TkDVI: DVI Previewing with Tcl and Tk

Anselm Lingnau

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.

Introduction

The history of DVI previewing is a history of the adequate rather than the excellent. Most DVI previewers are fairly good at what they do—displaying pages from DVI files—but there are lots of things a DVI previewer could do that have seldom if ever been attempted. Sure enough, some Programs support World-Wide-Web-style hyperlinks and even interface to the Web itself, and some programs can use external PostScript interpreters to display illustrations, but 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.)

An alternative approach to the construction of interactive programs (such as DVI previewers) takes into account that the requirements of speed on one hand and flexibility and extensibility on the other hand are difficult to attain when a program is written in a single language. This leads to the the notion of *scripting*, which in a general sense has a long and glorious history in the UNIX system but has been pioneered at the application level by John Ousterhout, who designed the Tcl language and the Tk toolkit [6]. The idea behind Tcl (short for "Tool command language") is that those parts of an application that need to be fast are written in a *systems programming language* such as C, while the parts that are supposed to be flexible and extensible, such as the user interface, are built on top of the resulting “building blocks” in a *scripting language* such as Tcl. In this combination, the relative slowness of languages like Tcl does not matter because all the heavy cycles-eating work is done in a compile-to-machine-code language like C, and since the complicated aspects of the systems programming language are hidden behind the abstractions visible in the scripting language, it is easy to change and adapt the user interface at the script level.

There are various ways in which a DVI previewer can benefit from this approach. 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 the scripting language—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, the DVI material itself can be given access to the scripting language (for example via the `\special` mechanism of TEX) 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 Tcl/Tk, 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.

Tcl/Tk: A Scripting Language and GUI Toolkit

The Tcl language was originally conceived as a *lingua franca* for interactive programs. Many interactive programs feature their own ad-hoc command languages, but many of these command languages are complicated, limited in expressive power and buggy. Also, their syntax differs wildly between applications. Tcl was supposed to offer a canned, lightweight but powerful command language that would be easy to integrate into other programs; this would relieve software authors from having to come up with their own command language. Tcl would also make life easier for users, who would no longer need to learn one command language per program but could concentrate on Tcl instead. Tcl is like the UNIX shell because it essentially works by substituting strings (albeit in a much saner way than most shells); it is also like

LISP because the boundary between executable code and data is very fluid; and it borrows control structures and other concepts from languages like C. Tcl does not have much syntax—the basic job of the Tcl interpreter is to generate command strings by substituting variable contents and the results of other commands, parsing these strings into their constitutent "words" (which can be blocks of dependent Tcl code, like loop bodies) and looking up and executing the first word of the string as a Tcl command. Tcl commands can define their own syntax, and it is in fact possible to re-implement most Tcl control structures at the Tcl level. (Tcl has recently acquired a more sophisticated internal data representation and a byte-code compiler, which clouds this issue to a certain extent, but the basic principles remain the same.)

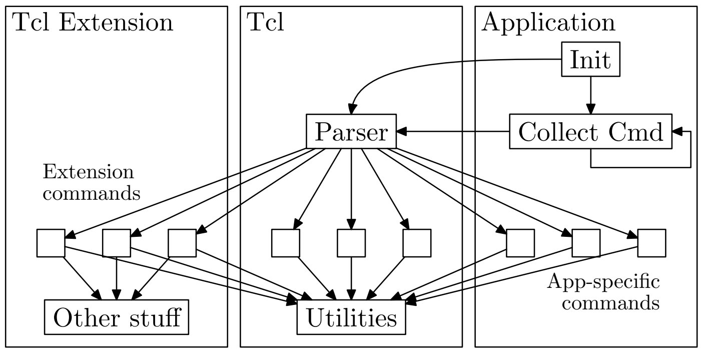

Figure 1: The structure of a Tcl-based application (adapted from [5])

Tcl is really a C library that can be linked into other programs. These other programs contain C code for additional Tcl commands that allow access to their specific functionality, and register these commands with the Tcl interpreter so that they can be used from within Tcl scripts. Tcl comes with a trivial "Tcl shell", which is the Tcl library together with a simple-minded main program that reads Tcl code from standard input and executes it. This is a convenient way to experiment with Tcl, and even to write applications in pure Tcl. It is also possible to extend a Tcl interpreter by dynamically loading modules written in languages like C (as well as Tcl). This inverts the original hierarchy of an application program including the Tcl library; here the Tcl shell includes application-specific code within an extension to serve as the basis of a particular application program (figure 1).

Tk is an X11 toolkit based on the Tcl language, much like the Xt toolkit is based on the C language. It offers a number of widget classes such as buttons, frames, labels, menu bars, or text entry fields, which can be configured and composed using Tcl code to form graphical user interfaces (see figure 2). It also allows Tcl actions to be triggered by events such as mouse movement, button or key presses, and interfaces with the window manager, the X11 option database and the X selection. One of the most powerful Tk widgets is the *canvas*, which is a base for structured graphics consisting of lines, polygons, ovals, text strings, images and similar items. The graphics items displayed on a canvas can be associated with events and actions, such that, for example, a mouse click on a picture of a floppy disk will invoke a Tcl script to display the directory of the disk currently in the floppy disk drive. More widgets can be defined through C code; it is possible to build additional widgets ("megawidgets") at the Tcl level but stock Tcl/Tk doesn't (yet) offer convenient support for this.

```
#!/usr/bin/wish
label .l -text {Hello Tcl/Tk world!} -background white
button .b -text {Exit} -command exit
pack .l .b -side top
```

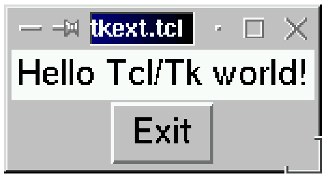

Figure 2: A simple Tcl/Tk program and its output

In addition to the Tcl shell, there is a similar program called `wish` ("window shell"), which is a Tcl shell that includes Tk. Even more than the Tcl shell, this has been used as the substrate for complete GUI applications. Of course, like the Tcl shell, `wish` can be extended through Tcl or dynamically-loaded binary modules.

There are literally hundreds of extensions and additions to Tcl and/or Tk, ranging from simple additional commands to support for networking protocols such as HTTP or FTP and environments for object-oriented programming or libraries of additional Tk widgets. Tcl/Tk has been ported to Microsoft Windows and the Apple Macintosh, but for the purposes of this paper it is convenient to think of Tk as an X11 toolkit. Also, various other programming languages such as Perl and Python have adopted Tk as their GUI.

The Tk Image Model

Since version 4.0, Tk contains a powerful mechanism to deal with *images* such as bitmaps or photorealistic graphics. In Tk, images can be displayed within various classes of widgets, such as buttons, labels and most notably the canvas widget. Figure 3 gives an overview of the Tk image model. Images are created using the `image` command, which takes as parameters the desired *image type* (in stock Tk, either `bitmap` for black-and-white images or `photo` for color images with dithering) as well as further information specific to the image type such as the size, the image format and the data source (a file or memory object). The result of the `image` command is the name of another, new command constructed by Tk which allows access and reconfiguration of the image. For example, `photo` images can be built up from pieces, copied partly or as a whole, with or without scaling or subsampling, individual pixels can be changed, the image data can be written to a file and so on. This image name is also used to refer the image from other widgets, e. g., `button .b -image foo` would make button `.b` contain the Tk image called `foo`. A given image can be used by various Tk widgets at the same time.

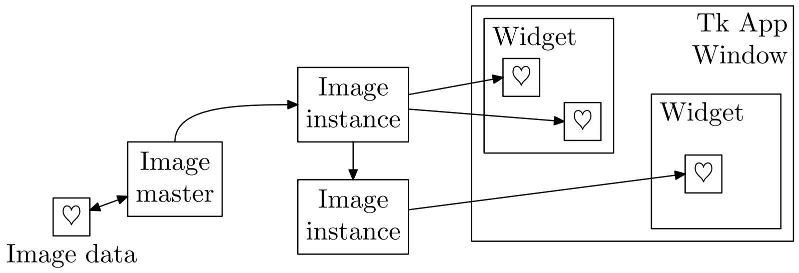

Figure 3: The Tk Image Model

At the C level, general information about an image (such as the configuration options specified when the image was created) are contained in an *image master* structure. For each occurrence of an image in a widget, there is also an *image instance* structure which keeps track of the widget- or display-specific aspects of the image, like colors or graphics contexts. It is possible to define new Tk image types by specifying a set of C functions which will be used by Tk when an image is about to be created, deleted, reconfigured, used by a widget and so on. Stock Tcl/Tk supports `photo` images in `GIF` and `PNM` formats as well as `bitmap` items in X11 bitmap format, but there is an extension by Jan Nijtmans called `Img` [4] which adds support for most common image formats such as

TIFF, JPEG, XPM or even PostScript (by calling out to an external PostScript interpreter such as Ghostscript).

The Architecture of TkDVI

The main goal behind the TkDVI project is to provide not only a state-of-the-art TEX previewer for UNIX systems, 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.

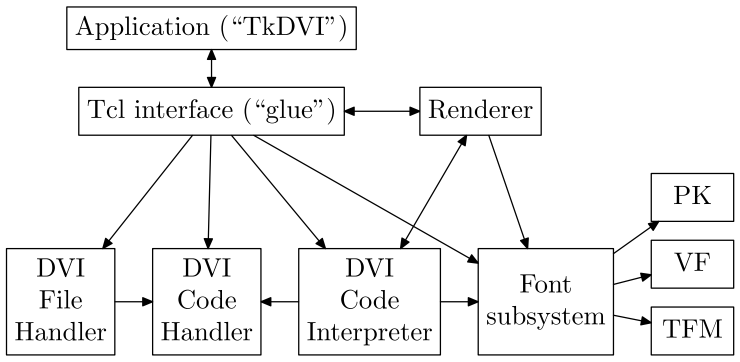

Figure 4: Main functional units within TkDVI. The arrows denote the direction of procedure calls.

The main functional units within TkDVI (see figure 4) 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 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.

The Implementation of TkDVI

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 (figure 5). 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.

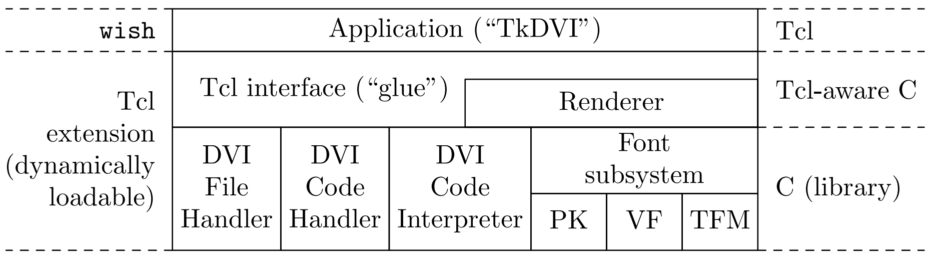

Figure 5: Implementation structure of TkDVI

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 *num* and *den* metric parameters 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.

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.

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"—$1/65536$ of a TEX `pt`) 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. To handle some DVI opcodes, the DVI interpreter needs font metrics, which it obtains from the font subsystem; it uses the canonical algorithm [3] to avoid pixel position rounding errors when displaying sequences of glyphs. Virtual fonts, which are essentially DVI code "macros", are implemented by recursive calls to the DVI interpreter from within itself.

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 or TrueType; 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.

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 User Interface

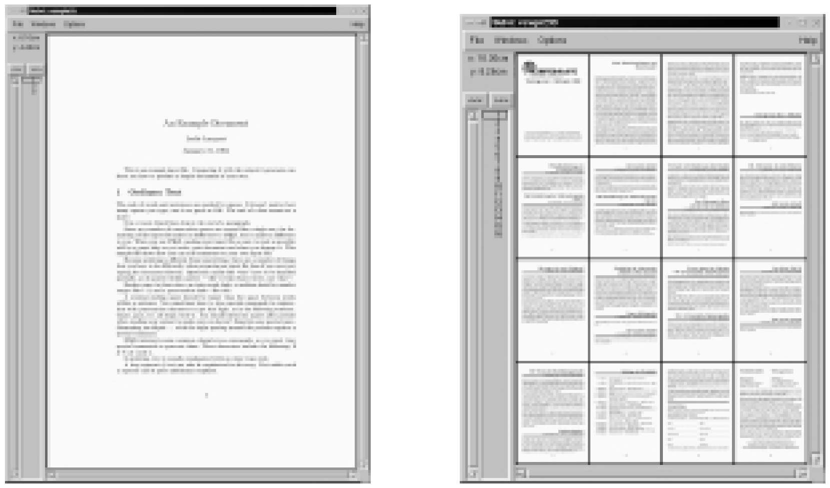

Figure 6: The TkDVI User Interface: Single-page mode (left), Overview mode (right)

The previous sections all dealt with the basic machinery necessary to handle DVI files and fonts and to render DVI material to an X11 device. What is still missing is a discussion of how this is all brought together to form a program that people can use. In a Tcl/Tk application, this is done through a user interface written in Tcl, which uses the groundwork laid in the C language code. It is important to note that the current TkDVI user interface is merely one possibility out of many; it would, for example, be straightforward to build a `xdvi`-like user interface.

TkDVI's user interface is patterned loosely on the `gv` Ghostscript front-end by Johannes Plass [8]. For example, it sports a "page selector" that displays a document's page numbers and lets the user 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 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 16-page reduced overviews. The former uses a double-width window and allows the viewer to judge the appearance 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. 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.

`\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` commands 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, the `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. Another straightforward application of this 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 this concerns security: what if a DVI file contains evil Tcl code that will format a viewer's hard disk? Fortunately, Tcl offers a mechanism [7] to execute untrusted code in a restricted environment where dangerous operations such as file access are not available in the first place. 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.

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 (beginning of July) is the lack of a mechanism for the inclusion of PostScript graphics. The `Img` extension offers limited access to **Ghostscript**, but it seems that for our requirements a purpose-built mechanism is necessary. This is still to be investigated in detail.

At the C language level, an interesting area to look at deals with color. The requirements for color management are set out succinctly in [9], 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 copy-editing 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. Therefore:

GO FORTH now and create *masterpieces of the previewing art!*

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

[4] Jan Nijtmans: "*Img Homepage*", See `http://home.wxs.nl/~nijtmans/img.html`, July 1999.

[5] John Ousterhout: "*Tcl: An Embeddable Command Language*", in *Proc. USENIX Winter Conf.*, pp. 133–146, January 1990.

[6] John Ousterhout: "*Tcl and the Tk Toolkit*", Addison-Wesley, Reading (MA), 1994.

[7] John K. Ousterhout, Jacob Y. Levy, Brent B. Welch: "*The SafeTcl Security Model*", Draft paper, see `http://www.scriptics.com/people/john.ousterhout/safeTcl.html`

[8] Johannes Plass: "*gv (PostScript viewer)*", see `http://wwwthep.physik.uni-mainz.de/~plass/gv/`

[9] Tomas Gerhard Rokicki: "*Driver Support for Color in TEX: Proposal and Implementation*", *TUGboat*, 15(3), pp. 205–212, September 1994.

Address

Anselm Lingnau
Schwanheimer Straße 66
60528 Frankfurt
Germany
E-Mail: `lingnau@tm.informatik.uni-frankfurt.de`