Inside PSTricks

Timothy Van Zandt
Department of Economics, Princeton University, Princeton, New Jersey USA
tvz@Princeton.EDU

Denis Girou
Institut du Développement et des Ressources en Informatique Scientifique
Centre National de la Recherche Scientifique, Orsay, France
Denis.Girou@idris.fr

Abstract

The macro-commands of the PSTricks package offer impressive additional capabilities to (LA)TEX users, by giving them direct access to much of the power of PostScript, including full support for color. The purpose of this article is to outline the *implementation* of a few of the features of PSTricks (version 0.94).

Introduction

When a PostScript output device and a dvi-to-ps driver are used to print or display TEX files, TEX and PostScript work together, as a preprocessor and a postprocessor, respectively. The role of PostScript may simply be to render TEX's dvi typesetting instructions. However, the full power of PostScript can be accessed through `\special`'s and through features, such as font handling, built into the dvi-to-ps driver.

One can divide the PostScript enhancements to TEX into roughly four categories:

1. The use of PostScript fonts.
2. The inclusion of PostScript graphics files.
3. The coloring of text and rules.
4. Everything else.

Most TEX-PS users are familiar with the first three categories. The PSTricks macro package, by Timothy Van Zandt, attempts to cover the fourth category.[1]

The PSTricks package started as an implementation of some special features in the Seminar document style/class, which is for making slides with LATEX 2_ε. However, it has grown into much more. Below are some of its current features:

1. Graphics objects (analogous to LATEX picture commands such as `\line` and `\frame`), including lines, polygons, circles, ellipses, curves, springs and zigzags.
2. Other drawing tools, such as a picture environment, various commands for positioning text, and macros for grids and axes.
3. Commands for rotating, scaling and tilting text, and 3-D projections.
4. Text framing and clipping commands.
5. Nodes and node connection and label commands, which are useful for trees, graphs, and commutative diagrams, among other applications.
6. Overlays, for making slides.
7. Commands for typesetting text along a path.
8. Commands for stroking and filling character outlines.
9. Plotting macros.

For information on PSTricks from the user's point of view, consult the PSTricks *User's Guide* (Van Zandt, 1994) and the article by Denis Girou (Girou, 1994) in *Cahiers GUTenberg*, the review of the French TEX users' group. The latter article is useful even to those who do not read French, because it consists predominantly of examples. Several of these examples appear in this paper, courtesy of *Cahiers GUTenberg*.

Who can use PSTricks?

A goal of PSTricks is to be compatible with any TEX format and any dvi-to-ps driver. Compatibility with the various TEX formats is not difficult to achieve, because PSTricks does not deal with page layout, floats or sectioning commands.

However, compatibility with all dvi-to-ps drivers is an unattainable goal because some drivers do

[1] PSTricks is available by anonymous ftp from `Princeton.EDU:/pub/tvz` and the CTAN archives.

not provide the basic `\special` facilities required by PSTricks. The requirements are discussed in subsequent sections. All of PSTricks' features work with the most popular driver, Rokicki's `dvips`, and most features work with most other drivers.

Two dvi-to-ps drivers that support the same `\special` facility may have different methods for invoking the facility. Therefore, PSTricks reads a configuration file that tells PSTricks how to use the driver's `\special`'s.

Header files

A PostScript header (prologue) file is analogous to a TeX macro file. It comes towards the beginning of the PostScript output, and contains definitions of PostScript procedures that can be subsequently used in the document.

It is always possible to add a header file to a PostScript file with a text editor, but this is very tedious. Most drivers support a `\special` or a command-line option for giving the name of a header file to be included in the PostScript output. For example, the `\special`

```
\special{header=pstricks.pro}
```

tells `dvips` to include `pstricks.pro`.

However, a few drivers, such as Textures (up through v1.6.2, but this may change) do not have this feature. Therefore, PSTricks can also be used without header files. From a single source file, one can generate a header file, an input file for use with headers, and an input file for use without headers.

For example, the main PSTricks source file, `pstricks.doc`, contains the line:

```
\pst@def{Atan}<%
  /atan load stopped{pop pop 0}if>
```

When generating the header file `pstricks.pro`, the line

```
/Atan {/atan load stopped{pop pop 0}if}def
```

is written to `pstricks.pro`. When generating the input file `pstricks.tex` for use with `pstricks.pro`, the line

```
\def\tx@Atan{Atan }
```

is written to the input file. The input file for use without `pstricks.pro` contains instead the line

```
\def\tx@Atan{%
  /atan load stopped{pop pop 0}if }
```

Other macros can use `\tx@Atan` in the PostScript code, without having to know whether it expands to a name of a procedure (defined in a header file) or to the code for the procedure (when there is no header file).

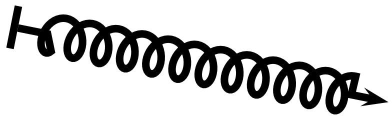
One can also use the source file directly, in which case no header is used. This is convenient when developing the macros, because TeX and PostScript macros can be written together, in the same file, and it is not necessary to make stripped input and header files each time one is testing new code.

The use of header files in PostScript documents reduces the size of the documents and makes the code more readable. However, the real benefit of using header files with PSTricks is that it substantially improves TeX's performance. It reduces memory requirements because, for example, the definition of `\tx@Atan` takes up less memory and, more importantly, `\tx@Atan` takes up less string space each time it is used in a `\special`. It reduces run time because the writing of `\special` strings to dvi output is very slow. A file that makes intensive use of PSTricks can run 3 to 4 times slower without header files!

Parameters and Lengths

To give the user flexible control over the macros, without having cumbersome optional arguments whose syntax is difficult to remember, PSTricks uses a key=value system for setting parameters.[2] For example,

```
\pscoil[coilarm=0.5,linewidth=1mm,
  coilwidth=0.5]{|->}(5,-1)
```



The `coilarm` parameter in this example is the length of the segments at the ends of the coil. Note that `coilarm` was set to `0.5`, without units. Whenever a length is given as a parameter value or argument of a PSTricks macro, the unit is optional. If omitted, the value of `\psunit` is used. In the previous example, the value of `\psunit` was `1cm`. Therefore, `coilarm=0.5cm` would have given the same result. Omitting the unit saves key strokes and makes graphics scalable by resetting the value of `\psunit`. This is why the arguments to LaTeX's `picture` environment macros do not have units. However, unlike LaTeX's `picture` macros, with PSTricks the unit can be given explicitly when convenient, such as `linewidth=1mm` in the previous example.

The implementation of this feature is simple. `\pssetlength` is analogous to `\LaTeX`'s `\setlength` command, but the unit is optional:

[2] PSTricks has recently adopted David Carlisle's improved implementation of the parsing, contained in the `keyval` package.

```
\def\pssetlength#1#2{%
  \let\@psunit\psunit
  \afterassignment\pstunit@off
  #1=#2\@psunit}
\def\pstunit@off{%
  \let\@psunit\ignorespaces\ignorespaces}
```

One advantage of the key=value system is that PSTricks has control over the internal storage of values. For example, PSTricks stores most dimensions as strings in ordinary command sequences, rather than in dimension registers. It uses only 13 of the scarce dimension registers, whereas, for example, PICTEX uses over 120. When PSTricks processes the parameter setting `coilarm=0.5`, it executes:

```
\pssetlength\pst@dimg{0.5}
\edef\psk@coilarm{\pst@number\pst@dimg}
```

`\pst@dimg` is a register. `\pst@number\pst@dimg` expands to the value of `\pst@dimg`, in `pt` units, but without the `pt`. Hence, `\psk@coilarm` is ready to be inserted as PostScript code.

Color

To declare a new color, the user can type:

```
\newrgbcolor{royalblue}{0.25 0.41 0.88}
```

The color can then be used to color text and can be used to color PSTricks graphics. For example:

```
\psframebox[linewidth=2pt,framearc=.2,
  linecolor=royalblue,framesep=7pt]{%
  \LARGE\bf It's {\royalblue here} now!!}
```

It's here now!!

The `\newrgbcolor` command defines `\royalblue` to switch the text color, and it saves the color specification under the identifier `royalblue` so that the PostScript code for setting the color can be retrieved by color graphics parameters.

This support for color has been part of PSTricks since its inception. However, a problem that has arisen is that there are now many packages available for coloring text, and the user is likely to end up using some other color package in conjunction with PSTricks. But then the color names used for text cannot be used with PSTricks graphics parameters.

It is therefore important that a dominant set of color macros emerge in the TEX community, and that the macros allow the PostScript code for the declared colors to be accessible, in a standard way, by packages such as PSTricks. Version 0.94 of PSTricks is distributed with an independent set of color macros that may be a prototype for such a standard color package.

Arithmetic

One of the limitations of TEX is its lack of fast, floating-point arithmetic. It is possible to write routines for calculating, for examples, sines and cosines using TEX's integer arithmetic, but these are notoriously slow. Therefore, PSTricks offloads such arithmetic to PostScript, whenever possible.

Such offloading is not always possible because PostScript cannot send information to TEX. If TEX needs to know the result of some calculation, it must do the calculation itself. For example, suppose that one wants a macro that puts a triangle around a TEX box, analogous to LATEX's `\fbox` command. The macro can measure the TEX box, and pass these dimensions to a PostScript procedure via a `\special`. PostScript can then use its trigonometric functions to calculate the coordinates of the vertices of the triangle, and then draw the triangle. However, it may be important for TEX to know the bounding box of the triangle that is drawn, so that the triangle does not overlap surrounding text. In this case, TEX must do (slowly) the trigonometric calculations itself.

Pure graphics

A large chunk of PSTricks consists of graphics macros, which you can think of as a fancy replacement for LATEX's `picture` environment. The qualifier "pure" means that the graphics do not interact with TEX. For example, a rectangle is "pure", whereas a framed box is not.

A pure graphics object scans arguments and puts together the PostScript code *ps-code* for the graphics. When the code is ready, the object concludes with:

```
\leavemode\hbox{\pstverb{ps-code}}
```

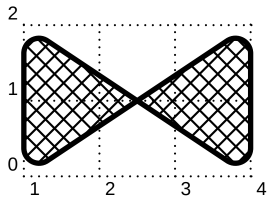
`\pstverb` should be defined in the configuration file to insert the code in a `\special` that reproduces *ps-code* verbatim in the PostScript file, grouped by PostScript's `save` and `restore`. The graphics state should have PostScript's standard coordinate system (`bp` units), but with the origin at TEX's current point. For `dvips`, the definition of `\pstverb` is:

```
\def\pstverb#1{\special{" #1}}
```

This `\special` is the only output generated. Thus, within TEX, the object produces a box with zero height, depth and width. Within PostScript, the graphics object is grouped by `save` and `restore`, and hence has no effect on the surrounding output.

For example, here is a polygon:

```
\pspolygon[linewidth=2pt,
  linearc=.2,fillstyle=crosshatch]
  (1,0)(1,2)(4,0)(4,2)
```



\pspolygon first invokes \pst@object, which collects (but does not processes) optional parameter changes, and subsequently invokes \pspolygon@i:

```
1  \def\pspolygon{\pst@object{pspolygon}}
2  \def\pspolygon@i{%
3    \begin@ClosedObj
4      \def\pst@cp{}%
5      \pst@getcoors[\pspolygon@ii}
```

\begin@ClosedObj (line 3) performs various operations that are common to the beginning of closed graphics objects (as opposed to open curves), such as processing the parameter changes and initializing the command \pst@code that is used for accumulating the PostScript code. \pst@getcoors (line 5) processes the coordinates one at a time (\pspolygon can have arbitrarily many coordinates), converting each one to a PostScript coordinate and adding it to the PostScript code in \pst@code.

Then \pst@getcoors invokes \pspolygon@ii:

```
6  \def\pspolygon@ii{%
7      \addto@pscode{\psline@iii \tx@Polygon}%
8      \def\pst@linetype{1}%
9    \end@ClosedObj}
```

Line 7 adds the PostScript code that takes the coordinates from the stack and constructs the path of the polygon. \pst@linetype (line 8) is used by the dashed and dotted linestyles to determine how to adjust the dash or dot spacing to fit evenly along the path (the method is different for open curves and open curves with arrows). Then \end@ClosedObj (line 9) performs various operations common to the ending of closed graphics objects, such as adding the PostScript code for filling and stroking the path and invoking \pstverb.

Here is the resulting PostScript code for this example:

```
1  tx@Dict begin STP newpath 2 SLW 0 setgray
2  [ 113.81097 56.90549 113.81097 0.0
3  28.45274 56.90549 28.45274 0.0
4  /r 5.69046 def
5  /Lineto{Arcto}def
6  false Polygon
7  gsave
8    45. rotate 0.8 SLW 0. setgray
9    gsave 90 rotate 4.0 LineFill grestore
10   4.0 LineFill
11 grestore
12 gsave 2 SLW 0 setgray 0 setlinecap stroke
13 end
```

Line 1 is added by \begin@ClosedObj. STP scales the coordinate system from PostScript's bp units to pt units, which are easier for TeX to work with (e.g., \the\pslinewidth might expand to 5.4pt, and the pt can be stripped).

Lines 2 and 3 are the coordinates, which are added by \pst@getcoors.

Line 4 sets the radius for the rounded corners and line 5 defines Lineto, a procedure used by Polygon, so that it makes rounded corners. If the linearc parameter had been 0pt instead, then, instead of lines 4 and 5, \psline@iii would have added /Lineto{lineto}def.

Lines 7 to 11 are added by the fillstyle, and line 12 is added by the linestyle, both of which are invoked by \end@ClosedObj.

The code for the graphics objects is highly modular. For example, nearly all graphics objects invoke the fill style to add the PostScript code for filling the object. To define a new fill style foo for use with all such objects, one simply has to define \psfs@foo to add the PostScript code for filling a path.

The graphics objects can be used anywhere, and can be part of composite macros such as for framing text. However, they are most commonly used by the end-user to draw a picture by combining several such objects with a common origin. For this purpose, PSTricks provide the pspicture environment, which is very similar to LaTeX's picture environment. In particular, it is up to the user to specify the size of the picture. This is an unfortunate inconvenience, but one that is insurmountable. The PSTricks graphics objects include curves and other complex objects of which TeX could not calculate the bounding box, at least not without doubling the size of PSTricks and slowing it to a crawl. This is the main way in which TeX's lack of graphics and floating point capabilities hinders PSTricks.

Nodes

Drawing a line between two TeX objects requires knowledge of the relative position of the two objects on the page, which can be difficult to calculate. For example, suppose one wants to draw a line connecting "his" to "dog" in the following sentence:

The dog has eaten his bone.

One could calculate the relative position of these two words, as long as their is not stretchable glue in the sentence, but the procedure would not be applicable to connecting other objects on a page.

With PostScript as a postprocessor, there is a straightforward solution. By comparing the transformation matrices and current points in effect at two points in the PostScript output, one can determine their relative positions. This is the basic idea that lies behind PSTricks node and node connection macros, and is one that PSTricks adapted from Emma Pease's `tree-dvips.sty`.

Here is how PSTricks connects the words:

```
\large
The \rnode{A}{dog} has eaten
    \rnode{B}{his} bone.
\ncbar[angle=-90,nodesep=3pt,arm=.3]{->}{B}{A}
```

The dog has eaten his bone.

`\rnode{A}{dog}` first measures the size of "dog". Then it attaches to "dog" some PostScript code that creates a dictionary, `TheNodeA`, with the following variables and procedures:

`NodeMtrx`	The current transformation matrix.
`X`	The x-coordinate of the center.
`Y`	The y-coordinate of the center.
`NodePos`	See below.

Here is the code that appears in the PostScript output for this example:

```
1  tx@Dict begin
2    gsave
3      STV CP T
4      8.33331 2.33331 18.27759
5      9.1388 3.0
6      tx@NodeDict begin
7        /TheNodeA 16 NewNode
8          InitRnode
9        end
10     end
11   grestore
12 end
```

This codes gets inserted with `\pstVerb`, which should be defined in the configuration file to include *ps-code* verbatim in the PostScript output, *not* grouped by (g)`save` and (g)`restore`. PostScript's current point should be at TEX's current point, but the coordinate system can be arbitrary. For `dvips`, the definition of `\pstVerb` is:

```
\def\pstVerb#1{\special{ps: #1}}
```

`\pstVerb` is used instead of `\pstverb` because the latter groups the code in `save` and `restore`, which would remove the node dictionary from PostScript's memory. However, PSTricks still wants to work in `pt` units, and so `STV` scales the coordinate system.

Line 4 contains the height, depth and width of the `dog`. The next line (9.1388 3.0) gives the x and y displacement from where the code is inserted (on the left side of dog, at the baseline) to the center of `dog`. Actually, by "center" we mean where node connections should point to. This is the center by default, but can be some other position. For example, there is a variant `\Rnode` that sets this point to be a fixed distance above the baseline, so that a horizontal line connecting two nodes that are aligned by their baselines will actually be horizontal, even if the heights or depths of the two nodes are not equal.

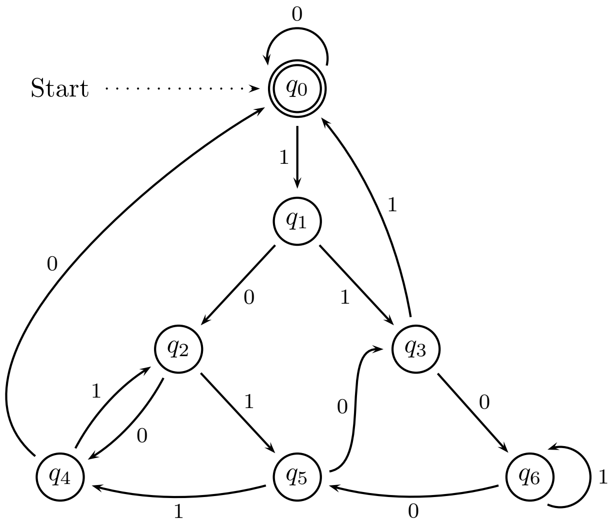
`NewNode`, in line 7, performs various operations common to all nodes, such as creating a dictionary and saving the current transformation matrix. Then `InitRnode` takes the dimensions (lines 4 and 5) off the stack and defines `X`, `Y` and `NodePos`.

A node connection that wants to draw a line between a node named `A` and a node named `B` can go anywhere after the nodes, as long as it ends up in the `dvi` file after the nodes, and on the same page. The node connection queries the node dictionaries for the information needed to draw the line. In the example above, `\ncbar` needs to know the coordinate of the point that lies on the boundary of "his", at a -90° angle from the center of node. After setting `Sin` and `Cos` to the sine and cosine of 90° and setting `NodeSep` to 0, the procedure `NodePos` in the `TheNodeA` dictionary returns the coordinates of this point, relative to the center of the node. The connection macro can then convert this to coordinates in the coordinate system in effect when the node connection is drawn, by retrieving and using `NodeMtrx`, `X` and `Y` from `TheNodeA`.

A node connection macro, after drawing the connection, should also save a procedure for finding the position and slope of a point on the line, so that labels can be attached to node connections. This task is similar to that of a node; it should save the coordinates of the node connection and the current transformation matrix and a procedure for extracting from this information a position on the node connection. Example 1 makes extensive use of labels.

There are many ways to position nodes, depending on the application. To create a diagram with arrows from one object to another, one can position the objects in a `pspicture` environment. For applications with more structure, one may want

```
\psmatrix
  [mnode=circle,colsep=.85cm,rowsep=1cm]
  % States:
  [mnode=R]{\mbox{Start}}
    & & [doubleline=true,name=0]$q_0$ \\
  & & [name=1]$q_1$ \\
  & [name=2]$q_2$ & & [name=3]$q_3$ \\[0pt]
  [name=4]$q_4$ & & [name=5]$q_5$ & &
      [name=6]$q_6$
\endpsmatrix
% Transitions:
\psset{nodesep=3pt,arrows=->,arcangle=15,
  labelsep=2pt,shortput=nab}
\footnotesize
\ncline[linestyle=dotted]{1,1}{0}
\nccircle{0}{.4cm}_{0}
\ncline{0}{1}_{1}
\ncline{1}{2}^{0}
\ncline{1}{3}_{1}
\ncarc{2}{4}^{0}
\ncarc{4}{2}^{1}
\ncline{2}{5}^{1}
\ncline{3}{6}^{0}
\ncarc{<-}{0}{3}^{1}
\nccurve[angleA=140,angleB=210]{4}{0}^{0}
\nccurve[angleA=10,angleB=180]{5}{3}^{0}
\ncarc{5}{4}^{1}
\ncarc{6}{5}^{0}
\nccircle[angleA=270]{6}{.4cm}_{1}
```



Example 1: An example of nodes and node connections and labels, used with the `psmatrix` environment. (Courtesy of Mark Livingston.)

a more automated way to position nodes. PSTricks does not come with any high-level macros explicitly for commutative diagrams, but it does have a `psmatrix` environment for aligning nodes in an array, and this can be used for commutative diagrams. Example 1 shows `psmatrix` beings used for a graph. PSTricks also contains very sophisticated tree macros.

Overlays

To make overlays with SLiTeX, for example, you have to use invisible fonts, and TeX has to typeset the slide once for each overlay. This makes it impossible to make overlays if a slide uses fonts other than the few for which invisible versions are available, or if the slide contains non-text material.

PSTricks uses a simple idea for creating overlays. Its operation is illustrated in Example 2. A box from which a main slide and overlays are to be created is saved, using the `overlaybox` environment. The `\psoverlay{2}` command in this box simply inserts the code

```
(2) BeginOL
```

and similar code at the end of the current TeX group to revert to the `main` overlay. `BeginOL` compares the string on the top of the stack to the PostScript variable `TheOL`. If it does not match, the output is made invisible. Otherwise, it is made visible. To print out overlay `2`,

```
\putoverlaybox{2}
```

simply has to insert

```
/TheOL (2) def
```

before a copy of the box.

Because we can insert PostScript procedures in the box that can be redefined before each copy of the box, TeX only has to typeset the box once, which saves processing time and saves us from having to come up with a way to read the TeX input for the box several times.

There are several ways to make output invisible with PostScript, none of which is entirely satisfactory. PSTricks' default method is to translate everything far away (e.g., over by the coffee pot) so that, except in very unusual circumstances, all the "visible" output ends up off the page. This is easy to undo, by translating back.

The only problem with translation is that the node connections and labels, which use absolute coordinates, end up on the same overlay as the nodes that are connected. Therefore, users can select an alternate method for making material invisible: setting a small clipping path off the page. The problem with this method is that it can only be

```
\large
\begin{overlaybox}
  $\frac{n-2}{n-3}
    + \psframebox{\psoverlay{2}
   \frac{n-1}{n}}
    = \frac{2(n-2)(n-1)}{n(n-3)}$
\end{overlaybox}
\psset{boxsep=6pt,framearc=.15,
  linewidth=1.5pt}
\psframebox{\putoverlaybox{main}}
\psframebox{\putoverlaybox{2}}
```

$$\boxed{\frac{n-2}{n-3} + \boxed{\phantom{\frac{n-1}{n}}} = \frac{2(n-2)(n-1)}{n(n-3)}}$$

$$\boxed{\quad\frac{n-1}{n}\qquad\qquad\qquad}$$

Example 2: Overlays.

undone with `initclip`, which can mess up other macros that set the clipping path.

PSTricks does not use PostScript's `nulldevice` operator, because this cannot be undone except by using `grestore`. It would thus be impossible to have nested overlays. The PSTricks overlay macros are used to implement overlays in the Seminar package.

Typesetting text along a path

One facility that TEX users have long desired but have been unable to obtain is to typeset text along a path. This is a task that also stretches the limits of PostScript `\special`'s, but PSTricks contains an implementation that works for several dvi-to-ps drivers. It is illustrated in Color Example 4.

The main difficulty is that the text that goes along the path should be typeset by TEX, not by PSTricks, and then converted to PostScript output by the dvi driver. For PSTricks to get this text along the path, it has to redefine the operators that the dvi driver uses to print the text. This requires knowledge of the PostScript code the dvi driver uses to print text.

In the best case, the dvi driver simply uses PostScript's `show` operator, unloaded and unbound. PSTricks simply has to redefine `show` so that it takes each character in the string and prints it along the path. The redefined `show` checks the current point and compares it with the current point at the beginning of the box that is being typeset to find out the x and y positions of the beginning of the character. The x position is increased by half the width of the character to get the position of the middle of the character. This is the distance along the path that the middle of the character should fall. It is straightforward, albeit tedious, to find the coordinates and slope of any point on a path. We translate the coordinate system to this point on the path, and then rotate the coordinate system so that the path is locally horizontal. Then we set the current point to where the beginning of the character should be, which means to the left by half the character width and up or down by the relative position of the base of the character in the box. Then we are ready to show the character.

This method works with Rokicki's `dvips`. For other drivers, one of two problems arises:

1. `show` is "loaded" or "bound" in procedures defined by the driver for displaying text. This means that the procedures do not invoke the command name `show`, which can be redefined by PSTricks, but instead invoke the primitive operation `show`, which cannot be altered. The workaround for this is to remove the appropriate `load`'s and `bind`'s from the driver's header file.

2. The driver uses PostScript Level 2's large family of primitives for showing text. The only workaround is to redefine all these operators, which has not been attempted. The usual dvi drivers do not use Level 2 constructs. However, NeXTTEX's TEXView, which is a dvi driver based on the NeXT's Display PostScript windowing environment, does use Level 2 operators. The workaround for NeXT users is to use `dvips` to generate a PostScript file and then preview it with Preview.

Stroking and filling character paths

It is also possible to stroke and fill character paths, as illustrated in Color Example 4. The methodology is the same as typesetting text along a path, but it is easier because `show` just has to be changed to `charpath`. Nevertheless, the two problems that can trip up PSTricks' `\pstextpath` macro can also trip up `\pscharpath`. Furthermore, `\pscharpath` only works with PostScript outline fonts, since bitmap fonts cannot be converted to character paths.

Charts

PSTricks has many primitives for a wide variety of applications, but sophisticated graphics can involve tedious programming. In such cases, a preprocessor can be constructed to automatically generate the PSTricks commands. The preprocessor can generate standardized representations using only a minimum amount of information, but the user does not lose flexibility because the PSTricks code can subsequently be tweaked as desired.

For instance, we can think of preprocessors for automatic coloration of maps, generation of graphs or trees, etc. For his own needs, Denis Girou has written (in Shell and AWK) a preprocessor (`pstchart.sh`) for automatic generation of pie charts, which he extended to generate other forms of business graphics (line and bar graphs, 2D or 3D, stacked and unstacked).

Example 3 shows a data file, the unix command line for generating the PSTricks code from the data file, and the output. Color Example 3 shows the output from another example, generated with the unix command line:

```
pstchart.sh vbar dimx=9 3d boxit center\
figure print-percentages < file2.data
```

Conclusion

There is much talk about the future of TEX and about the need to create a replacement for TEX because TEX is, by design, just a typesetting program for positioning characters and rules. We believe that when today's TEX is supplemented by PostScript, through the use of `\special`'s and good dvi-to-ps drivers, many of the special effects that users clamor for can be achieved today. PSTricks provides an example of this.

When PSTricks is combined with the Seminar LATEX 2_ε document class for making slides, plus PostScript fonts and macros for including graphics files, one has a complete presentation software package, that is quite far from the usual use of TEX for typesetting technical papers.

However, there are still some limitations that can only be solved by changes to TEX. The most obvious one is TEX's lack of fast, floating-point arithmetic. Although TEX can pass information to PostScript through `\special`'s, it is not possible for PostScript to pass information to TEX. This slows down many calculations and makes it impossible to calculate the bounding box of some graphics.

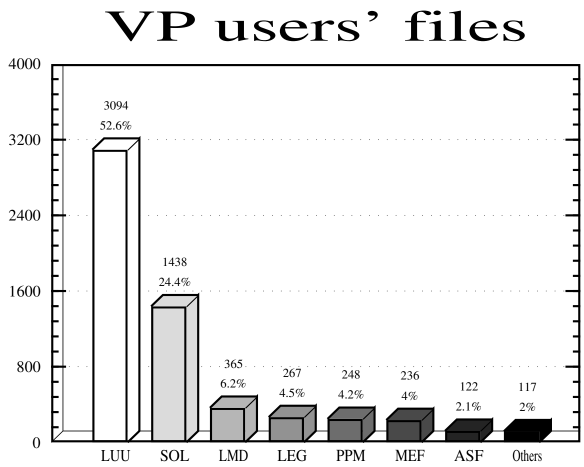
Data file:

```
3094 | LUU
1438 | SOL
 365 | LMD
 267 | LEG
 248 | PPM
 236 | MEF
 122 | ASF
  57 | DRT
  33 | AMB
  18 | TPR
   9 | RRS
```

Command line:

```
pstchart.sh vbar dimx=7 3d nb-values=8 \
print-percentages print-values \
grayscale=white-black data-change-colors \
title="VP users' files" center <users.data
```

Output:



Example 3: Using the preprocessor `pstchart` to generate PSTricks graphs.

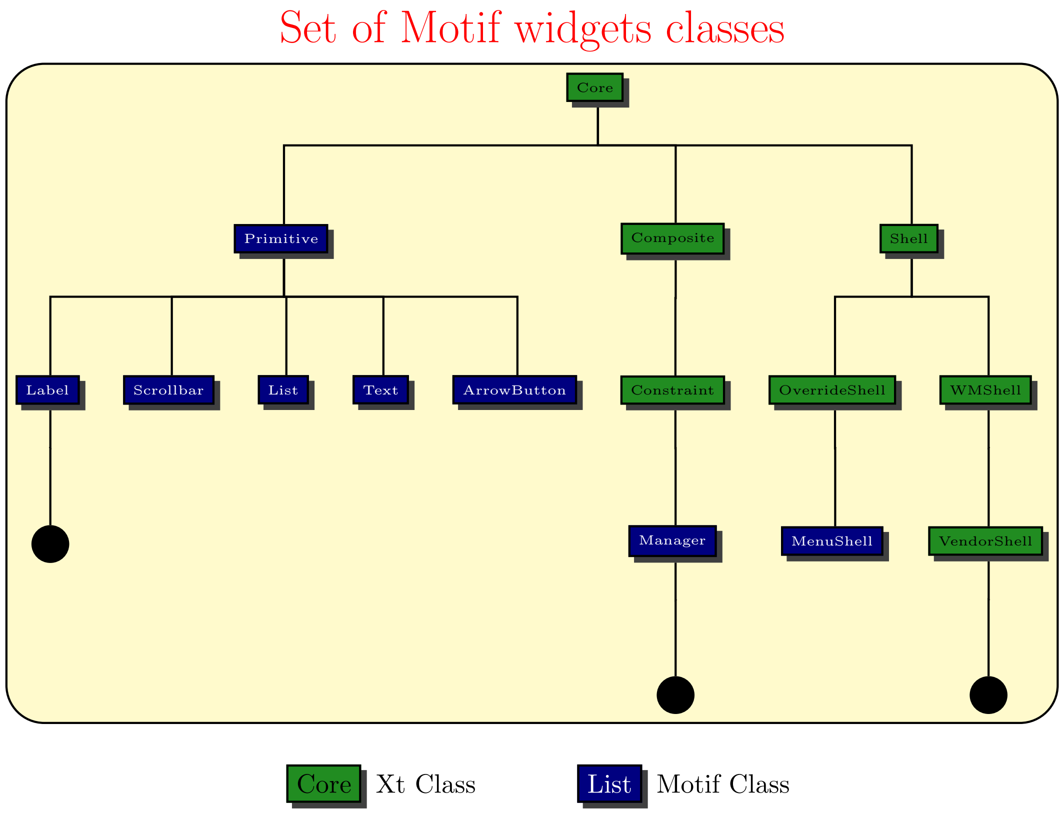
References

Girou, Denis. "Présentation de PSTricks," *Cahiers GUTenberg*, No. 16, pp. 21–70, Février 1994.

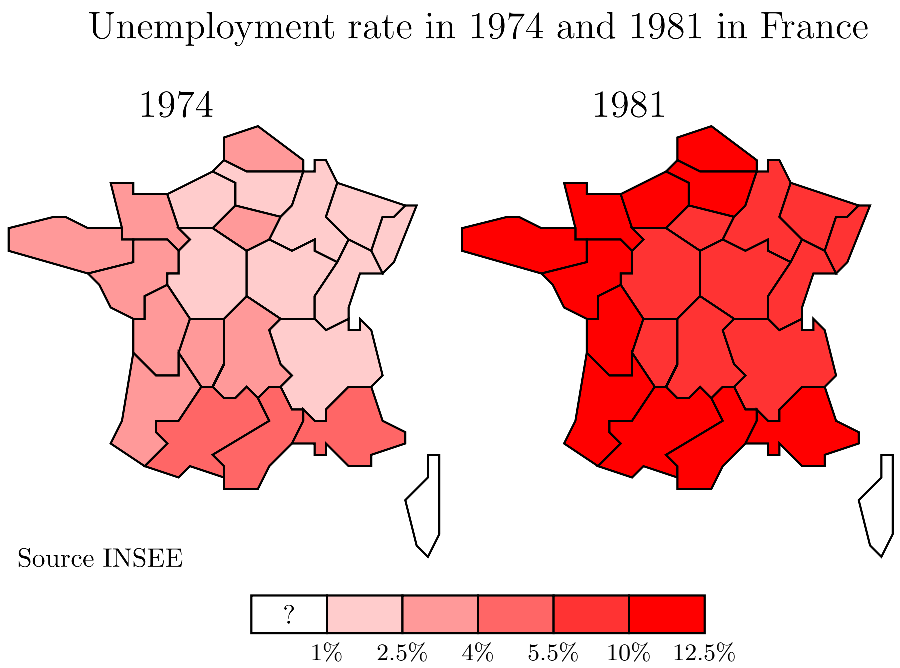
Van Zandt, Timothy. "PSTricks: Documented Code." 1994

Van Zandt, Timothy. "PSTricks: PostScript Macros for Generic TEX — User's Guide." 1994.

Example 4: Typesetting text on a path, filling character outlines, and using a TEX box as a fill pattern.



Example 5: Tree representation.



Example 6: Coloration of maps.