as a platform for further vendor development. Only by late 1988 did the first books on programming with X appear; there are now five such volumes [31, 48, 49, 50, 59]. X runs on most UNIX systems, VAX VMS, and (in a limited fashion) on PC DOS.

It was a condition of MIT Project Athena that all code developed for the X Window System be publicly available for a nominal distribution cost. The `COPYRIGHTS` file in the X distribution states

> The MIT distribution of the X Window System is publicly available, but is *not* in the public domain. The difference is that copyrights granting rights for unrestricted use and redistribution have been placed on all of the software to identify its authors. You are allowed and encouraged to take this software and build commercial products.

No vendor can therefore get a strangle hold on it, and all vendors are free to take the source code and adapt it to their own hardware, making internal changes to improve efficiency, or take advantage of their particular machine architecture.

The free availability of source code for X means that it provides a *vendor-independent* window system, freeing the user of dependence on a single vendor, and greatly enhancing the potential portability of code written to use the X Window System. However, X is written in C and uses long case-sensitive names and C data structures; no interface has been defined to any other programming language, such as Fortran or Pascal.

Importantly, the X Window System was designed to separate the notions of generation of the data for the window display, the actual display of that window, and the management of windows. In particular, generation and display can be on separate machines, since all communication between the two processes goes through the X library, which uses the X protocol on top of a network protocol, like TCP/IP or DECnet.

These are exceedingly important design features. They mean that, in the words of its developers, X *provides mechanism, but not policy.*

Several different window managers have been developed, based on both overlapping and tiled (non-overlapping) models, and users can run any one they choose.

A program generating graphical data for X might run on a supercomputer, while the display is viewed on a low-cost desktop workstation, possibly thousands of miles away. That situation has existed since 1987 with the NASA Cray 2 at Moffett Field, CA and NASA researchers at Langley, VA.

It is possible to design a *terminal* that displays X Windows, without offering an operating system environment. There are now about a half-dozen vendors with such products, ranging in price from about U. S. $600 to $2000 (comparable to terminal pricing of only 5 years ago). This makes it possible to reduce even further the cost of providing an extremely productive workstation environment, which otherwise currently represents an investment of U. S. $5000 to $30000 per user, depending on the power of the workstation.

Is X the final solution? No, certainly not. Further research in software and hardware will result in many new ideas that lead to new products. X will certainly provide a solution for most of the next decade. It does have a fundamental limitation, in that it is designed around the notion of *bitmaps*; all graphical operations must have knowledge of the resolution of the window they work in, and display results only to that resolution. Typical current screen displays have 480 to 1200 dots on a raster line, with a few systems reaching as many as 2048; on a large 19-inch diagonal monitor, that represents about 100 dots/inch (40 dots/cm). Popular laser printers today have 300 dots/inch (118 dots/cm), and phototypesetters have 1500 to 5300 dots/inch (600 to 2100 dots/cm). That higher quality is completely inaccessible with the X Window System.

Here is a list of the X graphics primitives; these currently are restricted to 2-D, but the PEX extensions [56] add 3-D support.

- initialize
- point and polypoint
- line and polyline
- rectangle and polyrectangle
- filled rectangle and polyrectangle
- arc and polyarc
- filled arc and polyarc
- filled polygon
- clip mask bitmap and origin
- text
- attributes (line type, line width, line endpoint style, line join style, foreground and background color, tile pattern, stipple pattern, fill pattern, Boolean drawing operation, plane mask, polygon fill rule, ...)
- terminate

X does not support the CORE concept of 'current point', but its polyxxx primitives have an argument that permits coordinates after the first to be relative to the preceding point. Because X is a windowing system, it has many more primitives that are relevant to window operations.

2.6 PostScript

The PostScript language is best described in the red, blue, and green volumes produced by Adobe Systems [30, 29, 53]. Because it is intended for page description, rather than graphics system standardization, it is quite different in many respects from CORE, GKS, and PHIGS. Nevertheless, it is useful to compare its basic graphics primitives with those of the other systems. All of its primitives are restricted to 2-D only.

- initialize
- current point
- arc
- Bézier curve
- move, absolute or relative
- line, absolute or relative
- path
- text (typeset quality)
- pattern and halftone area fill
- attributes (color, halftone, line style, line width, line endpoint and join styles)
- clipping path
- terminate

This list is necessarily a great simplification. PostScript is a programming language with named scalar and array variables and procedures, integer and floating-point arithmetic, bitwise and arithmetic operators. In addition, it provides user-defined dictionaries, or symbol tables, which make it possible to override the definition of *any* previously-defined name, and thereby arbitrarily extend the language, or even change its meaning.

The typeset-quality text provided by PostScript is the first obvious distinction from the other graphics systems. The notion of a *path*, which may be

either stroked or filled, or used as a clipping path on other objects, generalizes the line, polyline, marker, polymarker, and generalized drawing primitives of earlier systems. In addition, objects on the path need not be just points or line segments. They can be curve segments, and importantly, they can be procedures that are invoked dynamically when the path is traced. This facility offers enormous power. The support for color and halftoning, and support for arbitrary rescaling and resampling of raster images with one or more bits per pixel, are truly remarkable.

It seems quite clear that PostScript-based window systems will eventually replace bitmapped window systems like the X Window System, because they offer much greater power, and importantly, independence from output device resolution.

If PostScript has a weakness, it is that its painting model is opaque; that is, any object, whether black, white, or colored, when drawn on top of another object in the page image, completely replaces it. This feature can make preparation of complex images rather tricky. Further details can be found in [57, p. 3, p. 201].

Clearly, the future growth of graphics with systems like PHIGS, PEX, NeWS, and PostScript is going to be exciting.

3 Why Graphics is Hard in TEX

In the preceding section, I gave an overview of what facilities modern graphics systems can provide. A limited understanding of those facilities is essential to appreciate why graphics is hard in TEX.

TEX was perhaps developed a decade too soon, because it based its capabilities on typesetting models of 1978, and those models have radically changed since then. TEX is unparalleled in its ability to set beautiful text, particularly mathematical text, and its hyphenation, line breaking, and page breaking algorithms are very significant advances in publishing.

METAFONT [40, 41], TEX's sister, provides a way for a font designer to prepare entire families of fonts parametrized in such a way that the family resemblance is retained, that a computer can do the tedious work of generating the filled outlines, and that relatively simple parameter changes can be made to account for the differing imaging characteristics and resolution of output devices. Knuth's Computer Modern font family [42] is clearly a masterpiece using METAFONT, but it is also a first effort. Many decades will pass before we are likely to have other widely-used font families produced by other type designers with METAFONT. For further discussions of other approaches to computer-aided font design, see Karow's excellent book [33].

TEX solves the problem of placing *characters* on the page, but what about other, non-textual, marks. Well, TEX has horizontal and vertical rules (variable-width lines). And that is it! There are no diagonal lines, no line styles, no circles,

no arcs, no Bézier or B-spline curves, no halftones, no color, no fill areas, no fill patterns, no floating-point arithmetic, no 3-D.

While TEX can be made to handle right-to-left and top-to-bottom typesetting of Semitic [36, 52] and Oriental [58] languages, it is impossible for it to rotate character boxes to obtain text in any direction other than what the font designer provided for.

TEX's arithmetic is scaled fixed point, and its expression syntax is about as painful to write as that of Cobol; with suitable variable declarations, the simple assignment `x = 3*y + 4/z` must be written as

```
\x = \y
\multiply \x by 3
\temp = 4
\divide \temp by \z
\advance \x by \temp
```

I have tried to write a general expression parser for TEX that would let one shorten this to something like

```
\x=\expr{3*\y + 4/\z}
```

but have yet to succeed in that goal. Without it, writing macros that require significant calculation of expressions is exceedingly tedious and error-prone.

Aside from the four arithmetic primitives, TEX offers nothing in the way of elementary function support. It would be a nightmare to program implementations of sin, cos, sqrt, log, and so on in scaled fixed point arithmetic with the expression syntax requirements illustrated above. This is regrettable, because those functions are all available in the host language that TEX is written in, but the hooks to access them are absent. Presumably, Knuth's goal of complete machine independence to ensure identical output interdicted these facilities. Nevertheless, it would have been possible to provide a software emulation of a decent floating-point arithmetic system, such as the IEEE one [51], that could have preserved machine independence. TEX itself actually uses floating-point arithmetic internally for certain operations; METAFONT uses integer arithmetic exclusively.

TEX has no real concept of 'current point' accessible to the programmer. It certainly maintains such a quantity internally, but it is not available as a standard primitive. This necessitates memory-consuming workarounds, like using horizontal and vertical kerns to position objects in boxes.

Similarly, TEX has no way for the programmer to refer to an absolute page coordinate. The output routine collects the page image into a box and sends it to the DVI file, but there is no hook called `\everyshipout` to attach a function that could provide absolute page addressing.

TEX has no notion of graphics planes, or of writing modes; modern raster graphics hardware generally provides the 16 possible Boolean operations between source and destination pixels. In TEX, the mechanism for mapping of

objects to a page bitmap is curiously left to the DVI driver; most choose either a Boolean *or* operation, or a replacement operation, so overlapping objects simply overwrite one another. There are no white objects that can be used for erasure.

What does TEX offer to resolve these deficiencies? Almost nothing, except a `\special{}` command, that like the GKS and PHIGS generalized drawing primitive, provides an escape mechanism for getting a device-*dependent* request into its otherwise device-*independent* output file, the DVI file.

4 TEX and the \special{} Command

One of the great virtues of TEX is that it was very carefully designed to perform identically on all systems, independent of the underlying host computer architecture. In particular, its DVI file output must be identical, bit for bit, on all machines, permitting TEX to be used on a variety of machines in the publication process.

Translator programs, called DVI drivers, then have the job of interpreting this compact binary file that encodes TEX's typesetting results, merging typesetting commands with character bitmaps from font files to produce output for particular devices. While the speed, resolution, and quality of these devices may differ, the document layout, in particular, the line breaks, page breaks, and hyphenations, will be the same on all of them.

Yet, if one takes advantage of the `\special{}` command, the DVI file is no longer independent of the output device. The argument of the `\special{}` command is not interpreted by TEX, and is therefore solely a matter between the user, and the DVI driver program. Scores of such drivers exist, and I have argued [5] for maintaining all drivers in a common base with shared code, implemented portably to run on multiple operating systems, and with the code in the public domain. Version 3.0 of my DVI driver family currently under development consists of more than 45,000 lines of C code with over 200 pages of documentation, and supports more than 30 output devices on six different operating systems. Over 1000 sites in 28 countries are using earlier versions of this family, testifying, I believe, to the soundness of my principles.

Thus, at present, there exists little chance of compatibilities in `\special{}` commands between DVI drivers. The issue is complex, and a committee of the TEX Users Group has been working for almost two years now on the problem of DVI driver, and `\special{}` command, standardization. I am a member of that committee, and I do not see an end in sight in the near future. The committee uses the wonderful medium of network electronic mail to exchange ideas and commentaries, and the accumulated mail volume as of early April, 1989, is 615 Kbytes, corresponding to about 200 tightly-spaced pages of text.

The problem of driver dependence of `\special{}` command is brought home in this article, because I cannot illustrate the incorporation of a graphical image

here for you, because I have no control over what DVI driver the publisher of this article will use!

The inclusion of graphics files in response to requests from `\special{}` commands introduces a difficult problem because of the great variety of graphics file formats that exist. The commonest vector file formats in practice are those of popular graphics terminals and plotters, such as Tektronix, HPGL, and PostScript formats. There are about as many raster file formats as there are raster output devices, and they are additionally resolution-dependent. Graphics standards have included generic graphics files, called *metafiles*, but the amount of code required to parse such a file is larger than most existing DVI drivers.

Aside from the `\special{}` command, how can TEX support electronic incorporation of graphics? There appear to be four distinct avenues, all of which are likely to be useful for certain purposes. I choose to locate these in preprocessing, input processing, output processing, and post-processing steps.

5 TEX and Graphics Pre-processing

By pre-processing, I mean the stage before TEX begins to read the user's manuscript. Since its input must be well-formed according to definite rules described in the TEXbook [38], the only opportunity here is to take advantage of Knuth's design decision that TEX is entirely *ignorant* of the shape or appearance of font characters. It only knows their measurements (width, height, descender depth, etc.), which it obtains from the TFM (TEX Font Metric) files produced by the font generating program.

Since TEX can be directed to place character images anywhere on the page with great precision, and cares little about how big those images are, we may elect to store pictures in them.

The fonts most commonly used with TEX come from METAFONT, and METAFONT was designed to be able to *draw* characters. Thus, even though METAFONT [40, 41] has mostly been used for font design [42, 61, 62, 60, 63, 65, 64] and font generation, we should examine it for graphics capabilities, so we can compare it with the graphics systems discussed earlier. Like PostScript, METAFONT is a true programming language, with scalar and array variables, procedures, and operators for arithmetic, bit manipulation, and drawing. Here is a summary of METAFONT primitives:

- initialize
- move
- line
- polyline
- curve (Bernshteĭn polynomials, of which Bézier curves are a special case)

- pen shape
- pen 'color' (black to draw, white to erase)
- path
- terminate

METAFONT has no concept of true color, or of line style (solid vs. dashed). Line thickness and endpoint style can be controlled by the pen shape.

As in PostScript, METAFONT's path notion is a powerful generalization of polylines and polygons, and can contain procedural objects in addition to simple points, lines, and curves. Unlike PostScript, METAFONT draws with pens whose shapes can be defined by the user, subject to the restriction that they can be represented as convex polygons. This is in fact not a serious limitation, because a concave polygonal pen could be simulated by multiple convex polygonal pens moving on parallel paths, some drawing, and some erasing.

METAFONT's output is not designed to be device independent; it is a set of characters in a font, where the characters are compactly represented in the font file as run-length encoded bitmaps. However, by suitable parametrization, it is rather simple to get METAFONT to produce characters in any desired size for any desired output device resolution.

PostScript goes to great lengths to hide the underlying raster representation by encouraging programmers to use device-independent coordinates and paths, leaving the realization of the representation as a bitmap to the very end. PostScript has no primitives at all for accessing the bitmap; it must be viewed as a 'write-only' object.

METAFONT, on the other hand, admits from the start that it is drawing on a bi-level bitmap.

There are advantages to both approaches. For METAFONT and font design, where the resolution of common current output devices is marginal (50–200 dots/cm), knowledge of the raster resolution is essential to prepare acceptable character descriptions.

PostScript presupposes that the hard work of font design for the output device resolution has already been carried out, and in fact, current implementations of PostScript do not define the format of internal fonts, and use encrypted representations for them, even though the rest of the language has been specified publicly in [29].

One very reasonable approach for preparation of graphics to be included in TEX documents is therefore to use METAFONT to produce font characters that encode the picture as a bi-level bitmap. As noted earlier, this representation is dependent on output device resolution, but can be easily scaled by METAFONT for other resolutions. We can therefore achieve document portability, at the expense of perhaps having to generate several different METAFONT font representations, one for each output device resolution that will be used by the

DVI drivers. Since METAFONT program representations are printable text, just like TEX input files, they can be transmitted electronically with equal ease. Any properly-written DVI driver should be able to correctly handle TEX output with large font characters, so the only precaution the user must take is to ensure that the resolution of the METAFONT-generated pictures matches that of the output device.

Naturally, we should not expect that individuals learn METAFONT programming just so they can create pictures in their TEX documents. Instead, existing graphics packages based on the CORE, GKS, and PHIGS models could be modified to support yet another output device—METAFONT. Machine-independent translators for common graphics formats (e.g. Tektronix and HPGL) could be developed to permit conversion of existing graphics files to METAFONT input.

The two major difficulties here are (a) how to handle color, and (b) how to handle text characters that the graphics device is expected to generate itself.

Difficulty (a) is present in other approaches as well, and best ignored for now.

Difficulty (b) probably does not have a satisfactory solution that will produce output essentially identical to the original output device. For example, the HPGL language used on Hewlett-Packard pen plotters, and clones, provides for hardware characters that contain smooth curve segments generated by an unspecified method. One could of course generate the plotter's entire character repertoire, then redigitize it into vector form, but this leaves open the question of font copyright violation. Some graphics systems, such as my own <PLOT79> system, reject the use of hardware-dependent character sets in their quest for output device independence, and use vector character descriptions, such as those developed by A. V. Hershey at the U. S. Naval Weapons Laboratory [20, 21, 22, 23, 24, 69] and placed in the public domain. For such systems, the output graphics files contain no text requests, since all text characters were reduced to vectors at a higher level.

To my knowledge, no serious work in this direction has yet begun, although we have discussed it in the TUG DVI Standards Committee exchanges.

There will also be an as-yet-unknown limit to the complexity of graphics images that can be produced this way, due to the internal design of METAFONT.

6 TEX and Graphics Input Processing

The second approach to support of graphics in TEX is to implement a limited set of graphics primitives as TEX macros. This is certainly a common approach, and has enjoyed some limited success. LATEX picture mode [46, pp. 101–110] is probably the most widely available system. It provides lines (in a restricted set of two dozen slopes), vectors (lines with an arrowhead), rectangles (boxes), ovals, empty and filled circles, and with the `bezier` document style option, quadratic

Bézier curves. Its `\put` and `\multiput` commands allow easy positioning of objects, which may themselves be picture objects, allowing hierarchical decomposition of pictures into subpictures. The units of the coordinate system, and therefore, the size of the pictures, can be easily scaled with the `\unitlength` command.

Picture mode has many limitations, however. There are only two line widths available, and only one kind of line dashing is available, and then only in a rectangle. We will illustrate later how dotted lines can be obtained in picture mode. Special fonts are used for the diagonal lines, ovals, circles, and arrowheads, limiting the number of slopes and sizes. Coordinate scale must be the same horizontally and vertically. Changing the picture scale with `\unitlength` does not scale text objects. Coordinates are restricted by TEX's underlying fixed point representation of dimensions, permitting values with integer values up to $2^{14}-1$ (16383) and fractions in units of 2^{-16}, so overflow and underflow can sometimes be a problem (TEX catches such errors, so they can be worked around). There are no filled areas, so the common publishing technique of shading text boxes for emphasis is unavailable. Curves must be produced by juxtaposition of small boxes; this introduces resolution dependence, as illustrated in the Bézier curve figures below. Font characters cannot be scaled, rotated, or otherwise transformed, restricting labelling flexibility. Picture mode has no color primitives. SLiTEX provides limited color support which could be used in picture mode; see the later section *Color and TEX*.

Here are some examples that I produced for our *Local* LATEX *Guide* to illustrate the `bezier` style option, which appeared after the LATEX book was published, and is therefore not described there. The figures show the exact sequence of picture mode commands used to produce each image.

```
\begin{center}
  \setlength{\unitlength}{0.025in}
  \begin{picture}(100,105)

    \bezier{150}(0,25)(0,0)(25,0)
    \put(0,25){\line(0,-1){25}}      \put(0,0){\line(1,0){25}}
    \put(0,25){\circle*{2}}          \put(0,0){\circle*{2}}
    \put(25,0){\circle*{2}}
    \bezier{150}(100,40)(40,0)(100,0)
    \put(100,40){\line(-3,-2){60}}   \put(40,0){\line(1,0){60}}
    \put(100,40){\circle*{2}}        \put(40,0){\circle*{2}}
    \put(100,0){\circle*{2}}
    \bezier{150}(75,100)(95,95)(100,75)
    \put(75,100){\line(4,-1){20}}    \put(95,95){\line(1,-4){5}}
    \put(100,75){\circle*{2}}        \put(95,95){\circle*{2}}
    \put(75,100){\circle*{2}}
    \bezier{150}(0,75)(50,50)(25,100)
    \put(0,75){\line(2,-1){50}}      \put(50,50){\line(-1,2){25}}
    \put(25,100){\circle*{2}}        \put(50,50){\circle*{2}}
    \put(0,75){\circle*{2}}

  \end{picture}
\end{center}
```

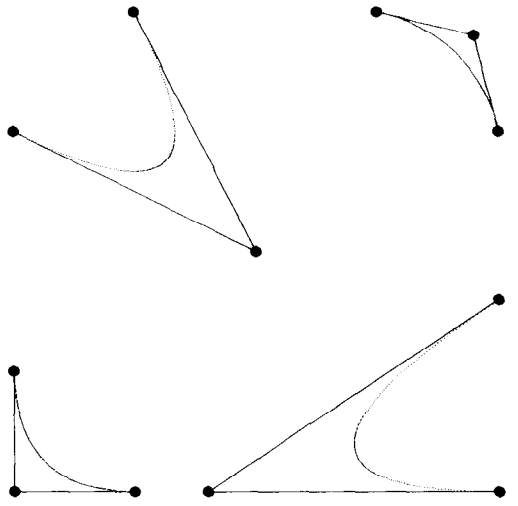


Figure 1: Quadratic Bezier curves (150 dots)

```
\begin{center}
  \setlength{\unitlength}{0.03in}
  \begin{picture}(100,110)

    \bezier{50}(0,0)(50,100)(100,0) \put(80,95){\makebox(10,10)[r]{25}}
    \bezier{100}(0,0)(50,80)(100,0) \put(80,75){\makebox(10,10)[r]{50}}
    \bezier{150}(0,0)(50,60)(100,0) \put(80,55){\makebox(10,10)[r]{75}}
    \bezier{200}(0,0)(50,40)(100,0) \put(80,35){\makebox(10,10)[r]{100}}
    \bezier{250}(0,0)(50,20)(100,0) \put(80,15){\makebox(10,10)[r]{125}}

    \put(0,0){\circle*{4}}    \put(100,0){\circle*{4}}

    \multiput(50,100)(0,-20){5}{\circle*{2}}

  \end{picture}
\end{center}
```

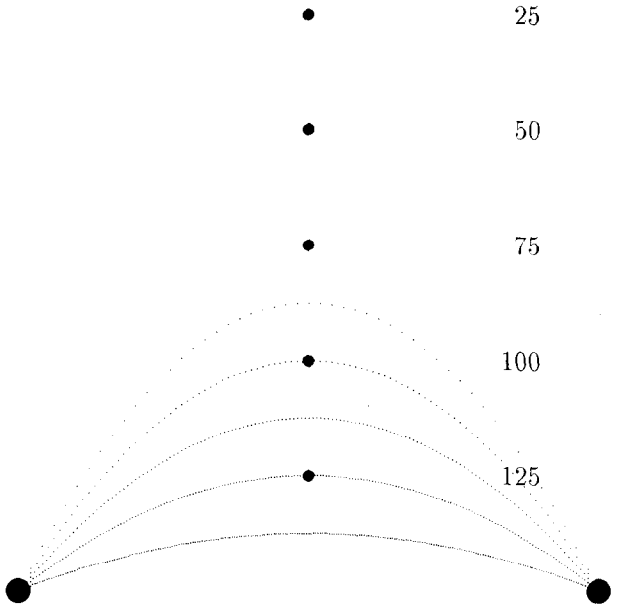


Figure 2: Quadratic Bezier curves with varying dot counts

```
\begin{center}
  \setlength{\unitlength}{0.03in}
  \begin{picture}(100,110)

    \bezier{200}(50,50)(0,100)(50,100)
    \bezier{200}(50,100)(100,100)(50,50)
    \bezier{200}(50,50)(0,0)(50,0)
    \bezier{200}(50,0)(100,0)(50,50)

    \put(0,0){\circle*{1}}     \put(50,0){\circle*{2}}
    \put(100,0){\circle*{1}}   \put(50,50){\circle*{4}}
    \put(100,100){\circle*{1}} \put(50,100){\circle*{2}}
    \put(0,100){\circle*{1}}

  \end{picture}
\end{center}
```

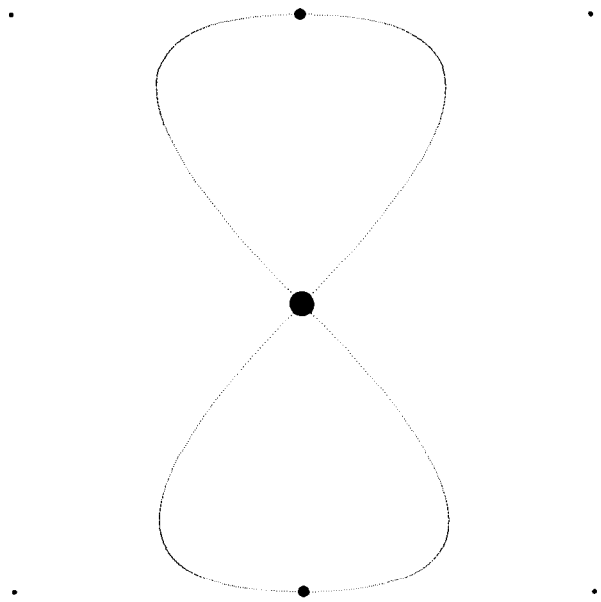


Figure 3: Quadratic Bezier figure eight constructed of four segments. Control point diameters are proportional to usage counts.

A more complex example is illustrated in the next pair of figures, which show the various LaTeX page layout parameters. I developed these in October 1986, and they should now be available in all standard TeX distributions as a file named **page-layout.tex**, or something similar. That file contains about 250 lines of macros definitions, and 210 lines of their use in picture mode, which is rather too long to include here.

Figure 4: LaTeX single-column page layout. The actual proportions correspond to parameter values in the 11pt BOOK document style. Note that standard-conforming DVI drivers are required to place the TeX upper-left page corner one inch over and down from the corner of the physical output page. This figure is scaled to 50% of actual page size. It was produced on April 17, 1989 at 12:41.

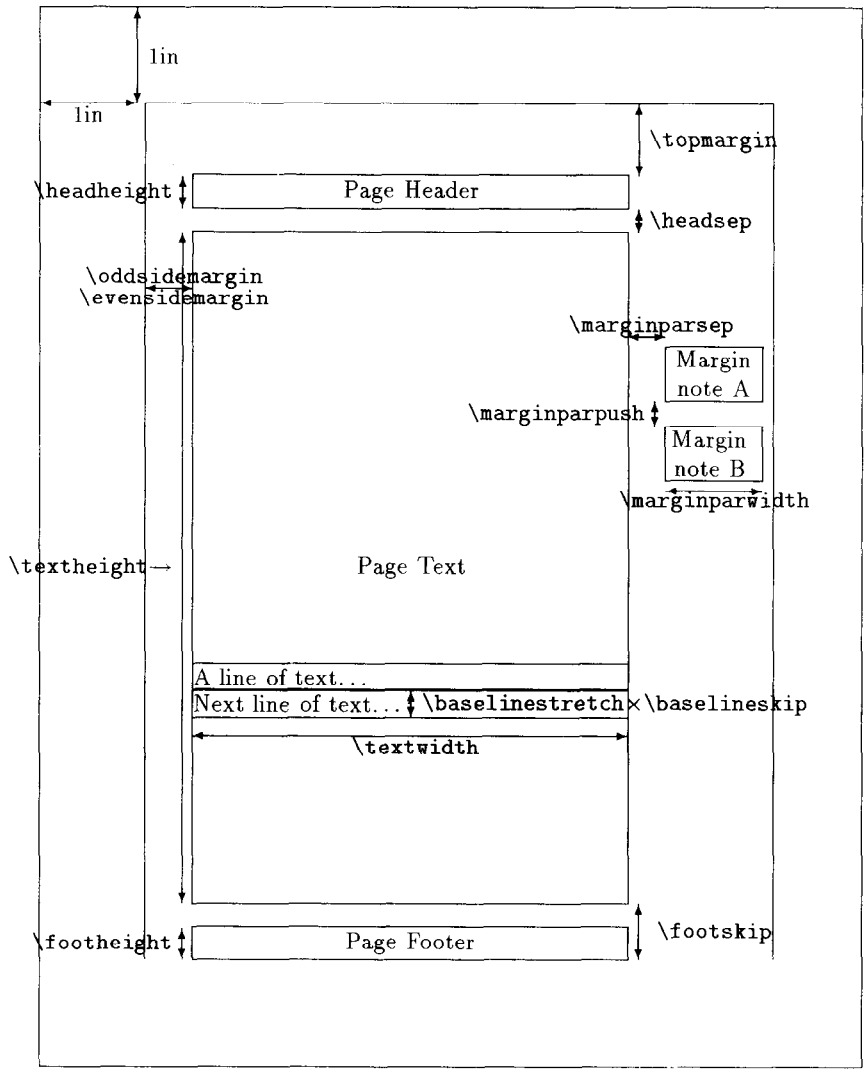
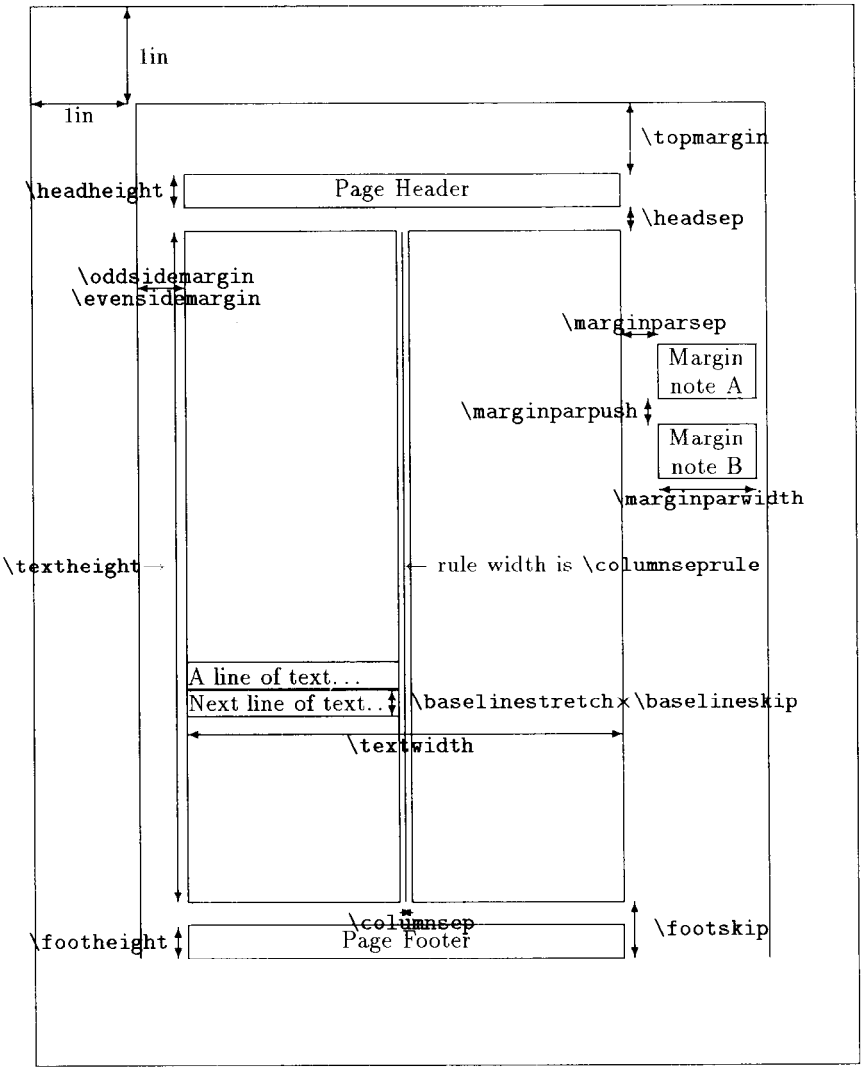


Figure 5: LaTeX double-column page layout. The actual proportions correspond to parameter values in the 11pt BOOK document style. Note that standard-conforming DVI drivers are required to place the TEX upper-left page corner one inch over and down from the corner of the physical output page. This figure is scaled to 50% of actual page size. It was produced on April 17, 1989 at 12:41.



With some additional effort at macro writing, picture mode can be extended to make certain types of common graphics illustrations fairly painless to produce. For a recent talk on workstations, I prepared some bar chart macros to make such figures. A set of about 130 lines of macros are needed to define the basic `\HBAR` and `\VBAR` macros, in terms of which are defined simpler-to-use macros like `\PERFORMANCE` and `\REVENUES` illustrated below.

For bar charts, it is desirable for the user to be able to enter the data in the original raw form, which in general will have different horizontal and vertical scale. My macros support this. In the figures, one axis is arbitrarily given a scale of 100, to facilitate positioning of bars for different vendors, and the other axis has a scale depending on performance or sales revenues.

The major limitations of these macros are that fractional values for bar lengths are not supported, and that excessively large values may require rescaling to avoid exceeding TEX's fixed point number range limitations.

Here now are two bar chart examples illustrating horizontal and vertical bar styles.

```
\setlength{\unitlength}{0.008in}
\newcommand{\PERFORMANCE}[3]{\HBAR{0}{#2}{#1}{5}{#3}{#1}}
\HEIGHT=700
\WIDTH=550

  \begin{picture}(\WIDTH,\HEIGHT)(0,0)
    \YMAX=90
    \XMAX=100000
    \thicklines
    \put(275,650){\makebox(0,0){\Large\bf Dhrystones per second}}
    \put(275,630){\makebox(0,0){Source:
        Byte, April 1989, p. 11}}
    \put(275,610){\makebox(0,0){Source:
        UNIX Review, December 1988, p. 65}}
    \put(275,590){\makebox(0,0){Source:
        UNIX Review, January 1989, p. 102}}
    \put(275,570){\makebox(0,0){Source:
        Research \& Development, March 1989, p. 53}}
    \put(0,0){\framebox(\WIDTH,\HEIGHT){}}
    \labellingtrue
    \dottedlinestrue
    \VLINES{0}{\YMAX}{10000}{10000}{90000}
    \PERFORMANCE{1640}{80}{\shortstack{\\VAX\\11/780}}
    \PERFORMANCE{3246}{70}{\shortstack{\\Sun\\3/160}}
    \PERFORMANCE{13043}{60}{\shortstack{\\Cray\\2}}
    \PERFORMANCE{18530}{50}{\shortstack{\\Cray\\X-MP/48}}
    \PERFORMANCE{19230}{40}{\shortstack{\\Sun\\4/260}}
    \PERFORMANCE{24876}{30}{\shortstack{\\MIPS\\M/120-5}}
    \PERFORMANCE{31250}{20}{\shortstack{\\IBM\\3090/200}}
    \solidbarfalse
    \PERFORMANCE{85000}{10}{\shortstack{\\Intel\\80860}}
  \end{picture}
```

Figure 6: Input for horizontal bar chart.

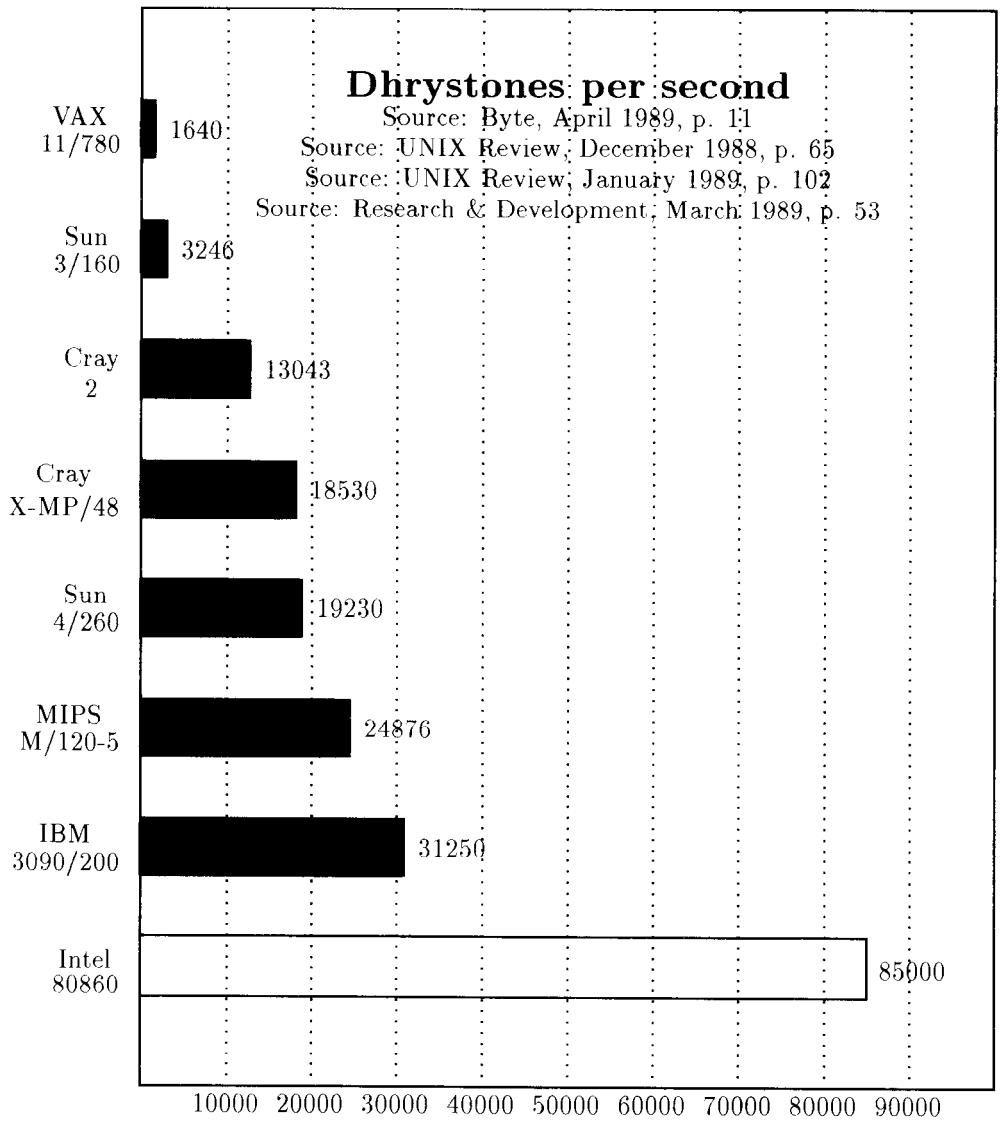


Figure 7: Horizontal bar chart illustrating Dhrystone string manipulation benchmarks. Larger values mean better performance.

```
\setlength{\unitlength}{0.009in}
\newcommand{\REVENUES}[3]{\VBAR{#1}{0}{5}{#2}{\$#2}{#3}}
\HEIGHT=700
\WIDTH=550

  \begin{picture}(\WIDTH,\HEIGHT)(0,0)
    % Data ranges (x = 10*company number, y = megabucks)
    \XMAX=90
    \YMAX=1300
    \thicklines
    \put(275,650){\makebox(0,0){\Large\bf
        1988 Workstation Revenues}}
    \put(275,620){\makebox(0,0){\Large\bf
        (U.~S. \$1,000,000)}}
    \put(275,590){\makebox(0,0){Source:
        Digital News 6-Mar-89 p. 85}}
    \put(0,0){\framebox(\WIDTH,\HEIGHT){}}
    \dottedlinestrue
    \labellingtrue
    \HLINES{0}{\XMAX}{0}{100}{1100}
    \REVENUES{10}{1165}{Sun}
    \REVENUES{20}{765}{DEC}
    \REVENUES{30}{695}{HP}
    \REVENUES{40}{555}{Apollo}
    \REVENUES{50}{275}{Intergraph}
    \REVENUES{60}{180}{SGI}
    \REVENUES{70}{105}{IBM}
    \REVENUES{80}{370}{Others}
  \end{picture}
```

Figure 8: Input for vertical bar chart.

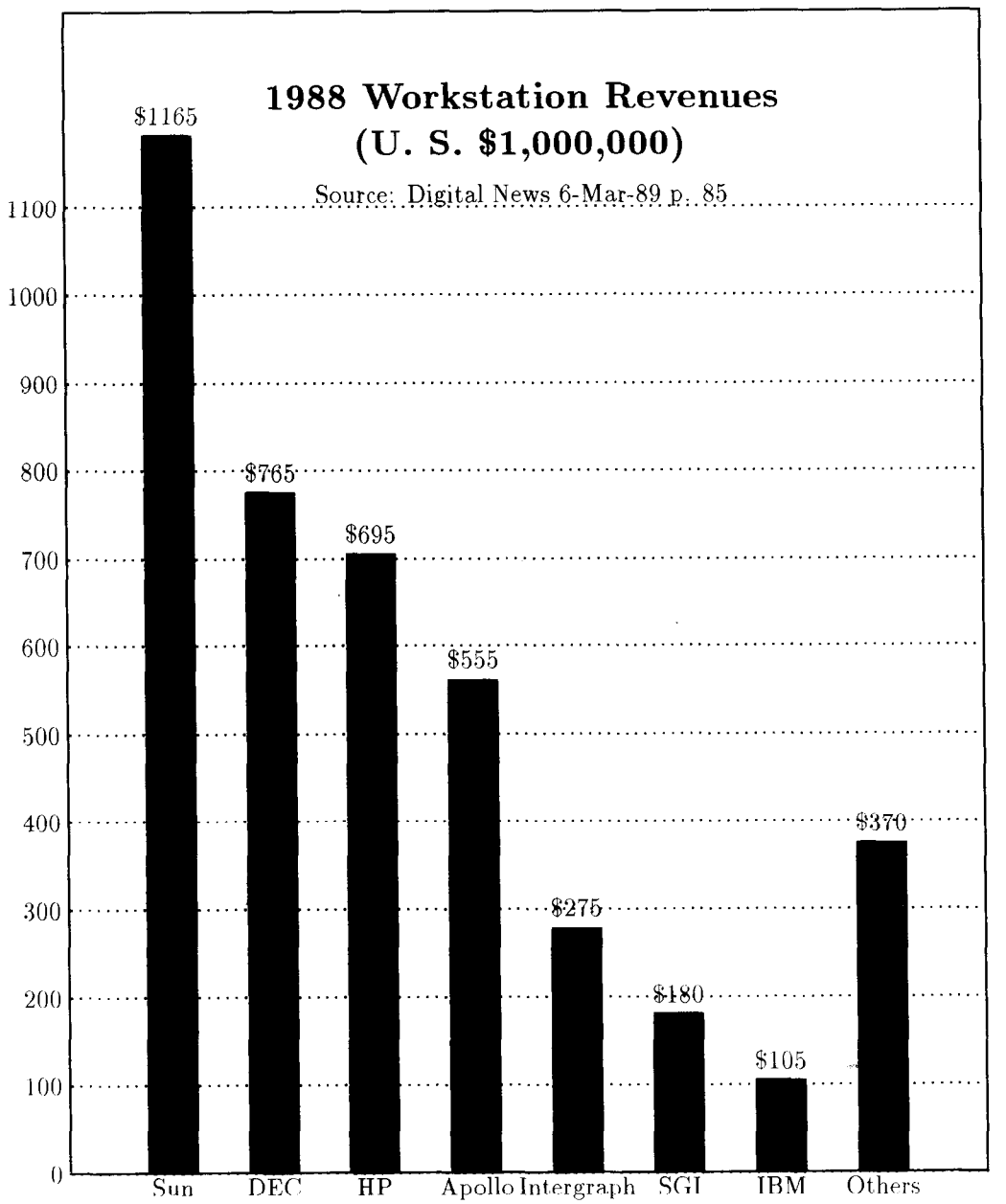


Figure 9: Vertical bar chart.

Normally, one does not bother with a background grid on bar charts, because it is visually distracting. The `\HLINES` and `\VLINES` macros provide a facility for drawing a grid; the lines will be solid, unless `\dottedlinestrue` is specified. How are the dotted lines created? Simple—they are quadratic Bézier curves with the 3 control points at the start, middle, and end of a straight line. The LaTeX implementation in `bezier.sty` requires the user to specify the number of dots to draw along the path; I chose 100 dots in the macro definitions. When I tried 150 for the dot count, the second bar chart overflowed TeX's memory. This suggests that a normal-sized TeX can handle only slightly more than 1000 Bézier dots. That rather small number is caused by the need to represent them inefficiently as kerned boxes.

To further investigate TeX's handling of Bézier curve dots, I turned on the option `tracingstats` [38, p. 300] on pages containing only Bézier curves. That revealed that TeX uses 15 words (60 bytes) of internal memory for each dot (actually a square 0.4pt rule); by comparison, the compact Tektronix vector encoding format averages about 3 bytes per coordinate pair in a polyline in typical hidden-line and contour plots. Standard TeX has 30000 words of memory [39, p. 6], which imposes a strict limit of 2000 Bézier dots when there is nothing else on the page. Each Bézier dot takes 18 to 19 bytes in the output DVI file (positioning and *setrule* commands), which is about 6 times as verbose as the Tektronix encoding.

Were we to use Bézier curves with dense dots to create diagonal lines on a 300 dot/inch laser printer, a *single* line as wide as the page would fill TeX's memory!

This situation could be improved greatly if TeX had a general vector primitive. In fact, an even better choice would be a non-uniform rational B-spline curve primitive, which is used in the Alpha-1 Computer-Aided Design and Modelling System developed at the University of Utah. That single primitive can represent straight lines, general space curves, and exact circles and arcs.

If TeX had a grey-scale primitive that supplied an intensity for rules (filled rectangles), then it would be simple in these bar charts to draw a white rule covering the legend area, erasing anything already there, before drawing the legend text. Without such a primitive, obtaining the same effect is tedious in general, and in some cases, impossible. Since virtually all current TeX output devices, with the exception of older optical phototypesetters, are easily capable of grey-scale rectangle fill, it is regrettable that such a primitive is missing from TeX. It would also supply the need for light background shading of emphasized text. Without it, one must either use grey-scale fonts to fill a rectangular region (a non-trivial operation to program in TeX), or the DVI driver must provide such a feature with the `\special{}` command. It seems evident that support for something like `\special{ruleblackness=0.0}` for white rules, `\special{ruleblackness=0.2}` for lightly-shaded rules, and `\special{ruleblackness=1.0}` for black rules, would be a valuable addition to DVI drivers.

A student named Thomas Taylor produced an X Window System Version 10 Release 4 interface to LaTeX picture mode called TeXdraw; the sources were distributed on the Internet, but I cannot find an institutional affiliation for him. Availability of such a graphical interface can help to relieve the tedium of preparing picture mode graphs by hand.

Other approaches than LaTeX picture mode are possible.

Ehrbar [13] described a set of macros for the production of statistical graphs with plain TeX.

Van Haagen [17] discussed the production of box and scatter plots with TeX macros.

PICTeX [68] seems to be reasonably powerful, but unfortunately has distribution restrictions that will likely prevent its wide adoption.

TeXtyl [54] was developed in support of the Music Typesetting Project under the direction of Professor John Gourlay at Ohio State University [16]. Music is itself an interesting application of both TeX and graphics, but appears to be very difficult, and the OSU project has since been terminated. For further details on computer music software, see [55, Section 3].

In the TUG DVI Standards Committee, we have been experimenting with sets of macros that define graphical primitives for TeX, and I have prepared a <PLOT79> device driver to produce TeX output using those macros. The results are not encouraging, because even with careful coding of the macros by a seasoned TeXpert, practical pictures with the complexity of contour and hidden line plots produce TeX files that are hundreds of kilobytes long with up to fifty thousand line segments. The compact Tektronix vector files are 6 or 7 times smaller than the graphics files using these TeX macros.

Such files can be processed only by greatly enlarged versions of TeX with internal tables increased to over two megabytes; on my Sun workstation, the virtual memory image of such a TeX is over 7 Mbytes. Comparison of the number of line segments with TeX's internal memory sizes suggests that each segment requires about 44 words of memory; later redesign of the macros has reduced that figure, but not sufficiently to permit handling of large graphs.

Given that many TeX users employ personal computer versions of the program, use of enlarged TeX's that cannot run on small machines is a great hindrance to document portability. TeX's internal design is such that a single page is constructed at one time, and there is no provision for preparation of partial pages by implementing software virtual memory swapping. Modified versions of TeX that actually do this exist. The first was apparently also the first port of TeX to a small machine, the HP-3000, by Lance Carnes [9], who later used his skills to produce the first IBM PC port of TeX, and found Personal TeX Inc. The second seems to be Kinch Computer's TurboTeX, a new port of TeX to the IBM PC and other machines; it supports very large internal table sizes, at the expense of software-managed virtual memory swapping. My own experience with it is that on a IBM PC XT without an extended memory RAM disk, it runs many times slower than non-virtual implementations of TeX.

One might wonder whether it is possible to get TEX to empty its memory contents to the DVI file on command, without starting a new page in the DVI file. Examination of the program [39, part 32], and experiment, show that this is not possible. TEX does not call the `\output` routine until either it determines that the collected vertical list of boxes is larger than the current page, or an `\eject` has been issued. The `\output` routine in turn handles the jobs of attaching headers, footers, inserts, and footnotes, boxing and outputting the page with `\shipout`, and advancing the page counter. `\shipout` moves TEX's page image to the DVI file, *followed by an* eop *(end-of-page) command*, and then frees the page memory. This makes it impossible to dump a partial page to the DVI file. `\shipout` can be called at any time on the page to force premature output of the current page image, but in order to make use of this for memory reduction, it would be necessary to add a feature to the DVI driver, presumably through a `\special{}` command, to identify the incomplete page. This approach therefore does not seem attractive.

7 Graphics and TEX Post-Processing

We can delay the handling of graphics until the DVI driver processing stage, after TEX has finished off the DVI file, if we use the `\special{}` command. We have already discussed that in an earlier section.

There is another possibility which has been suggested in the TUG DVI standards exchanges. That is to use font characters numbers, rather than their bitmaps, to encode graphics commands. Thus, instead of TEX macros generating lots of little dots to represent a diagonal line, they could instead output a series of characters in a special font that the DVI driver could interpret as graphics commands. This would reduce the size of the DVI file, since vectors are not expanded into points, and also reduce TEX's memory requirements. By a suitable definition of the font file, it could be made to produce no visible characters if processed by an older DVI driver, and newer ones would be able to act upon the request codes.

8 Color and TEX

SLITEX provides a simple scheme for making color transparencies. It produces multiple pages in the output DVI file which can be printed on colored transparencies that can be overlaid to give a multicolor image. For support of applications that only require a few different colors, this seems a reasonable approach. The macros necessary to permit the user to type something like

```
Celui-l\'{a} est {\blue bleu} et celle-ci est {\red rouge}.
Les autres sont tous noirs.
```

could be extracted from the file **slitex.tex** present in every standard TEX distribution. Nothing is put into the DVI file to identify the color required for each page; that job must be done manually.

The TUG DVI Standards Committee has spent considerable time on the subject of color. Because TEX itself lacks any notion of color, support for that feature must be done by other methods, such as the use of **\special{}** commands.

While one can imagine simple uses of color, such as for shaded backgrounds, for which simple **\special{}** commands might be specified, the subject is really much more complicated.

The reason color is hard is that its specification depends on the characteristics of the output device, on the ambient light when the output is viewed, and on the physiology of the human visual system. The author of a color document may have some control over the first, but none whatever over the second and third.

Let us first consider the output device. The same printing or photographic device on different runs may not produce the same colors, because of changes in the inks or film. As in the case of high-quality professional photography, calibration of batches of film or color toner material may be necessary for consistent results. A particular mix of primary colors, such as red, green and blue for additive color display (e.g. a display screen), or cyan, magenta, and yellow for subtractive color display (e.g. color printing with ink or wax) will not result in the same appearance on different types of output devices, again requiring calibration. For some types of color printing, it is necessary to prepare color separations, in three or four colors. This is not as trivial as one might think, because the lack of purity of the inks or waxes used for the color primaries results in equal blends producing muddy brown, instead of black. To get around this problem, the printing industry uses a technique known as *undercolor removal*, in which part of the contribution of the primaries is replaced by black. This may serve an economic purpose as well, since black ink is usually less expensive than colored inks. When each of the primary colors is overprinted to prepare the final image, paper registration will be slightly different each time, resulting in color overlap that gives spurious colors around edges. This phenomenon is known as a *color trap*. The color resulting from overprinting of inks depends on the transparency of the inks; a completely opaque ink will mask underlying colors, rather than producing a blend. Even if one is producing a grey-level image, it is unlikely that a linear intensity scale will produce a uniform linear intensity pattern on the display; a non-linear adjustment known in the photographic trade as *gamma correction* is necessary. There are several other technical problems in color printing that we will not go into here. Further details can be found in [57, p. 192, p. 302, p. 263], [15, ch. 17], and [19, ch. 10].

The human visual system is not equally sensitive to different colors (light frequencies). The eye is most sensitive to yellow, and least sensitive to blue, and the relative sensitivities may vary between individuals. Various forms of

color blindness also affect color perception, as do other colors surrounding the one being looked at.

What all of this means in practice for computer applications of color printing is that it is impossible to unambiguously specify colors by names like *red*, *yellow*, and *blue*, or even as fractions of three primary colors. Thus, it must be possible to remap the intensities of color primaries after an electronic document has been produced, but prior to the printing process.

For TEX, this means that DVI drivers that support color via `\special{}` commands should probably do so via a single color model, probably red-green-blue, and that they should be able to read mapping tables from startup files. If named colors are to be supported, it must be possible to provide the mapping of color name to color primary values at run time. The X Window System provides this in the form of a standard library file, *rgb.txt*, which contains over 300 entries like

```
112 219 147             aquamarine
50 204 153              medium aquamarine
50 204 153              MediumAquamarine
0 0 0                   black
0 0 255                 blue
...
255 127 0               coral
0 255 255               cyan
142 35 35               firebrick
165 42 42               brown
...
0 0 0                   grey0
3 3 3                   grey1
5 5 5                   grey2
...
250 250 250             grey98
252 252 252             grey99
255 255 255             grey100
```

The first three values on each line are the intensities of red, green, and blue on a scale of 0 to 255. The remaining text is a color name, spelled in lowercase letters with embedded blanks, or with initial capitals with blanks removed. Synonyms are often available; *black*, *grey0*, and *gray0* all map to the triple 0 0 0. X programs accept these color names as values of command-line options; they also accept hexadecimal strings defining the red-green-blue mix. A standard library routine takes care of the data base lookup and parsing.

Recent work at Xerox PARC Laboratories on the specification of color has resulted in the publication of a new color encoding mechanism [11]; an article discussing this will appear in [12]. The Xerox standard contains a good descrip-

tion of several color encoding schemes, and also treats the mapping of color onto monochrome.

9 Halftone Images and TEX

The commonest TEX output devices are capable of displaying only two colors, usually black and white. Grey-level images can be displayed on such devices by either of two techniques—dithering and halftoning [15, ch. 17]. In dithering, rectangular blocks of pixels with varying numbers of black pixels are used to simulate intensities. In halftoning, intensity is simulated by variable dot sizes. Halftoning is the method commonly used in the printing industry.

Significant progress in computer-generated halftoning has occurred in the last few years, with contributions from PostScript developers [53, pp. 131–135] [57, ch. 10], Knuth himself [43, 44], and Ulichney [67]. The work of Knuth and Ulichney happened around the same time without either apparently being aware of the other, even though Ulichney used TEX to typeset his book!

From the point of view of the TEX user, what this means is that it is in principle possible to simulate grey-scale images on most kinds of output devices. The major problem is how to get a grey-scale image into the TEX document.

Knuth's approach [44] is to use METAFONT to create fonts whose characters represent a range of grey levels, then to use these to typeset images represented as scanlines, each pixel of which has a specific intensity, and is in turn represented by a single printable character. The important advantage of this scheme is document portability.

Another approach would be to request the DVI driver, through a `\special{}` command, to incorporate a grey-level image file in some format, using a dithering or halftoning algorithm to convert it to a bi-level image that could be displayed on the page. It would be best to implement this conversion in a separate filter that maps a grey-level image onto a bi-level image, and then to require only of the DVI driver that it be able to input a bi-level image in some (preferably simple) format. Such a format has yet to be specified by the TUG DVI Standards Committee.

The major difficulty here is the profusion of image formats. Almost every vendor has adopted different encoding schemes, just as happened with vector graphics in the 1960's and 1970's. A few vendors espouse a format called Tag Image File Format (TIFF) [10], which is produced by some grey-level scanners now on the market. The specification of this format is long, and I estimate that implementation of a complete TIFF decoder would take between 5 and 15 thousand lines of C code. For comparison, the DVI drivers in my family average 10 to 12 thousand lines of code each.

Some work has already happened on the problem of conversion between various image formats.

Jürgen Wagner <gandalf@csli.stanford.edu> at Stanford University has

produced a package called *bmx*, which is library of C functions that supports conversions of about 20 different bitmap, grey-level, and color image formats, including two types of FAX images.

Jef Poskanzer, <apple!well!pokey@bloom-beacon.mit.edu> and <jef@rtsg.ee.lbl.gov>, at Lawrence Berkeley Laboratory has produced a portable bitmap package, *pbm*, that is included in the X Window System contributed software distribution; it handles only monochrome images.

Michael Mauldin <Michael.Mauldin@nl.cs.cmu.edu> at Carnegie-Mellon University has a 'fuzzy pixmap' package, *fbm*, for conversion between a variety of bitmap, grey-level, and color image formats. His code includes one of the new halftoning algorithms discovered by Ulichney.

The Alpha-1 Computer-Aided Design and Modelling group at the University of Utah Computer Science Department has produced a package called the 'Utah Raster Toolkit', or *urt*, which handles conversions between many different formats, and also includes facilities for compression, scaling, rotation, and reflection of images.

The Internet X-WINDOWS newsgroup has recently carried extensive discussions about a portable image file format, *pdf*, that has been proposed. We may hope that this leads to something useful.

All of these packages are freely available, but are likely to be directly usable only on the UNIX operating system. TEX users on other systems are still largely without support software. Considering this lack, it might prove desirable for a collaborative project to be initiated by interested individuals to provide similar tools written in Web, and perhaps even for TEX User Groups to contribute financial support to such a project

10 Languages for Typesetting Graphics

Before TEX, there was *troff*, a typesetting system developed by Joseph Osanna at AT&T Bell Laboratories about 1976 on the UNIX operating system. This has been further enhanced into device-independent troff, *ditroff*, by Brian W. Kernighan [32, 34]. *troff* is a very low-level formatter, and in the UNIX tradition, it has been enhanced by the addition of separate filters that preprocess higher-level input into *troff* commands. Such filters include *eqn* [3, ch. 9] for mathematics, *tbl* [3, ch. 10] for tables, *pic* [6, 35] for pictures, *grap* [7] for line graphs, and *ideal* [71, 72] for pictures.

Most of these were developed at AT&T Bell Laboratories, and are available in some, but not all, versions of UNIX, and are completely unavailable outside that operating system. *ideal* was produced at Stanford University where its author worked under the direction of Knuth, but was polished and released at AT&T Bell Laboratories.

pic, *grap*, and *ideal* are noteworthy in handling the common requirement of simple sketches of text in boxes or ovals connected with lines with very little

effort from the author. However, they are all much more powerful, and can be used to prepare quite sophisticated diagrams.

So far, little has been done in the TEX world to emulate these. *grap* is actually translated to *pic*, so only *pic* and *ideal* need to worry about translation to typesetter codes. Since TEX can easily do anything that *troff* can, there is no significant impediment to reimplementing *grap*, *pic*, and *ideal* in public-domain Web code that could make these valuable tools available to a wider community.

PICTEX [68] and TEXtyl [54] seem to be the closest in spirit to these UNIX tools, but are only superficially similar. I do not have sufficient personal experience with them to comment on their relative power.

11 Conclusions

In this paper, I have surveyed several approaches to the incorporation of graphics in TEX documents. No single solution emerges as a clear choice.

The most desirable solutions retain device independence; this is possible only when the graphics primitives are implemented entirely with TEX macros, possibly supported by fonts for primitive objects (as in LATEX picture mode) or grey scale.

If the graphics are generated by METAFONT programs, then font characters containing the graphics images are resolution-dependent, but can straightforwardly be regenerated for any output device resolution.

With either of these approaches, no additional support from DVI drivers is needed. This means that it is possible to display DVI files on a screen with a suitable DVI driver, and see the graphics embedded with the typeset text.

Less desirable approaches are to use special fonts to encode graphics primitives, and `\special{}` commands for requesting graphics file inclusion, or output of specified graphical objects. Both of these require changes to DVI drivers, and in the case of graphics file inclusion, require resolution of the messy issues of file format, and image positioning and scaling. DVI driver standardization efforts may assist in resolution of some of these problems, and may define a recommended syntax for the content of the `\special{}` strings, so that different DVI drivers at least could be expected to do the same thing with the request.

I personally find extremely attractive the idea of little languages for graphics typesetting, as discussed in the section *Languages for Typesetting Graphics*. Having new *public-domain* highly portable re-implementations of the existing work in this area would be extremely valuable. In the past, AT&T Bell Laboratories has been willing to release some of their copious software production for free use elsewhere, subject only to their retention of copyright. If someone wishes to start such a project, then it would be a good idea to discuss the possibility of prior source code release with the relevant authors.

It would also be advisable to discuss the project with the Free Software Foundation, in Cambridge, MA, whose goal is to produce a complete UNIX-like

operating system, called GNU, that will be freely available to all; they may already have work in progress on these languages. Interestingly, they use TEX for all their documentation, and when their version of *troff* is completed, it will output DVI files, rather than old style C/A/T phototypesetter files, or *ditroff* output files.

Finally, further work is necessary on the problem of handling complex graphs that exceed TEX's (or METAFONT's) memory limitations. I have intentionally refrained from suggesting changes to either TEX or METAFONT, because I believe strongly that these programs must remain frozen if we are not to interfere with their widespread adoption. It is not clear whether modification of TEX output routines is a viable approach for handling large graphs, but it deserves to be further investigated by people who are intimately familiar with TEX macro programming. The output routines of Plain TEX and LATEX are fragile pieces of code, and changes to them must be made with great care if one is to avoid breaking the output of non-graphics pages.

If, at some time in the future, Donald Knuth wants to implement upward-compatible additions to TEX, then I would suggest that the following be considered:

- a non-uniform rational B-spline curve primitive
- grey-scale rules
- selected graphics primitives from PostScript

In the meantime, they can be added by suitable `\special{}` commands with support from DVI driver code. A properly-designed set of graphics macros could hide the `\special{}` commands entirely, so that later addition of their features in TEX itself could be handled without changes to any user documents. It would also give the chance for considerable experimentation about what a suitable set of graphics primitives for TEX might be, before they become hard-coded in TEX itself.

References

[1] ACM/SIGGRAPH. Status report of the Graphic Standards Planning Committee of ACM/SIGGRAPH. *ACM SIGGRAPH Computer Graphics*, 11(3), 1977.

[2] ACM/SIGGRAPH. Status report of the Graphic Standards Planning Committee of ACM/SIGGRAPH. *ACM SIGGRAPH Computer Graphics*, 13(3), August 1979.

[3] AT&T. *UNIX Programmer's Manual*, volume 2. Holt, Reinhart, and Winston, 1983.

[4] Nelson H. F. Beebe. A user's guide to <PLOT79>. Technical report. University of Utah, 1980.

[5] Nelson H.F. Beebe. A TEX DVI driver family. *TEXniques*, 5:71–114, August 1987. Proceedings of the Eighth Annual Meeting of the TEX Users Group.

[6] Jon Louis Bentley. Programming pearls—little languages. *Communications of the Association for Computing Machinery*, 29(8):711–721, August 1986. Description of the *pic* language.

[7] Jon Louis Bentley and Brian W. Kernighan. Grap—a language for typesetting graphs. *Communications of the Association for Computing Machinery*, 29(8):782–792, August 1986.

[8] Maxine Brown. *Understanding PHIGS*. Template, 1985.

[9] Lance Carnes. TEX for the HP3000. *TUGBoat*, 2(3):25, November 1981.

[10] Aldus Corporation and Microsoft Corporation. Tag image file format (TIFF) specification revision 5.0. Technical report, Aldus Corporation, 411 First Avenue South, Suite 200, Seattle, WA 98104, Tel: (206) 622-5500, and Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717, Tel: (206) 882-8080, August 8 1988.

[11] Xerox Corporation. The Xerox Color Encoding Standard. Technical Report XNSS 288811, Xerox Systems Institute, March 1989.

[12] M. Dvonch, P. Roetling, and R. Buckley. Color descriptors in page description languages. *Proceedings of the Society for Information Display*, 30(2):???–???, June 1989.

[13] Hans Ehrbar. Statistical graphics with TEX. *TUGBoat*, 7(3):171, October 1986.

[14] G. Enderle, K. Kansy, and G. Pfaff. *Computer Graphics Programming. GKS—The Graphics Standard.* Symbolic Computation, Editor: J. Encarnação and P. Hayes. Springer-Verlag, 1984.

[15] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. The Systems Programming Series. Addison-Wesley, 1982.

[16] John S. Gourlay. A language for music printing. *Communications of the Association for Computing Machinery*, 29(3):388–401, 1986.

[17] A. J. Van Haagen. Box plots and scatter plots with TEX macros. *TUGBoat*, 9(2):189–192, August 1988.

[18] Steven Harrington. *Computer Graphics A Programming Approach*. McGraw-Hill, 1983.

[19] Steven Harrington. *Computer Graphics—A Programming Approach.* McGraw-Hill, second edition, 1987.

[20] Allen V. Hershey. Calligraphy for computers. Technical Report TR-2101, U. S. Naval Weapons Laboratory, Dahlgren, VA 22448, August 1967.

[21] Allen V. Hershey. FORTRAN IV programming for cartography and typography. Technical Report TR-2339, U. S. Naval Weapons Laboratory, Dahlgren, VA 22448, September 1969.

[22] Allen V. Hershey. Preparation of reports with the FORTRAN typographic system. Technical Report TN-K/27-70, U. S. Naval Weapons Laboratory, Dahlgren, VA 22448, September 1970.

[23] Allen V. Hershey. A computer system for scientific typography. *Computer Graphics and Image Processing*, 1:373–385, 1972.

[24] Allen V. Hershey. Advanced computer typography. Technical Report NPS012-81-005, U. S. Naval Postgraduate School, Monterey, CA 93940, December 1981.

[25] David A. Holzgang. *Understanding PostScript Programming.* Sybex, 1987.

[26] F. Robert A. Hopgood, Julian R. Gallop, David A. Duce, and Dale C. Sutcliffe. *Introduction to the Graphical Kernel System (GKS).* Academic Press, 1983. A. P. I. C. Studies in Data Processing No. 19.

[27] F. Robert A. Hopgood, Julian R. Gallop, David A. Duce, and Dale C. Sutcliffe. *Introduction to the Graphical Kernel System (GKS).* Academic Press, second edition, 1986. Revised for the International Standard. A. P. I. C. Studies in Data Processing No. 28.

[28] Adobe Systems Incorporated. Colophon—Adobe Systems News Publication.

[29] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, 1985.

[30] Adobe Systems Incorporated. *PostScript Language Tutorial and Cookbook.* Addison-Wesley, 1985.

[31] Oliver Jones. *Introduction to the X Window System.* Prentice-Hall, 1989.

[32] Mark Kahrs and Lee Moore. Adventures with typesetter-independent TROFF. *USENIX Summer Conference Proceedings*, pages 258–269, June 12–15 1984.

[33] Peter Karow. *Digital Formats for Typefaces.* URW Verlag, Hamburg, 1987.

[34] Brian W. Kernighan. A typesetter-independent TROFF. Technical Report Computer Science Report 91, AT&T Bell Laboratories, Murray Hill, New Jersey, 1981.

[35] Brian W. Kernighan. PIC—a language for typesetting graphics. *Software—Practice and Experience*, 12(1):1–22, January 1982.

[36] Donald Knuth and Pierre MacKay. Mixing right-to-left texts with left-to-right texts. *TUGBoat*, 8(1):14, April 1987.

[37] Donald E. Knuth. *TEX and METAFONT—New Directions in Typesetting.* Digital Press, 1979.

[38] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting.* Addison-Wesley, 1986.

[39] Donald E. Knuth. *TEX: The Program*, volume B of *Computers and Typesetting.* Addison-Wesley, 1986.

[40] Donald E. Knuth. *The* METAFONT*book*, volume C of *Computers and Typesetting.* Addison-Wesley, 1986.

[41] Donald E. Knuth. METAFONT*: The Program*, volume D of *Computers and Typesetting.* Addison-Wesley, 1986.

[42] Donald E. Knuth. *Computer Modern Typefaces*, volume E of *Computers and Typesetting.* Addison-Wesley, 1986.

[43] Donald E. Knuth. Digital halftones by dot diffusion. *ACM Transactions on Graphics*, 6(4):245–273, October 1987.

[44] Donald E. Knuth. Fonts for digital halftones. *TUGBoat*, 8(2):135, July 1987.

[45] Donald E. Knuth. The errors of TEX. Technical Report STAN-CS-88-1223, Stanford University Computer Science Department, September 1988.

[46] Leslie Lamport. LATEX—*A Document Preparation System—User's Guide and Reference Manual.* Addison-Wesley, 1985.

[47] David Ness. The use of TEX in a commercial environment. *TEXniques*, 5:115–123, August 1987. Proceedings of the Eighth Annual Meeting of the TEX Users Group.

[48] Adrian Nye. *Xlib Programming Manual for Version 11*, volume 1. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, 1988.

[49] Adrian Nye. *Xlib Reference Manual for Version 11*, volume 2. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, 1988.

[50] Tim O'Reilly, Valerie Quercia, and Linda Lamb. *X Window System User's Guide for Version 11*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, 1988.

[51] IEEE Task P754. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, New York, 1985. A preliminary draft was published in the January 1980 issue of IEEE Computer, with several companion articles.

[52] Lynne A. Price. Hebrew letter (with source). *TUGBoat*, 2(1):122, February 1981.

[53] Glenn C. Reid. *PostScript Language Program Design*. Addison-Wesley, 1988.

[54] John S. Renner. TEXtyl: a line-drawing interface for TEX. Technical Report OSU-CISRC-4/87-TR9, Department of Computer Science, Ohio State University, March 1987.

[55] Curtis Roads and John Strawn, editors. *Foundations of Computer Music*. MIT Press, 1987.

[56] Randi J. Rost. PEX introduction and overview. Technical Report Version 3.20, Digital Equipment Corporation, Workstation Systems Engineering, April 1988. This document is present in the X Window System Version 11 Release 3 in the file `X11/X11/doc/extensions/pex/doc/intro/doc.ms`.

[57] Stephen E. Roth, editor. *Real World PostScript*. Addison-Wesley, 1988.

[58] Yasuki Saito. Report on JTEX: A Japanese TEX. *TUGBoat*, 8(2):103, July 1987.

[59] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System C Library and Protocol Reference*. Digital Press, 1988.

[60] G. K. M. Tobin. A bit of doggerel. *TUGBoat*, 6(1):12, March 1985.

[61] Georgia K. M. Tobin. Computer calligraphy. *TUGBoat*, 4(1):26, April 1983.

[62] Georgia K. M. Tobin. The OCLC Roman family of fonts. *TUGBoat*, 5(1):36, May 1984.

[63] Georgia K. M. Tobin. Some empirical observations on METAFONT design. *TUGBoat*, 8(1):26, April 1987.

[64] Georgia K. M. Tobin. Designing for low-res devices. *TUGBoat*, 9(2):126–128, August 1988.

[65] Georgia K. M. Tobin. The ABC's of special effects. *TUGBoat*, 9(1):15–18, April 1988.

[66] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 1983.

[67] Robert Ulichney. *Digital Halftoning*. MIT Press, 1987.

[68] Michael J. Wichura. PICTEX: Macros for drawing PICtures. *TUGBoat*, 9(2):193–197, August 1988.

[69] Norman M. Wolcott and Joseph Hilsenrath. A contribution to computer typesetting techniques. Tables of coordinates for Hershey's repertoire of occidental type fonts and graphics symbols. Technical Report PB-251 845, U. S. National Bureau of Standards, April 1976. NBS Special Publication 424.

[70] Patrick Wood, editor. PostScript Language Journal. Pipeline Associates, Inc., P.O. Box 5763, Parsippany, NJ 07054.

[71] Christopher J. Van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[72] Christopher J. Van Wyk. AWK as glue for programs. *Software—Practice and Experience*, 16(4):369–388, April 1986.

[73] ANSI Subcommittee X3H3. *Information Systems—Computer Graphics—Graphical Kernel System (GKS). ANSI X3.124-1985*. American National Standards Institute, 1430 Broadway, New York, N. Y., 10018, 1985. Includes Fortran bindings to GKS.

[74] ANSI Subcommittee X3H3. *PHIGS+ Functional Description, Revision 2.0*. American National Standards Institute, 1430 Broadway, New York, N. Y., 10018, July 20 1987.

[75] ANSI Subcommittee X3H3. *Information Systems– Computer Graphics—Programmer's Hierarchical Interactive Graphical System. Draft proposal X3.144.1988*. American National Standards Institute, 1430 Broadway, New York, N. Y., 10018, 1988.