TkDVI: A Tcl/Tk-based TEX DVI Previewer

Anselm Lingnau
Fachbereich Informatik (ABVS)
Johann Wolfgang Goethe-Universität Frankfurt, Germany
E-Mail: lingnau@tm.informatik.uni-frankfurt.de

Abstract

Application-level scripting is a powerful method for structuring software. This paper introduces TkDVI, a TEX DVI previewer based on the Tcl/Tk scripting language and graphics toolkit. After a brief introduction to Tcl/Tk, we present the design and major components of the previewer, pointing out the specific advantages gained by using Tcl/Tk. A number of extensions and future projects is also discussed.

1 Introduction

In spite of the recent predominance of direct-manipulation word processing tools, markup-based document production systems like TEX [4] and LATEX [7] are still going strong. In fact, support for international fonts, modern output formats like Adobe PDF [8] and convenient freely-available solutions for many different typesetting problems have made LATEX in particular a strong contender in today's typesetting world.

One area where the TEX community lags behind what is possible on modern systems concerns previewing—the ability to display the typeset output on a bitmapped screen in order to proofread the content or check the formatting. For the most part, previewers for TEX's DVI (for *device-independent*) file format have contented themselves with showing individual pages from a DVI file, possibly displaying PostScript illustrations as well by means of external rendering engines, but, e. g, navigation facilities have been rudimentary at best. Even fairly obvious extensions like the ability to show "spreads" of adjacent pages in a book are far less widespread than they ought to be. The main reason for this seems to be that DVI previewers, while not complicated programs *per se*, are often (as an examination of some of the freely available ones will illustrate) fairly inscrutable and difficult to adapt and extend by people other than the original authors. This may be partly due to the fact that the programs are generally written in systems programming languages like C, for speed and portability, and that, while these goals are usually achieved, flexibility and extensibility at the user interface level often suffer in the process. (This is a problem that is by no means particular to the area of DVI display software.)

For a Tcl/Tk hacker, the obvious approach to this problem is to try and come up with an implementation of DVI previewing for use in Tcl/Tk programs. There are various ways in which a DVI previewer can benefit from this: If the DVI display functions are suitably encapsulated and made available at the scripting language level, it is easy to use them as "building blocks" for more sophisticated things such as the two-page spread display mentioned above. Since many other, unrelated building blocks are available to Tcl—for example, access to the World-Wide Web via an HTTP module—the previewer can be extended in various interesting ways with relatively little effort. Finally, since TEX includes a provision for putting arbitrary material in a DVI file, the TEX output itself can be given access to Tcl to support features such as primitives for simple graphics, inclusion of graphics files, or even interactive extensions such as hyperlinks, sound and animation.

This paper introduces TkDVI, a DVI previewer based on Tcl/Tk. After a brief introduction to TEX output and its interpretation, we present the design and implementation of the major components of TkDVI, pointing out the specific advantages gained by using Tcl and Tk. The paper closes with an exposition of interesting future projects and ideas.

2 DVI Files: A Brief Overview

DVI (short for *device-independent*—and not to be confused with Intel's digital video file format that uses the same abbreviation) is the canonical output format of TEX. It describes a typeset document by giving instructions to a "virtual machine" for every page. The virtual machine maintains a notion of "current position" on the page, and the instructions mostly consist of commands that change this position or place glyphs or rules there. Besides registers for the current position, the virtual machine implements a few more registers for optimization purposes; the content of all registers can be pushed on or popped off a stack to make regular movement easier. There are a few additional commands that deal with font management and the inclusion of arbitrary strings in the DVI file for the benefit of specialized DVI processing software. DVI file also contain a *preamble* and a *postamble* giving information about the file as a whole, like a global magnification factor or the maximum stack depth necessary for the virtual machine to process the file. A full explanation of the DVI format is given in [5].

It is important to note that DVI files do not contain any information about the glyphs of a particular font—they just contain the font name and the scale that the font is to be used "at". (When TEX was invented, scalable fonts were not yet widely available or used, so from the point of view of TEX, Times Roman at 10pt and Times Roman at 10pt scaled by a factor of 1.2 are two different fonts.) The actual bit maps giving the glyph shapes of the characters need to come from elsewhere, and that is totally up to the DVI rendering program. Traditionally, a companion program to TEX called METAFONT would be used to generate bitmapped fonts from algebraic descriptions of the glyph shapes, and it is customary for DVI processors to be able to read these bitmapped fonts, which are typically in a compressed format called PK [11]. Of course, DVI processing programs are free to use whatever source of fonts is convenient, and there are various DVI renderers that use PostScript Type 1 fonts or even TrueType fonts instead of or in addition to PK-format bitmapped fonts. (There are tools that will generate appropriate PK fonts from Type 1 or TrueType fonts for the use of renderers that cannot access the scalable formats directly.) A discussion of the merits of these different approaches is outside the scope of this paper.

The TEX program itself, when typesetting, concerns itself only with the *metrics* of the glyphs in a font rather than their appearance. It considers every glyph as a box of given dimensions, and these boxes are put together with "glue" (varying amounts of white space) to form lines, paragraphs and pages. It is important to note that the box for a given glyph is usually roughly coincident with a "bounding box" for the glyph in the sense that all the pixels of the glyph are contained within its box, but this is not necessarily the case; for example, an italic "*f*" often exceeds the dimensions specified for its box as visible to TEX. TEX obtains the metrics it uses for typesetting from special "TFM" (or *TEX font metric*) files. In addition to character dimensions, TFM files contain ligature tables (for transformation of certain character pairs such as "f"-"f" into special single-character glyphs) and kerning tables (which optimize the spacing of pairs of adjacent characters such as "AV" or "OT").

A fairly new addition to TEX are "virtual fonts", which make it possible to re-encode other fonts or generate new glyphs on the fly by pulling glyphs from other fonts (possibly with scaling) and combining them into new characters. Glyphs in virtual fonts are represented by short snippets of "DVI machine code" largely similar to that found in DVI files proper. To TEX, virtual fonts are presented via ordinary TFM files, but DVI renderers must be able to process the actual character definitions, which are contained in "VF" files. Virtual fonts are essential for processing PostScript Type 1 fonts in LATEX, among other things.

The basic unit of measurement in TEX is the "scaled point", which is 1/65536 of a printer's point, of which there are 72.27 to the inch (PostScript uses what TEX calls "big points"—72 to the inch). So one scaled point, at 5.3 nm (less than the wavelength of visible light) is a much smaller dimension than current printing technology can handle. One consequence of this that DVI rendering programs need to deal with is that the positions given in a DVI file must necessarily be rounded to the much coarser resolutions that available technology gives us (such as 600 DPI for popular low-cost laser printers). Unfortunately, while the human eye cannot discern a simple difference of a few pixels at 600 DPI, shifted glyphs within a word due to rounding errors are quite obvious (and undesirable). The TEX approach to this problem is to maintain the "pixel" position on the page independently

from the "official" position measured in scaled points. While a single word is typeset (defined as a sequence of characters separated by "small" spaces according to a notion of "small" dependent on the current font size), glyphs are positioned according to their widths measured in output device pixels rather than their widths measured in scaled points, and any rounding is postponed until the next inter-word space, where a correction is, again, invisible. There is a canonical algorithm for this which is explained in [6].

Another important observation is that DVI files contain no provision for graphics other than oblong rules, which are generally used for horizontal or vertical lines. There are no primitives for lines of arbitrary slope, curves, circles, filled shapes or anything—a restriction that probably stems from the capabilities of the phototypesetters available to Knuth when TEX was developed. However, there is a "back door": TEX contains a `\special{`*argument*`}` command whose *argument* is written literally to the DVI file. Many DVI renderers have used this to implement either additional graphic primitives or inclusion of graphic files such as GIF or PostScript. In the meantime, there have been various proposals for standards for extending the graphics capabilities of TEX DVI files, but no consensus has been reached [9]; however, attempts to introduce a certain amount of abstraction at the document level, e. g., for LATEX, have been moderately successful [3].

Having concluded this whirlwind tour of the technicalities of TEX output, we can now summarize the requirements on a DVI rendering program such as a previewer:

DVI file format The program must be able to open a DVI file, obtain any pertinent information describing the file as a whole, to locate a given page in the file and to "execute" the virtual machine code for that page, keeping track of the proper values of all the virtual machine's registers etc.

Fonts The program must be able to locate the PK (or whatever) files giving the bit maps for the glyphs used in the document. In the absence of a suitable file, the font's TFM file should be used to be able to introduce the proper amount of white space that would otherwise be occupied by the unavailable characters. If possible, a generic font can be used to show at least the characters that are supposed to show up, if not the correct glyphs. Virtual fonts should be supported.

Rounding The program must import the canonical rounding algorithm for pixel positions.

Graphics It would be nice if the program could deal with the most popular styles of specifying graphics, such as PostScript inclusion. (Depending on the availability of a PostScript engine such as Ghostscript, this may be quite a difficult job.)

3 The Architecture of TkDVI

The main goal behind the TkDVI project is to provide not only a state-of-the-art TEX previewer, but also a flexible platform for experiments with DVI previewing in general. The DVI-handling functionality should be packaged in a manner that would make it usable not only by the TkDVI program, but by other Tcl/Tk programs that want to handle DVI files (one might imagine a World-Wide Web browser offering transparent support for both HTML and DVI documents). It would be even better if the basic functionality could be made available in a library that would not rely on direct help from Tcl, in order to be usable with other programming languages as well. For example, Tk is popular as a graphics toolkit for other languages such as Perl, Python or Scheme, and the design of the DVI-handling code in TkDVI should not preclude its use with Tk in a non-Tcl environment.

The main functional units within TkDVI (see figure 1) consist of a DVI file handler, a DVI code handler, a DVI code interpreter, the font subsystem and a renderer which is specific to the targeted graphics system. The DVI file handler is responsible for loading DVI files and making their content accessible to the DVI code handler. The DVI code handler, on the other hand, locates individual pages within a glob of DVI code (usually, but not restricted to, the content of a DVI file) and exports an interface for finding particular pages by page number. This in turn is used by the DVI code interpreter, which parses and executes DVI code and calls user-specified routines to deal with positioning glyphs and rules on a page and executing `\special` commands. The DVI code interpreter calls the

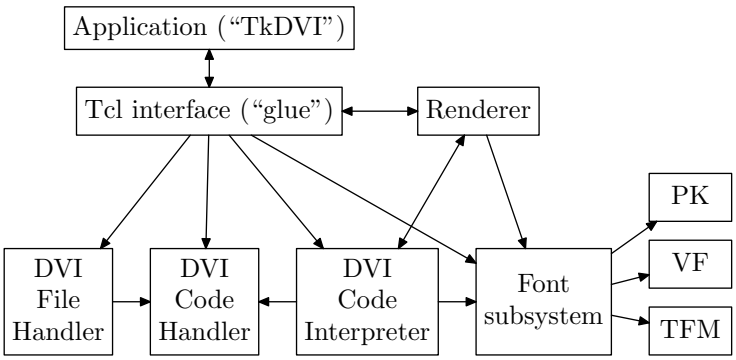

Figure 1: Main functional units within TkDVI. The arrows denote the direction of procedure calls.

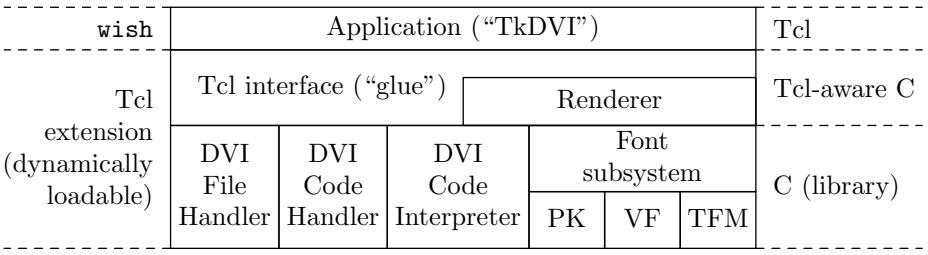

Figure 2: Implementation structure of TkDVI

font subsystem to obtain the metric information for rendering glyphs; the font subsystem also supplies glyph bitmaps and metrics to the rendering system. All these subsystems—file handler, code handler and code interpreter—are designed to support multiple instances of files, code and interpreters. This makes it easy to display multiple pages from multiple DVI files in multiple windows. The font subsystem maintains a pool of loaded fonts which can be shared between DVI interpreters, thus a font that is used in several different DVI files needs to be loaded in memory only once.

The rendering subsystem's task is to display glyphs and rules at appropriate positions on the page, and to interpret `\special` commands found in the DVI file. All the knowledge of the output device is encapsulated here. TkDVI's renderer is, of course, designed with Tk and X11 in mind, but it would be straightforward to re-target it to a different graphics system or use the underlying functionality as the basis for a printer driver.

For most of the DVI-handling subsystems there is "glue code" to make their functionality available to Tcl programs. This is necessary to connect TkDVI's user interface—which is written in Tcl—to the actual DVI-handling code, but another important aspect of this is to be able to exercise the various parts together and in isolation for debugging. Tcl contains support for regression testing, and one of the goals of TkDVI is to allow extensive automatic testing of the DVI-handling parts. Therefore there are various Tcl commands and subcommands that allow examination and consistency checks of the internal C language datastructures. TkDVI comes with a regression test suite which is unfortunately not as complete as it could be, but is a great boon even in its current state.

4 The Implementation of TkDVI

4.1 Overview

From the implementation point of view, TkDVI can be divided into two parts: the user interface (in Tcl) and the underlying DVI machinery (in the C language). The latter can be further subdivided into the actual DVI-handling code and the Tcl-specific interface code (Fig. 2). The idea is to separate the Tcl-specific code from the general DVI-handling code in order to make it easier for interested parties to provide bindings to other scripting languages such as Perl or Python (the author, sadly, lacks the

time to perform his own experiments but would be happy to hear from anybody attempting this).

We will examine the various subsystems outlined in the previous section one after the other, and point out some of the peculiarities and implementation decisions underlying each.

4.2 The DVI File and Code Handlers

TkDVI draws a distinction between *DVI files* and *DVI interpreters*. In particular, more than one DVI interpreter can work on the same DVI file, which is necessary to be able to display the same file several times without having to load it into memory more than once. TkDVI makes a further distinction between *DVI files* and *DVI code*, where a DVI file is viewed mostly as an unstructured chunk of bytes, and "DVI code" is an abstraction that knows about individual pages, the parameters used to set up appropriate scaling for the metrics and so on. This separation is not strictly necessary for the operation of TkDVI, but it takes into account the notion that DVI code does not necessarily come from a DVI file. For example, it would be possible to use TkDVI as part of a near-WYSIWYG TEX implementation where a TEX compiler and previewer are more tightly coupled as usual, or to use TkDVI as a back-end for a simple interactive word processor based on TEX fonts. At the moment there is no API support for this but it may be introduced in a later version as the need arises.

The most important aspect of the DVI file handler is a function which is responsible for loading a file by name. Since it is possible to access the same file several times, the function checks whether the file is already open and, if this is the case, returns a new reference to it without opening it a second time. For efficiency, on systems that support the `mmap()` system call, the DVI file is mapped directly into the TkDVI process's virtual address space. It can then be treated as a big random-access array of bytes, which is much more convenient than reading it byte-wise and seeking back and forth on the file. TkDVI also keeps track of when a file was loaded, in order to be able to find out if the file has been updated (e. g., by a TEX run) and to reload the file contents if necessary. There is a callback mechanism to notify users of the file in this case—for example, if pages from the file are currently being displayed they need to be re-rendered. There is also (among other things) a function that "closes" a DVI file; the file is only removed from memory when it has been closed as many times as it was loaded before.

Activities beyond making the DVI file contents accessible are left to the DVI code handler. The DVI code handler is usually invoked after a DVI file has been opened. It locates individual pages in the file and pre-scans the file for font definitions and `\special` commands (see below). It is important to note that the code is scanned from the front, so the mechanism can deal with DVI files that have not yet been completely written out. (The current Web2C TEX implementation contains a largely-undocumented facility for using a socket to pass DVI output to another program [2]; since it is simple to write socket servers in Tcl this will be investigated further in due course.) During the pre-scan, the DVI code handler constructs a "page table" which allows random access to every page according to its absolute position in the sequence of pages that makes up the DVI file. It also translates from TEX page numbers as found in the DVI file to absolute page numbers; this is important since TEX page numbers are not necessarily unique in a file.

4.3 The DVI Interpreter

A DVI interpreter operates on pages of DVI code as furnished by the DVI code handler. It is usually called from the renderer (see below) whenever a DVI page must be displayed, for example because the content of a window needs to be refreshed or updated. The DVI interpreter serves as a "virtual machine" that executes DVI operation codes for movement, placing of glyphs and rules and so on. Its main task is to keep track of the current position on the page both in terms of DVI dimensions (usually TEX "scaled points") and in terms of pixel positions of the output device; the more difficult job of actually rendering glyphs and rules is delegated back to the renderer by means of callback functions. Font metrics are obtained from TFM files as required, and the canonical algorithm is used to avoid ugly pixel position mismatches in sequences of glyphs. Virtual fonts, which are essentially DVI code "macros", are implemented by recursive calls to the DVI interpreter from within itself. This means that the maximum stack depth as specified in the DVI file may not be sufficient to actually

render the document; for this reason the size of the actual stack is increased by "a bit", and if this isn't enough it can also be increased further as the need arises, while the interpreter is running.

4.4 The Font Subsystem

Both the DVI interpreter and the renderer need access to font information. While the DVI interpreter uses the width of glyphs for position updates, the renderer must be able to construct the actual glyphs in order to display them on the page. The TkDVI font subsystem maintains a "font repository" which is capable of holding different types of fonts at different resolution and making metrics and glyph information available to other parts of the code. Currently, TkDVI supports PK fonts and virtual fonts directly; it can read TEX font metric (TFM) files to obtain correct positioning if a needed font is not available. The font subsystem is designed to be extensible, so it would not be difficult in principle to support other font formats such as GF (METAFONT's immediate output format), PostScript Type 1 or TrueType (possibly via the *FreeType* library for which there is Tcl support); the only condition is that there must be some means of converting the external font representation to a bitmap, which can then be passed to the renderer. As with DVI files, the same font can be used simultaneously by many DVI interpreters working from a single copy in memory. The font subsystem uses the standard "Kpathsea" library [1] to locate font files in the file system according to a specification of the output device's resolution and METAFONT mode; Kpathsea also converts Type 1 or TrueType fonts to PK fonts as needed, so they can be used indirectly by TkDVI.

4.5 The Renderer

DVI support for Tk can be implemented in two different ways. The more obvious method uses a "DVI widget", which is similar to other Tk widgets such as the `button`, `text` or `canvas` widgets in that it occupies an X11 window of its own as a first-class member of a Tk interface. (This approach was used in a former prototype of TkDVI.) Another possibility is to implement DVI support as a Tk *image type* as discussed above. In this method, DVI pages do not appear as widgets but can be included in all Tk widgets where a Tk `image` is allowed, such as the `button`, `label`, or, most notably, the `canvas` widget. The main advantage of this approach is that several DVI pages can share the same `canvas`, and can also be mixed with the more usual canvas features such as structured graphics items, other images or even full-blown Tk widgets. For example, it is possible to embed forms or interactive graphs in a DVI page.

The TkDVI renderer is based on boilerplate code from the Tk distribution and implements both a set of data structures and callback functions required by Tk for the management of an image type and another set of callback functions for use by the underlying DVI interpreter. Each DVI image "master" has an associated DVI interpreter and a "current page" which it displays, possibly in multiple different windows at the same time. The current page number can be queried and changed from Tcl to allow paging back and forth in the DVI code. The TkDVI renderer works in a manner similar to the popular `xdvi` previewer; it uses comparatively high-resolution fonts—normally fonts for a 600-dpi laser printer—internally but scales the glyphs down for display by an arbitrary integer scaling factor. Anti-aliasing is employed to improve the quality of the scaled glyphs.

The main problem with TkDVI's images is that they are opaque; their background is filled with the background color specified for the widget. It would be nice if one could set a "transparent" background such that only the character glyphs would be painted, similar to the other canvas graphics items such as rectangles which support `-background {}`, but this is difficult due to the way glyphs are put on the page. Right now, in order to anti-alias the glyphs, the renderer uses X images where the color for each pixel is determined by interpolating between the background and foreground colors of the image. These images are then transferred to the page. In order to support "transparent" display; the glyph-to-X-image mechanism would need to be rethought.

4.6 The User Interface

TkDVI's user interface is patterned loosely on the `gv` GhostScript front-end by Johannes Plass [10]. For example, it sports a "page selector" that displays a document's page numbers and lets the user

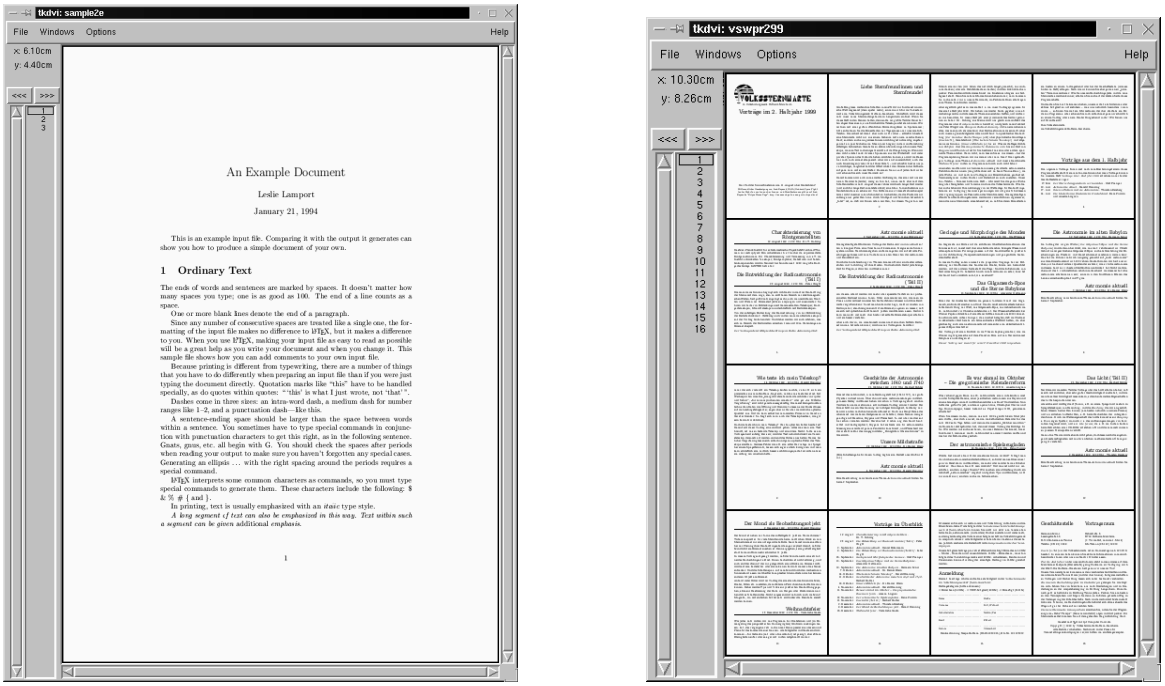

Figure 3: The TkDVI User Interface: Single-page mode (left), Overview mode (right)

jump to any page by clicking on its number with the mouse. It is also possible to select individual pages for printing using `dvips`. A DVI image is displayed in a big window and can be scrolled either by means of scroll bars or by directly dragging the window contents. A mouse button pops up a "magnifier" similar to that in `xdvi`, so part of the page can be examined more closely; various sizes of magnifier are accessible by holding the Shift or Control keys when the button is pressed. It is interesting to note that the enlarged image is implemented by opening a new DVI image in another window displaying the same page of the same file, but without shrinking. This window is designated an "override-redirect" window—it remains under TkDVI's control rather than being taken over by the X11 window manager—and tracks the current position on the shrunken page just as in `xdvi`. Even though all the event handling and window positioning is passed through Tcl, this is not visibly slower than the corresponding feature in `xdvi`. There are buttons to page back and forth, and the DVI window can be scrolled and paged using key shortcuts as well.

Another unusual feature of TkDVI's user interface is the display of two-page spreads and "overviews" showing an array of tiny page images. The former uses a double-width window and allows the viewer to judge the appearence of two facing pages as in a book, while the paging mechanism adjusts to this by moving two pages at a time rather than one. The latter gives a high-level overview of a document that makes it easy to survey the placement of floating objects, while allowing the user to move easily to any displayed page simply by clicking on it. All of this is made possible through the judicious distribution of Tk DVI images on a canvas.

The possibility of superimposing Tk structured graphics on a DVI image is exemplified by the "measuring device". A position on the page can be fixed by a mouse click; subsequent dragging of the mouse shows a "rubber line" from the current position to the original point, while the distance from the current position to the original is displayed in a choice of units of measurement. This is easily implemented at the Tcl level using appropriate button press, release and motion bindings.

The author is not entirely happy with the way the interface has turned out; fortunately Tcl makes it easy to experiment with alternative styles, and it is expected that TkDVI's user interface will gravitate towards one more like that of Adobe Acrobat Reader (while trying to avoid some of that program's drawbacks).

The ease with which the program's user interface can be enhanced through Tcl cannot be overstated. For instance, after a presentation on LaTeX-based hypertext features at EuroTeX'99, including the PDF feature of "bookmarks" that make it easy to jump to the beginning of chapters or sections in

the document, an experimental implementation of DVI-based bookmarks for TkDVI could be completed during the conference, in the space of about two hours. The amount of work this would have taken for a C-based previewer like `xdvi` is unpleasant to contemplate, but Tcl lends itself well to casual user-interface hacks such as this.

5 `\special` Commands

An enticing feature of a Tcl-based previewer is more flexible handling of TEX `\special` commands, which can be used to communicate directly with a DVI driver from a TEX manuscript. In TkDVI, the execution of `\special`s is in the domain of the renderer, which must supply a callback function to the DVI interpreter for the purpose. TkDVI's renderer understands a number of `\special` commands directly and calls a Tcl procedure from the user interface to handle the rest. This makes it possible to include arbitrary Tcl code in `\special` commands.

The most obvious application for this is to push out graphics inclusion to the Tcl side of the program, where it can be performed using canvas primitives, Jan Nijtmans' `Img` extension, or external programs such as Ghostscript, but it also opens the way to DVI files containing forms, interactive elements or World-Wide Web links. Of course it must be considered that device-specific extensions such as these go against the notion of a "device-independent" file format, but they can nevertheless be useful in some circumstances. In particular, inclusion of PostScript graphics would be a very desirable feature, if only for compatibility to existing programs like `xdvi`. It turns out that the facilities of the `Img` extension are difficult to use directly for this purpose: `Img` renders PostScript by invoking an external Ghostscript process for each image which generates a "portable pixmap" (PPM) file corresponding to the PostScript input. This PPM file is then read by Tk using the existing method for the format. To support PostScript graphics at a level comparable to `xdvi`, it should be possible to honor the scaling and rotation specifications given for an embedded PostScript file in the DVI `\special`, but this is partly outside the facilities of `Img`. Besides, for a document containing lots of embedded pictures, forking an external PostScript interpreter for every single image would be very inefficient. The `xdvi` previewer renders PostScript by sending commands to a single instance of Ghostscript (or another Postscript engine) which then outputs directly to `xdvi`'s X11 window. While this approach is not without its problems even in `xdvi`—for example, if one switches to another page while the external Ghostscript process is busy rendering an image, chances are that one will end up with half the picture on the new page—it would be even more difficult to support for TkDVI, due to the double-buffering nature of the Tk canvas and the fact that many pages may be displayed simultaneously. Possibly the most promising approach to this problem would be to adopt an intermediate solution where a single Ghostscript process would be employed to render pictures directly into off-screen pixmaps, which could then be displayed using the standard Tk image mechanisms. Such a method could be packaged as a Tk image type external to TkDVI and would probably be useful to other programs as well.

Another straightforward application of the `\special` mechanism are "source specials", which correlate locations in the DVI file with input file names and lines and make it possible to jump from a position in the DVI window to the approximate place in the corresponding TEX manuscript. For this, the previewer must know how to communicate the input file name and line number from the `\special` to the editor, and Tcl makes this easy to do and configure.

An important issue in connection with arbitrary Tcl statements in DVI files concerns security: what if a DVI file contains evil Tcl code that will format a viewer's hard disk? The answer to this is to arrange for Tcl from DVI files to be executed in a safe interpreter from which potentially dangerous facilities have been removed. It is still possible to arrange for limited access to Tk for this code, so the Tcl code in a `\special` could, for example, display graphics within a specifically designated area on the page. The issues involved in this are similar to the issues surrounding programs such as the Tcl/Tk WWW browser plugin.

6 Tcl-ish issues

Nowadays, with various "flavors" of Tcl/Tk in use ranging from vintage $7.x/4.x$ to the most up-to-date, it is often difficult for extension writers to decide which version of Tcl/Tk to target. TkDVI requires at least Tcl/Tk 8.0 because it uses, as far as possible, the new "objectified" interface for the glue code between Tcl and the internal DVI support procedures. Tk is not yet fully "objectified", and while a `Tcl_Obj` interface to the image subsystem has been released in Tk 8.3, at the time of writing this paper the image code in the TkDVI renderer is still written to the old interface. TkDVI has been tested with versions of Tcl/Tk ranging from 8.0 to 8.2, and no version-related problems could be detected.

Another interesting issue is the management of extensions. Notionally, TkDVI consists of five different modules. While all of these could be compiled and linked into individually loadable extensions, their interconnectedness makes this a bit silly. Right now all of TkDVI's C code ends up in a single loadable extension, which is the simplest approach; however, it would be desirable to split off the Tk-related parts so that DVI tasks not related to rendering could be handled in a non-GUI program. This would make TkDVI's DVI-interpreting code more useful in a wider range of applications. The main obstacle to this is a general difficulty in setting up appropriate mechanisms for compilation: Earlier versions of TkDVI used the `libtool` machinery to automate building sharable libraries on various platforms, and this did not allow libraries that depended on other libraries being built, as would have been the case if the Tk and Tcl portions of the previewer had been separate. The current version no longer uses `libtool` (the abolition of which simplified the build environment to an amazing degree), but the separation has not yet been reinstated. The author follows with great interest the development of the TEA framework for compiled extensions to Tcl, and hopes to be able to make the C portions of TkDVI TEA-compliant as soon as possible.

Finally, a recently-introduced feature of Tcl, *stub libraries*, make it easier to decouple compiled extensions from the particular versions of Tcl they have been compiled for. No work has yet been done to utilize this for TkDVI, but the author is fairly confident that this will be taken care of during the TEAification of the package.

TkDVI currently consists of approximately 8400 lines of C code (including comments) and about 1100 lines of Tcl. Of the Tcl code, approximately 750 lines deal with the "browser", the main TkDVI user interface, and most of the rest (150 lines) is the implementation of the `gv`-style page selection widget.

7 Future Plans and Possibilities

While TkDVI is already quite usable as a program, work on it is by no means finished. The most glaring omission to date (March 2000) is the lack of a mechanism for the inclusion of PostScript graphics.

At the C language level, an interesting area to look at deals with color. The requirements for color management are set out succinctly in [12], and work is ongoing to re-cast and extend TkDVI's `\special` mechanism in the light of that paper. Basic color support has been implemented on a prototypical basis, but limitations in the anti-aliasing procedure make it difficult to support cases which are easily handled in a program such as `dvips`.

The user interface offers many fascinating possibilities, mainly through the flexibility of Tcl/Tk. Some ideas that have been thrown around include an annotation mechanism for DVI files where a viewer could mark up and comment the DVI material in different colors. The marks and comments would then be written to another file which could be sent back to the original author. In connection with "source specials", this would make for a very powerful copyediting environment.

Color and a user interface modified for full-screen display would make TkDVI into a powerful presentation tool, especially for mathematical material, which is notoriously mishandled by other such programs. This could be augmented with animations, multimedia etc. through `\special` commands. Since Tcl/Tk can handle multiple X11 displays at the same time, it would also be possible to "project" such a presentation onto a number of screens in a classroom, controlled from the teacher's

workstation.

Many of these applications can be supported by conventional DVI previewers only through massive development work at the systems programming level. A scripting language such as Tcl brings such experiments within the compass of people who do not usually fancy hacking the innards of a DVI previewer. The author would be pleased to hear from people who have used TkDVI in interesting and unforeseen ways.

`http://www.tm.informatik.uni-frankfurt.de/~lingnau/tkdvi/`

References

[1] Karl Berry. Kpathsea library. GNU Info file; see, for example, `ftp://ftp.dante.de/tex-archive/systems/unix/`, October 1997.

[2] Karl Berry. Web2c. GNU Info file; see, for example, `ftp://ftp.dante.de/tex-archive/systems/unix/`, December 1997.

[3] Michel Goossens, Sebastian Rahtz, and Frank Mittelbach. *The LaTeX Graphics Companion: Illustrating Documents with TeX and PostScript*. Addison-Wesley, Reading, MA, 1997.

[4] Donald E. Knuth. *The TeXbook*. Addison-Wesley, Reading, MA, 1984.

[5] Donald E. Knuth. *TeX: The Program*. Addison-Wesley, Reading, MA, 1986.

[6] Donald E. Knuth and David R. Fuchs. TeXware. Stanford Computer Science Report 1097, Stanford University, April 1986. See in particular §84 and §85 of "DVItype", which is available, e. g., from `ftp://ftp.dante.de/tex-archive/systems/knuth/texware/.`

[7] Leslie Lamport. *LaTeX, A Document Preparation System*. Addison-Wesley, Reading, MA, 1986.

[8] Thomas Merz. *Postscript & Acrobat/PDF: Applications, Troubleshooting and Cross-Platform Publishing*. Springer, 1997.

[9] NTG TeX future working group. TeX in 2003: Proposal for a `\special` standard. In *Proc. of the 19th Annual TeX Users Group Meeting*, pages 181–188, August 1998.

[10] Johannes Plass. gv (PostScript viewer). See `http://wwwthep.physik.uni-mainz.de/~plass/gv/`.

[11] Tomas Rokicki. Packed (PK) font file format. *TUGboat*, 6(3):115–120, November 1985.

[12] Tomas Gerhard Rokicki. Driver support for color in TeX: Proposal and implementation. *TUGboat*, 15(3):205–212, September 1994.