Mixing TEX & PostScript : The GEX Model

Alex Kostin & Michael Vulis
MicroPress, Inc.
68-30 Harrow Street
Forest Hills, New York, 11375, USA
Phone: 1 (718) 575 1818
Fax: 1 (718) 575 8038

support@micropress-inc.com
http://www.micropress-inc.com

Abstract

VTEX is a (commercial) extended version of TEX, sold by MicroPress, Inc. Free versions of VTEX have recently been made available, that work under OS/2 and Linux. This paper describes GEX, a fast fully-integrated PostScript interpreter which functions as part of the VTEX code-generator.

GEX offers one-pass compilation of text (TEX) and graphics and thus easy incorporations of graphics files (`.eps`) and inline PostScript code (PStricks, PSfrag) within a document. While it is this graphics support which is of primary interest to the end users, the presence of the PostScript interpreter within TEX and its ability to provide feedback to TEX raises interesting questions about mixing text and graphics in general and leads to new graphics-oriented packages.

This article serves as a short introduction to GEX, seeking to explain the design issues behind GEX and the extensions which now become possible.

Unless specified otherwise, this article describes the functionality in the freeware version of the VTEX compiler, as available on CTAN sites in `systems/vtex`.

What did GEX come from?

During the early work on the VTEX PDF backend circa 1998 it became apparent that the only way the backend can handle PStricks graphics is by incorporating a limited PostScript interpreter. GEX (which stands for Graphics EXtensions and is to be pronounced `g-e-ks`) arose primarily from the author's misguided optimism about the amount of work required. By the time GEX fully supported PStricks, the code amounted to more than 20,000 lines of `C++` code, supported almost the entire PostScript language, and even went beyond it. GEX has become powerful enough to handle not only PostScript files (`.eps`) but also the common inline PostScript graphics packages (PStricks, PSfrag, XYpic, or Seminar). In addition, it has become possible to design new macro packages with GEX in mind.

While the `.eps` file and inline PostScript inclusion is the main attraction to the end user, this article has very little to say about it. This is because from the end-user standpoint, using GEX amounts to using standard and familiar commands like `\includegraphics` or `\begin{pspicture}` and seeing the results appear as expected in the output. Instead, we will concentrate on the design issues and the extensions.

What is GEX ?

GEX is a graphics counterpart to TEX. The basic design assumes that TEX is responsible for handling of the text; GEX is responsible for processing the graphics components of the document. Both TEX and GEX contribute to the output; since the overall handling of the document is TEX's responsibility, TEX has overall control.

Usually, but not always, GEX functions within the TEX `\shipout` routine and accepts responsibilities which would otherwise be given to a DVI driver. In more interesting cases, GEX functions during the TEX formatting phase; when so doing, it is capable of returning information to TEX and thus influencing TEX formatting.

Since GEX may exercise subtle influence on TEX (load fonts, or change TEX registers), GEX is optional in VTEX implementations: the default operation of the program is with GEX off; it is enabled by a command-line switch.

Of the four primary output modes of the VTEX compiler (DVI, HTML, PDF, PostScript), GEX is currently supported with two: PDF and PostScript. The majority of GEX-related activities are identical in these two modes. Where a behavioral difference is desired, a macro writer can use the `\OpMode` count primitive (with magic values of 0,1,2 and 10 for DVI, PDF, PS, and HTML output modes).

In PDF mode, GEX is basically a PS↦PDF compiler; in the PS mode, it is a PS↦PS compiler which reinterprets input PostScript and produces output similar to what would be produced by printing PDF to PostScript, albeit faster, often tighter and cleaner. (One of the benefits of this in comparison with DVI↦PS drivers is the combination of the fonts and other resources that are often repeated in included `.eps` files.)

While GEX is a PostScript language interpreter, it is not 100% PostScript; there are subtle design differences, that while not impeding the ability of GEX to process standard PostScript code, allow new applications.

The basic design paradigms

During the design of GEX it has become apparent that a number of extensions will be needed to be added to TEX to support the extra functionality. In all cases, the basic approach was to try to keep the TEX syntax as close to the standard as possible, and avoid introducing additional keywords. Most of the TEX language extensions[1] are merely `\specials` which are understood and resolved by the `\shipout` code in VTEX. Thus, VTEX syntax would not have new words like `\pdfimage` or `\pdfoutline`; these would be backend `\specials`. In practice, we did end up with adding some primitives, but these were primarily new `count` and `skip` registers.

In designing the syntax of the `\specials` themselves, an attempt has been made to avoid dependency on the PDF output mode. This makes them either applicable or at least safely ignorable in other operation modes of VTEX (DVI, HTML), not just in the PDF and PostScript modes, where GEX is fully operational. Thus, VTEX's `\special` never uses PDF-specific code. While a direct write to the output is supported (with `\special{!=...}`, analogous to `\pdfliteral{...}` in pdfTEX), it is generally discouraged.

Finally, wherever possible, the `\specials` are screened from the user, mostly by means of extending the `graphicx` package.

[1] There are also PostScript language extensions in play.

How does it work

The basic model of TEX-GEX interaction is the two `\specials`:

- `\special{pS: .....}` with the argument containing valid PostScript code
- `\special{ps: .....}` with the argument being a name of PostScript file to include

When the backend sees one of these `\specials`, it passes it to GEX for compilation. (In PDF mode, with GEX off, it is simply thrown out; with GEX on, it is compiled. In PostScript mode, with GEX off, the parameter is pasted to the PostScript output, as in traditional DVI↦PS drivers; with GEX on, it is re-compiled).

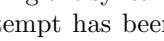
Prior to giving control over to GEX, VTEX updates the information in PostScript's graphics state (setting the coordinates for the current point, for example). Upon the return from TEX, the *relevant* parts of the PostScript graphics state are given back to TEX.

Because of the need to support inline PostScript packages the information about the current font is also shared between TEX and GEX. For example, passing

```
\special{pS: currentfont setfont}
```

to the PostScript interpreter is entirely legal (and is done by PStricks); but the design implication is that GEX is aware of the currently used TEX font and can access it by itself. Access may mean actually loading the font and executing the instructions in the font file; this would happen, for example, if one writes

```
\special{pS: gsave currentfont 2
  scalefont setfont 0.5 0 0 setrgbcolor
  [4 1] 0 setdash (Test stroke) false
  charpath stroke grestore}
```

which yields 

Observe that here we use `gsave`/`grestore` to screen TEX (and subsequent PostScript fragments) from the color and dash changes done in GEX.

Design implications: the font machinery is to be shared between TEX and GEX; GEX should be able to load TEX fonts and operate on them.

Solution: Provide all conceivably useful TEX fonts in Type 1. Extend GEX with command(s) for loading a font given its TEX name and point size (the `.settexfont` PostScript extension which loads the current TEX font into PostScript at the current size, as well as the `.loadfont` extension which allows GEX to select any TEX font by its name. The second extension is of use for MetaPost; see below.).

Unresolved issues: The ability to pass fonts to GEX is currently unsupported on TrueType or CID fonts (used in CJK PDF generation). Thus, no font effects are currently possible on Asian fonts[2].

Error handling

Unlike in TEX, error correction of PostScript code is hardly possible. Errors are therefore converted into TEX-style errors, followed by PostScript-style stack dumps.

For example, passing the following code to GEX

```
\special{pS: 1 2 movetoo}
```

results in

```
! PS interpreter error, code=21
 (Undefined name [movetoo]).
```

(This assumes, of course, that the `movetoo` name has not been previously defined.) GEX errors are usually fatal; while the TEX portion of the document can be compiled through the end, PostScript compilation is abandoned on the first error. Only with several common errors, like font unavailability, or leaving junk entries on the operand stack (as explained below), will GEX continue.

Interesting cases

While the above model is sufficient *for most cases*, there are unusual situations which arise in specific cases.

Typesetting text on a curve which is the activity of the `pst-path` component of PStricks provides one difficulty. In PStricks, this is implemented by redefining the `show` operator. In conventional PStricks, the expectation is that the redefined `show` should hack the code which originated in TEX, but now, after TEX↦DVI and DVI↦PS conversions, the code has become PostScript. In the VTEX case, we want PStricks to work within the TEX `\output` routine, where there is no PostScript code to hack.

Solution: The VTEX backend senses the redefinitions of PostScript text output operators like `show`; if it detects that `show` has been changed it temporary switches to PostScript generating mode; then passes the output to GEX for recompiling.

A similar situation arises when a TEX macro package "cuts out" a piece of PostScript code for reuse or discarding. Both PStricks and PSfrag do it by inserting a definition around PostScript code generated by TEX:

```
/something {
<ps code>
} def
```

Design implications: The TEX backend must sense when GEX is in such a "definition" mode, and switch to PostScript generation if needed. In the above example, upon processing

```
\special{pS: /something {}
```

GEX returns back an indicator that it did not fully handle the operator; only after

```
\special{pS: } def }
```

will the TEX backend be allowed to return to normal processing.

Transfer handling

⟨*int*⟩ **.enabletransfer** A problem which arises with some `.eps` images is the use of the `settransfer` PostScript and related operators. The problem is that these operators are used for both device-dependent and device-independent color manipulations. The first usage is more common and is essentially for minor color adjustments. In such situations the best strategy for producing device-independent PDF files is to disregard the transfer altogether. This is the default behaviour of GEX (and of the Acrobat Distiller).

However, in some (fortunately, rare) `.eps` files the same operators are used to effect major device-independent adjustments. An example of such an adjustment would be to invert a black-and-white picture; this can be done with the

```
{ 1 exch sub } settransfer
```

PostScript code snippet. Disregarding this code will produce an inverted image. Thus, both Acrobat Distiller and GEX allow the user to change their behaviour. In the case of Distiller, the override is a global Job option which will apply to all parts of a document; GEX allows you to override only the handling of an individual image. This is accomplished with the extension operator

```
.enabletransfer
```

With an argument of zero, `.enabletransfer` disables processing of `settransfer` code; a non-zero argument enables `settransfer` processing. Figure 1 is an example of a small `.eps` file that uses transfer code.

MetaPost support

While GEX can handle MetaPost-generated files, it is important to state that MetaPost outputs invalid EPS files. Rather than use the standard fonts or embed fonts into the EPS, MetaPost merely includes declarations like

[2] see `http://www.micropress-inc.com/CJK` for additional information on CJK/PDF support.

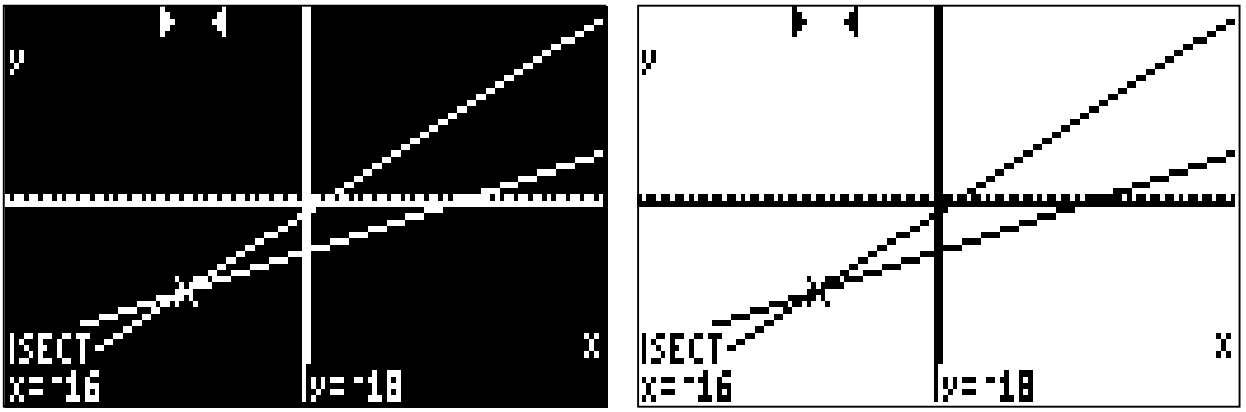


Figure 1: To the left the figure included with default settings, on the right the figure after enabling `settransfer`.

```
/cmr10 /cmr10 def
```

and expects post-processing to find and substitute the fonts. Instead of such post-postprocessing, GEX ignores (i.e., processes and discards the result) this declaration, but requires either explicit loading of needed fonts via the `.loadfont` extension

```
\special{pS: /cmr10 .loadfont}
```

(using one such command for each required font) or by enabling of the autoloading feature via the `.autofontload` extension

```
\special{pS: 1 .autofontload}
```

⟨***string***⟩ **.loadfont** loads the font by its TEX TFM name into the GEX font machinery and makes it available to `findfont` and related operators.

⟨***int***⟩ **.autofontload** If the integer argument is non-zero, GEX will query the TEX font configuration files when the `findfont` operator cannot resolve a font name. The default is *not to load* fonts implicitly and substitute Helvetica.[3]

These commands must be issued before a MetaPost-generated file is actually included.

EPS-specific problems

In the process of testing GEX over many hundreds of "real-life" `.eps` files, some common problems have been discovered. While these are technically bugs in `.eps`, they are common enough so workarounds had to be provided.

The majority of the `.eps` related workarounds (as well as many new options) have been incorporated as new keys to the `\includegraphics` command; this provides for an easy end-user interface[4].

Leaving entries on the PostScript operand stack is surprisingly common misbehavior which we have seen in files generated by many applications. If the `.eps` file is sound otherwise, it will be processed correctly; but an error may occur in handling the PostScript code that comes after.

Because of the common nature of this error (and especially because it causes the error message to point not to the culprit, but some later PostScript code), this GEX error is reported TEX-style:

```
! junk on PostScript stack, 4 items
? h
The PostScript code you just executed
has left some junk on the operand stack.
I'm taking it off; cross your fingers
and pray that this is all to it.
```

The fix required from the user is to add 4 `pops` at the end of the `.eps` image.

Degenerate matrices *Near-degenerate* matrix transforms often cause serious problems with the Acrobat's 16-bit computational limit. One can show that the problem is not solvable correctly in general; and Adobe Acrobat Distiller would fail on degenerate transforms.

The example file

```
% lwid.ps
0 0 moveto
gsave 100 200 lineto 2 3 scale
 1 0 0 setrgbcolor stroke grestore
gsave 200 100 lineto 0.5 0.3 scale
 0 1 0 setrgbcolor stroke grestore
```

[3] One of the subtle differences between GEX and PostScript is substituting Helvetica rather than Courier for fonts that are not available; in the author's opinion Courier is not a font to be used for *any* purpose.

[4] Special thanks to DC for providing the ability to define custom keys in `graphicx`.

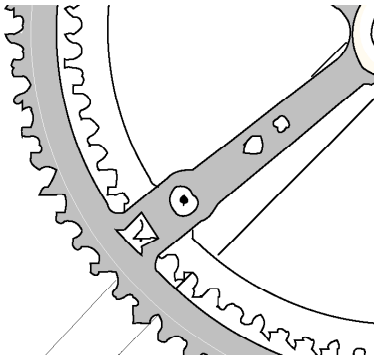


Figure 2: Fragment of a sample `gears.eps` included with `\includegraphics{gears.eps}`

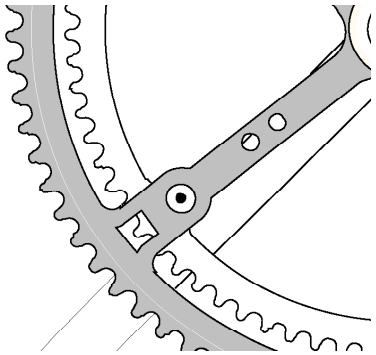
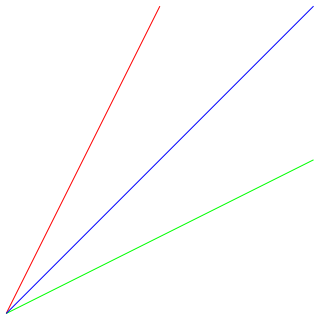


Figure 3: Same as Fig. 2 included with `\includegraphics[innerscale=4]{gears.eps}`

```
gsave 200 200 lineto 0 0 1 setrgbcolor
 [0.18672 -0.565306 0.87384 -2.64563 0 0]
 setmatrix
 stroke grestore
showpage
```

should produce three lines from the origin. Distiller, however, will miss the middle line. GEX, on the other hand, will produce correct output:



Near-degenerate matrices are not a perverted aberration: they tend to be generated by common software, such as CorelDraw. The particular set of numbers in the source above came from a Corel example.

While GEX does the work correctly in most cases (the precision limit in PDF guarantees that no PS↦PDF conversion can work correctly in all cases): some distortion of the line widths is possible and is not avoidable.

Level 1 strokeadjust Some graphics applications (for example, Freehand) output Level 1 PostScript code which fits the coordinates to an integer grid. This code, if executed literally, will produce rather disastrous results with GEX. Figure 2 shows one of the "real-life" examples.

The nature of the problem is a bug (or *feature*) in the Freehand adjustment code which does not bother to check for the device matrix and assumes that it corresponds to the output pixel resolution of 300 dpi or higher (which would imply a device matrix `[4 0 0 4.. ..]`). However, the GEX device matrix is chosen to be an identity, to avoid extra rounding by TEX's ⇔ GEX's coordinate translation. This causes extremely coarse coordinate rounding (72dpi) in the default case.

GEX's workaround to this problem consists of `\specials` that switches the device matrix to an appropriate one; this is incapsulated in the

```
innerscale=
```

option to the `\includegraphics`.

The corrected picture on Fig. 3 was processed with `innerscale=4`.

Level 1 / 2 differences While PostScript Level 2 is supposed to be a superset of Level 1, it is wrong to conclude that PostScript graphics displayed correctly on a Level 1 interpreter will appear the same way (or at all) on a Level 2 interpreter. It is all too common for `.eps` files to actually check the interpreter version and then execute totally different code, depending on the version found.

Both images came from the same PowerPoint-produced `.eps` file. Since in this case the end user might prefer the Level 1 appearance (but in some other `.eps`, perhaps in the same document, Level 2 may be required), GEX provides an ability to switch between Level 1 and Level 2 dynamically. On the lower level, this is done by the

```
N .setlanguagelevel
```

extension operator. Alternatively, the user might prefer to use the `gexlevel` option provided for the `\includegraphics` command and enter

```
\includegraphics[gexlevel=N]{paths.eps}
```

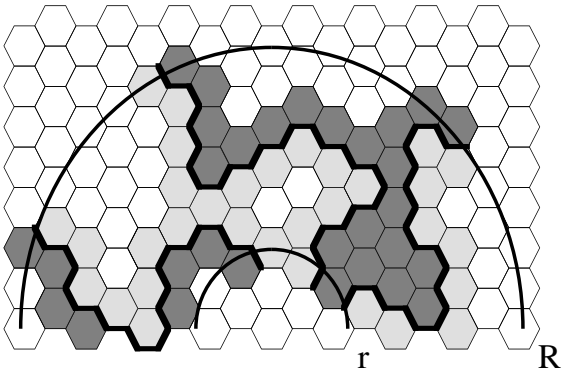


Figure 4: Level 1 appearance

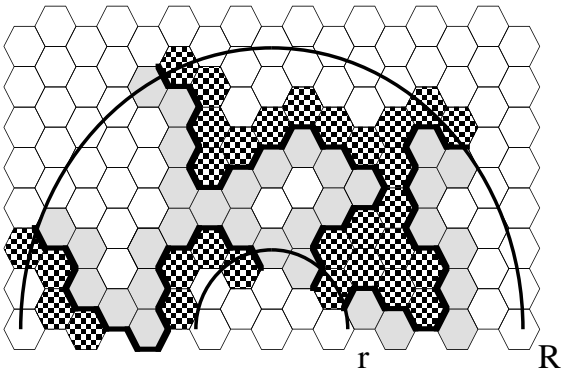


Figure 5: Level 2 appearance

Feedback to TEX

Since both TEX and GEX operate at the same time, it is possible to make them share information. While passing information from TEX to PostScript is trivial and has been done for ages (by putting them inside the PostScript language `\special`, in the case of VTEX, `\special{pS:..}`), getting information back from PostScript is new.

In GEX this is accomplished by PostScript syntax extensions that allow access to TEX `\toks` registers within GEX. The three new operators are:

- ⟨*int*⟩ ⟨*string*⟩ `.tkread` ↦ ⟨*int*⟩ ⟨*string*⟩

  where the ⟨*int*⟩ parameter should be between 0 and 65535 and designate a TEX token register[5]; the ⟨*string*⟩ parameter is the receiving string. In the output, the integer value is the new length of the string; the string contains the contents of the `\toks` register.

  Control sequence tokens are converted to spaces during `.tkread`; they are counted as one character for `.tklength`.

  During `.tkread` a `rangeerror` may occur if the `\toks` register contains more characters than can be placed into the receiving string; one can use the `.tklength` operator to find out how big the receiving string should be before allocating it.

- ⟨*int*⟩ `.tklength` ↦ ⟨*int*⟩

  where the ⟨*int*⟩ parameter should be between 0 and 65535 and designate a TEX token register; the output integer is the length of the contents of the TEX `\toks` register.

- ⟨*boolean*⟩ ⟨*int*⟩ ⟨*string*⟩ `.tkwrite` ↦

  where the ⟨*boolean*⟩ argument determines if the data should be appended to the `\toks` contents (`true`) or overwrite it (`false`); the ⟨*int*⟩ parameter is between 0 and 65535 and designates a TEX token register; the contents of the ⟨*string*⟩ parameter will be globally placed into the specified `\toks` register.

  Token strings produced by `.tkwrite` contain only tokens with TEX `\catcode` 12 (other).

The interface is deliberately kept very general; It is assumed that a TEX macro writer would unpack the `\toks` string as desired.

Here is how one can try to use GEX to generate a few random numbers:

```
\def\rand{%
  \special{pS: false 100 rand
    10 string cvs .tkwrite}%
  \the\toks100
  }

The numbers are = \rand, \rand, \rand.
```

The numbers are = 16807, 282475249, 1622650073.

Immediate execution

While the syntax above provides a way to deliver information from GEX to TEX, the information will arrive too late to be of much use. This is because `\special`s are executed during the page building (`\shipout`), when it is too late to use the returned data. For this reason, the example as written above will actually not work as specified.

While it is possible to overcome the problem with usual TEX multi-pass tools (the `.aux` file), we chose to instead enhance TEX with the

```
\immediate\special{...}
```

[5] Not a typo; the VTEX compiler has larger limits than other versions of TEX and `\toks10000` is legitimate.

form. The semantics here are identical to those when the `\immediate` command as used with the file operations which are already in TEX.

Besides making the example above work, the immediate form of `\special` proves very handy in a number of other cases, for instance:

- setting the background color for a page
- defining a PostScript header file
- thumb generation specials (PDF mode)

These actually cause difficulties for VTEX: a DVI driver can scan a page ahead to see if such `\special`s are present, but the one-pass nature of VTEX compilation requires them to be processed before the `\shipout` gets under way; the immediate form solves exactly this.

Creating objects in immediate mode

The GEX feedback can be used for many purposes, some of which can be accomplished by TEX means (if barely) and some which cannot be. One reason is because PostScript is a better computational language than TEX, and the `\immediate` form of `\special` makes it fully available to TEX. `trig.sty`, for example, is one casualty of this approach.

More interestingly, PostScript is more aware than TEX of the nature of the graphics objects that are in the document. For example, it is possible to use GEX to compute the exact locations of the extremes in a graph and then pass these locations back to TEX for placing of tags.

To allow development of this type of applications, we provide some additional machinery:

- It is allowed to have GEX compile and generate code for graphics objects in the `\immediate` mode; this code, however, is written to a memory stream.
- A memory stream can be frozen and closed with the `\special{ice}` command (naturally, another `\immediate`); when a stream is closed, its handle is provided in the `\pdflaststream` register.
- As graphics are drawn, the placement of the TEX tags can be computed as well, and reported to TEX via `\toks`.
- A stream is placed into the output page with the `\special{!stream ...}` command. Here we do not use the `\immediate` form, since the graphics should be emitted and properly placed during the usual `\shipout`.

Note: Code emitted prior to the `\shipout` cannot go to the output page right away since the formatting of the output page is not yet known. Thus, such code is emitted relative to the $(0, 0)$ origin; during the actual `\shipout` the code is shifted to the position of the `\special{!stream...}` command.

The technique outlined above has been successfully utilized in several new macro packages, including `vfplot`. They, however, use the extensible nature of GEX as well, and it would be prudent to explain this first.

Extending GEX

While in principle PostScript has as much computational power as a conventional programming language, writing computations in PostScript is much more time consuming than in, for example, C or Pascal. (Complex PostScript code may also take a long time to be interpreted.) The `.extend` operator in GEX seeks to add the extra power of conventional programming to GEX. In essence, a user can implement extra computational (or drawing) abilities in a compiled dynamic library (DLL for Windows/OS2, SO for Unix/Linux), then have these abilities available as new PostScript operators.

We call such addon DLLs **GEX plugins**. Syntactically, one writes

```
(pluginname) .extend
```

where `pluginname` refers to the name of a DLL (Windows/OS2) or a shared library (Linux/Unix) which contains the implementation of new extension operators. Upon encountering the above line, GEX will

- look for the requested plugin module
- ensure that its version matches the version of the GEX interpreter
- find out which new operators are implemented in the library, add their names to the PostScript namespace, and record the location of their implementation code.

Upon encountering a new operator, GEX calls the implementation code in the plugin.

GEX API

The GEX Application Programming Interface (API) represents PostScript internal operators as callback functions. For example, where a PostScript program would execute

```
10 20 moveto
```

a GEX plugin written in C shall do

```
GeXi->moveto(10,20);
```

The C/C++ API is specified in the `gexi.h` header file; it generally parallels PostScript drawing operators.

Rather than list the entire API here, we shall outline its principles:

- GEX API functions call the GEX kernel and are generally equivalent to PostScript operators.
- GEX API functions return 0 on success, and the PostScript error number on failure; it is the plugin's responsibility to handle the errors.
- GEX API functions cover most PostScript drawing abilities, but not text output. This is because plugins are not intended to do text formatting; this task should be passed over to TEX.
- The GEX API includes functions for working with the PostScript operand stack.
- An exception to the above is the `show()` function which is provided for the purposes of debugging only.
- Just like GEX itself, plugins can talk to TEX; this is done with the `tkwrite()`, `tkread()`, and `tklength()` functions.

For example, an extension operator `square` can be defined to draw a 10×10 square with C code like this:

```
int square(GEXI GeXi) {
   double x,y;
   if (GeXi->currentpoint(&x,&y)!=0)
      return error_nocurrentpoint;
   GeXi->lineto(x+10,y);
   GeXi->lineto(x+10,y+10);
   GeXi->lineto(x,y+10);
   GeXi->setrgbcolor(1,0,0);
   GeXi->closepath();
   GeXi->fill();
   return 0;  //success
   }
```

This example is, of course, useless: it is so simple that the task can be accomplished much easier in PostScript. However, the benefits of C programming become clear in more complicated cases.

PieChart

The first GEX plugin was the PieChart plugin. The implementation consists of

- `piechart.[dll|so]`, the extension library.
- `piechart.sty`, a matching LATEX 2ε style, to shield the interface details from the end user.

The code below shows how the end user may be using the plugin; a TEXnically minded person might also want to examine the sources which are available together with the GEX API description.

```
%% Define some colors
\definecolor{lightyell}{rgb}{1,1,0.75}
\definecolor{peach}{cmyk}{0,0.50,0.70,0}
\definecolor{orange}{cmyk}{0,0.61,0.87,0}
\definecolor{navyblu}{cmyk}{0.94,0.54,0,0}

\begin{center}
Shares of \TeX\ dialects:\par
\fbox{\begin{PieChart}[rt]{1.8in}
\PieSlice{lightyell}{65}{\LaTeX\ 2e}
\PieSlice{green}{20}{Plain \TeX}
\PieSlice{navyblu}{10}{AmS\TeX}
\PieSlice{yellow}{4}{\LaTeX\ 2.09}
\PieSlice{orange}{1}{Other}
\end{PieChart}}
\end{center}
%%
```

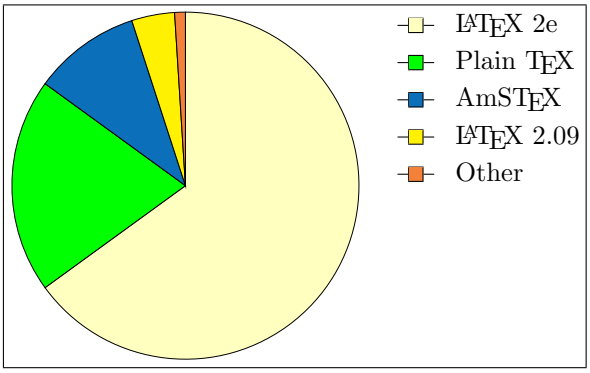


Figure 6: Shares of TEX dialects

A sample PieChart produced by this extension is shown in Figure 6.

Vchart

While PieChart is simple enough that it can be implemented in TEX/inline PostScript, its descendant, Vchart, breaks the barrier.

The Vchart package implements several formats of business graphs. Like PieChart, it is a combination of a plugin and a macro package.

To structure the user input, Vchart provides several environments. One defines the colors:

```
\definecolor{c1}{rgb}{.565,.592,1}
\definecolor{c2}{rgb}{.565,.184,.373}
\definecolor{c3}{rgb}{1,1,.753}
```

the headers:

```
\begin{header}{sides}
\entry[fillcolor=c1]{West}
\entry[fillcolor=c2]{East}
\entry[fillcolor=c3]{South}
\end{header}

\begin{header}{ABCD}
\entry{A}\entry{B}\entry{C}\entry{D}
```

```
\end{header}
```

and the data:

```
\begin{datatable}{example}
20.4 & 27.4 & 90 & 20.4 \\
30.6 & 38.6 & 35.6 & 31.6 \\
45.9 & 46.9 & 45 & 43.9 \\
\end{datatable}
```

and applies the `\DrawGraph` command.

The first graph in the series below has been produced with

```
\colorbox{grbkcolor}{%
  \DrawGraph{graphdata=example,
  graphtype=column,width=100pt,
  height=70pt,rowheader=sides,
  colheader=ABCD}}
```

The other graphs differ only in the `graphtype=` setting. The separation of data from the actual command allows to produce different charts from the same values.

Vfplot

The most powerful GEX plugin designed so far is Vfplot. The name stands for the Visual Function Plot; Vfplot converts functions given as formulas into the plots within the document. Unlike the facilities offered by standard plotting tools (MathCad or MatLab), Vfplot was specifically designed with TEX in mind; the plots it produces are visually compatible with the TEX document (plots use the document fonts and TEX-formatted text).

Like the plugins mentioned above, Vfplot comes with a comprehensive macro package (`vfplot.sty` for LATEX 2_ε and `vfplot.tex` for Plain TEX) which screens the plugin details from the end user.

Samples of inputs to Vfplot and its outputs are shown in figures 8, 9, and 10.

Vfplot and PSfrag

In principle, plots similar to Vfplot's can be also be achieved using a standalone math plotting system (like MatLab) in conjunction with the PSfrag package. The basic advantage of Vfplot is that the plot is an integral part of the TEX document; it can be changed by changing the plot code within a TEX file directly, or employing the Visual plot editor (see below). PSfrag, on the other hand, is essentially a write-once format, which requires a separate program for making the plot and additional manual work in setting the substitution tags.

However, PSfrag also has an advantage of being more portable; Vfplot (and plugins in general) are VTEX-specific.

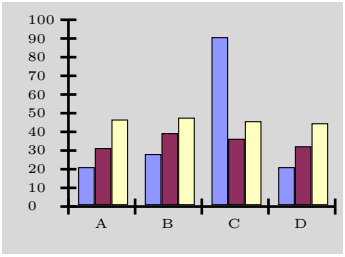
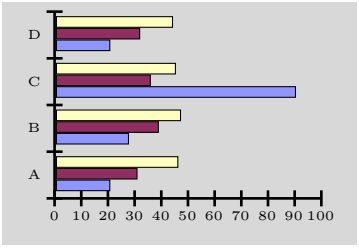
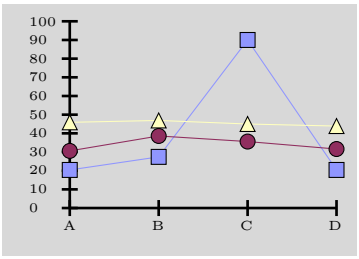
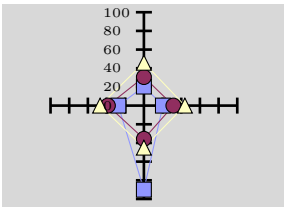
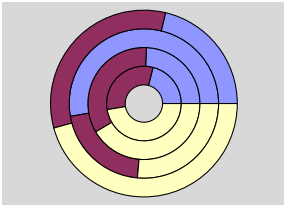
Graph Type/`type`	**Result**
Column graph/ `graphtype=column`	
Bar graph/ `graphtype=bar`	
dots and lines/ `graphtype=dots`	
Radar graph type/ `graphtype=radar`	
Doughnut graph type `graphtype=doughnut`	

Figure 7: Sample Vchart output. Notice that Vchart does not have a `piechart` type of graph; but a pie, after all, is just a degenerate doughnut.

```
\begin{plot}[legend=rt]{x-axis=MyAxis1,y-axis=MyAxis2,plotfill=CoorFill0}
 \function[linetype=MyLine1]
    [minlimit=-3.14,maxlimit=3.14,level=0,hatching=FunHatch,fill=FunFill]
    {x/2+sin(x)+cos(x^2) | x in [-5,5]}{$+{x\over 2}+\sin x+\cos x^2$}
\end{plot}
```

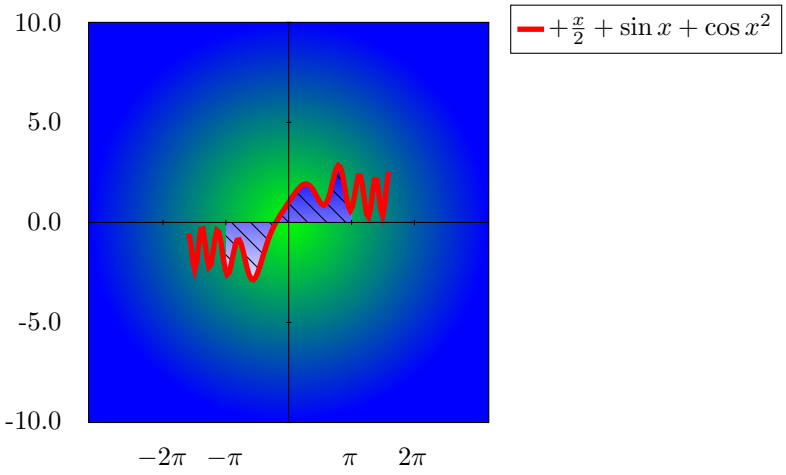


Figure 9: Vfplot drawing: Color Map

```
\begin{plot}{
x-axis=SineXAxis, % use a predefined axis
y-axis=SineYAxis,
gapsfill=Sunset2  % use a predefined
                  % gradient stretch
}
\function[linetype=MyLine]{sin(t)}{}
\end{plot}
```

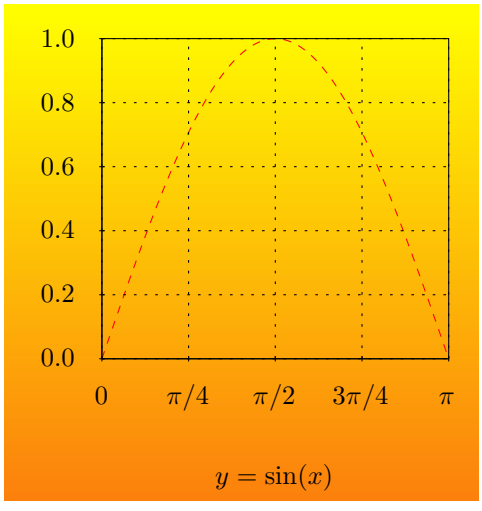


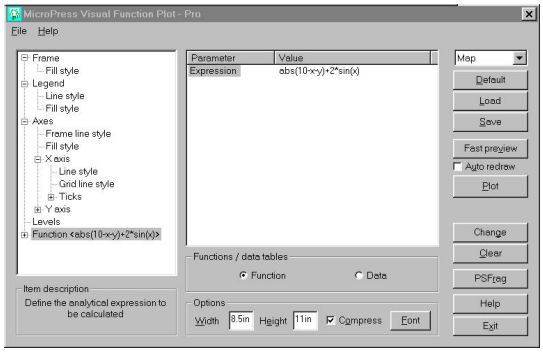
Figure 8: Vfplot drawing: 1D Plot

We, therefore, support exporting Vfplot environments into `.eps`/`.inc` file pairs; the `.eps` file contains the plot itself, while an `.inc` file contains a PSfrag wrapper for it.

Vfplot Visual Frontend

While it entirely possible to create Vfplot input "by-hand", the number of possible parameters is so large that a Visual frontend becomes useful. Such a frontend currently exists under Windows only; its functionality includes abilities to

- edit `plot` environments within TEX documents.
- visually manipulate all possible options of such environments:



- instant preview of the plot:

```
\begin{plot3d}{x-axis=AxX,y-axis=AxY,
   z-axis=AxZ,isolines=false}
\function[x-numpoints=60,y-numpoints=60,
   uppersidefill=MyFill3,
   lowersidefill=MyFill2,lineoff]
 { (1-x)^2+100*(y-x*x)^2 | x in [-1.5,1.5];
    y in [-0.5,1.5]}
\end{plot3d}
```

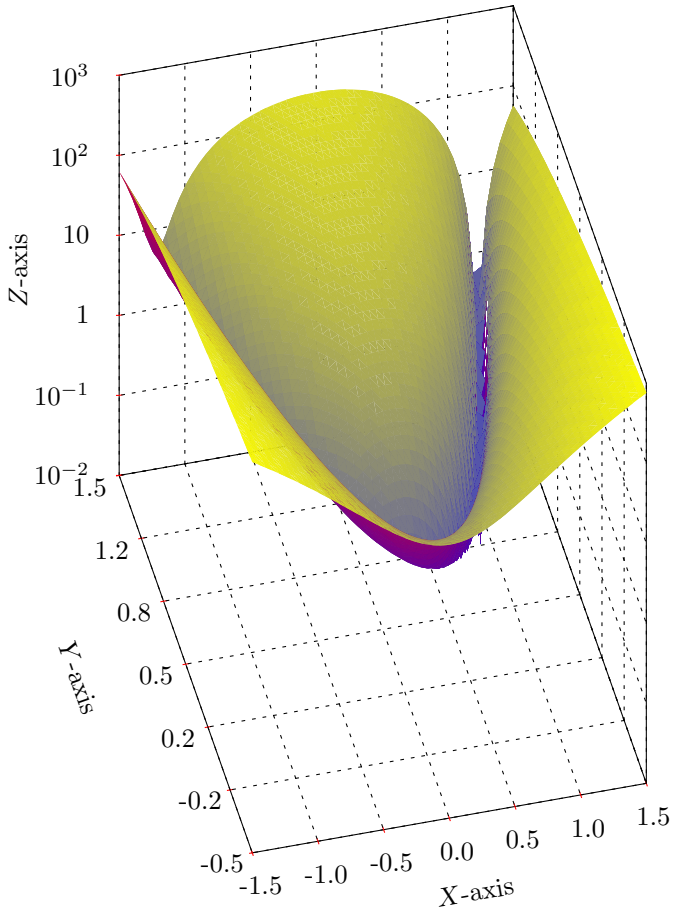
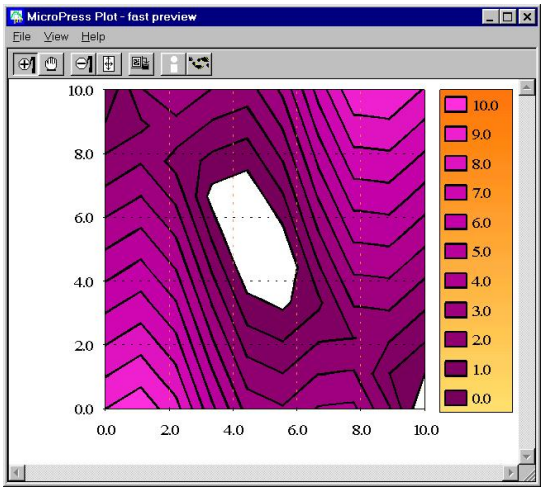


Figure 10: VFPlot drawing: 3D Plot. The Rosenbrock function, $z = (1-x)^2 + 100(y-x^2)^2$



as well as somewhat slower full plot. (The instant preview does not expand TEX notation).

- export plots in many bitmapped formats.
- export plots as `.eps`, or `.eps` together with a PSfrag header (`.inc`).

Image processing

One important place where GEX differs from ordinary PostScript is the image handling. In "normal" PostScript, `image` is a primitive operator. By default, it is the case in GEX as well; however, `image` is implemented as a combination of two new extension operators:

.loadimage does the unpacking and produces an *image* object on the operand stack.

.produceimage emits the *image* object to the output stream.

Thus, one can define

```
/image {.loadimage .produceimage} def
```

without changing the way `image` operates. Similarly, `imagemask` and `colorimage` use

.loadimagemask for the unpacking of the `imagemask` data

.loadcolorimage for the unpacking of the `colorimage` data

and are internally defined as

```
/imagemask{ .loadimagemask
  .produceimage} def
/colorimage{  .loadcolorimage
  .produceimage} def
```

By itself, this adds nothing. However, it opens a door for inserting a new operator between the two components of `image`:

```
/image {.loadimage myfilter
  .produceimage} def
```

Such an operator can manipulate the image data in memory.

A filtering operator as defined above cannot be written in PostScript — there is no *image* data type in the PostScript language; and from the point of view of the PostScript processor, *image* is just an `int`. A curious user can see this by trying

```
/image {.loadimage pstack
  .produceimage} def
```

However, image filters can be easily implemented via plugins.

Two plugins have been developed to perform image manipulations:

TransBit can alter the color model of the image. One of the applications is to convert color (RGB or CMYK) images to grayscale for printing purposes. TransBit functionality also covers the brightness and the contrast of the image.

Degrade downsamples the image; this can be used to decrease (often, greatly) the size of the resulting output file.

Both plugins take additional parameters. For example, if we want to brighten an image by 10 units, we would issue

```
\special{pS: (transbit) .extend}
\special{pS: save}
\special{pS: /image {.loadimage
  (toBright 10) transbit
  .produceimage } def
\includegraphics{mypic.eps}
\special{pS: restore}
```

The `save`/`restore` pair is needed to restore the original definition of `image`.

The functionality of both plugins has been incorporated in the `\includegraphics` command, so the end user will merely write

```
\includegraphics[brightness=10]{mypic.eps}
```

Note: The `graphicx` package automatically loads the required plugins upon seeing keys that are implemented in plugins.

Non-PostScript images

Although the above functionality applies to images stored within PostScript code (like the ones produced by `jpeg2ps`), we can easily extend it to the bitmapped image files.

The idea here is to be able to load image files into the GEX/PostScript environment; this is done with the `.readimage` extension operator. This operator takes a string argument with the image file name and converts it to an *image* object on the PostScript operand stack; such an object can be followed

up by a `.produceimage` or by imaging filter(s), and then by a `.produceimage`.

The end-user interface is again trivial. For example:

```
\includegraphics[colorspace=grayscale 16]
  {picture.gif}
```

will load the `picture.gif` file into GEX, and convert it to a 16-color (4-bits) grayscale using the TransBit plugin.

TransBit example

The examples in this and subsequent sections show the same image, `macaw.jpg`, processed with different `\includegraphics` keys. The original picture appears in the middle of the first example. TransBit related keys are `brightness`, `contrast`, and `colorspace`; these keys force the image processing via `.readimage`, followed by a plugin application.

Sample code

```
\includegraphics[width=1.3in,
  contrast=-0.3]{macaw.jpg}
\includegraphics[width=1.3in,
  contrast=0]{macaw.jpg}
\includegraphics[width=1.3in,
  contrast=+0.3]{macaw.jpg}
```

results in

`contrast=-0.3`

`contrast=0`

`contrast=+0.3`

Color model conversion would be, in particular, of use when the document is to be eventually printed on paper. For example, type

```
\includegraphics[width=1.3in,
   colorspace=bw]{macaw.jpg}
\includegraphics[width=1.3in,
   colorspace=grayscale 16]{macaw.jpg}
\includegraphics[width=1.3in,
   colorspace=grayscale 256]{macaw.jpg}
```

to produce

`colorspace=bw`

`colorspace = grayscale 16`

`colorspace = grayscale 256`

Color space conversion to grayscale also often substantially reduces the size of the output.

Degrade example

The Degrade plugin is triggered by the `degrade` key of `\includegraphics`; `degrade=1` corresponds to no downsampling.

```
\includegraphics[width=1.3in]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.6]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.4]{macaw.jpg}

\includegraphics[width=1.3in,
  degrade=0.3]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.2]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.1]{macaw.jpg}
```

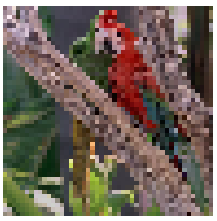
degrade=0.6

degrade=0.4

degrade=0.3

degrade=0.2

degrade=0.1



The transformations given above usually result in a drastic decrease of the size of the output. A minimal TEX source file which consists of a solo `\includegraphics` with different `degrade=` coefficients will result in PDF files of decreasing sizes:

Coeff.	File Size	
	Uncompressed	Flate-Compressed
1.0	1,278,158	1,029,377
0.6	459,780	392,863
0.4	204,804	181,256
0.3	115,908	104,714
0.2	51,970	47,932
0.1	13,952	13,242

Note: Downsampling can also be accomplished by means of the Discrete Cosine Transform; this is triggered by the `dct` and `dctquality` keys for `\includegraphics`. For photo-quality images this often leads to better results.

Color stack issues

The `color` package offers two distinct ways to maintain color: rely on the color stack in the backend (usually, a DVI driver), or — when such a stack is not available — emulate it within TEX.

As turns out, with GEX neither approach is fully adequate. The color stack within TEX is generally incapable of preventing color leaks from one page to another; but full use of the backend color stack is not possible since GEX already implements the full PostScript graphics state stack (GSS). While the GSS saves colors, it also saves the current point. This breaks some of the PStricks sub-packages, such as `pst-text` or `pst-path`.

The workaround used in GEX is to support both TEX and backend color stack approaches:

- `vtex.def` provides a macro `\if@colorstack`; when `true`, the `color` style uses the GSS stack; when `false`, `color` emulates the color stack with TEX means.
- By default, `\if@colorstack` is true; the driver color stack is used.
- Environments like `pspicture` are redefined to set `\if@colorstack` to `false`; this assures that PStricks are not broken.

Credits & Acknowledgements

The VTEX/GEX system itself was written by Michael Vulis. Most of the supporting macro packages were written by Alex Kostin. Vchart was written by Kirill Lebedev. Other plugins quoted in the article have their respective authors.

The authors wish to express thanks to

- Walter Schmidt and Taco Hoekwater for extraordinary efforts in making the freeware versions of VTEX possible.
- Denis Girou and Timothy van Zandt for cooperation and help in cleaning bugs in PStricks and Seminar which made their use with GEX possible.
- David Carlisle for providing an extendable version of the Graphics package which makes a natural interface to GEX features possible.
- Heiko Oberdiek for outstanding efforts in making sure that Hyperref manages all the multiple modes of VTEX.
- Many end users who discovered and reported bugs in GEX — thank you all — and please send more.