Introducing GeX

Taco Hoekwater,
Bittext VOF
Michael Vulis,
The City College of New York & MicroPress, Inc.

abstract

This is a short introduction to the pilot release of GeX. GeX is the most rapidly evolving part of VTEX; more detailed documentation is available in the distribution of VTEX. This article specifically describes GeX as implemented in the public domain version of VTEX/Linux. While the same or additional features may be available in the commercial Windows version, we describe what exists in the freely downloadable version. For information on downloading VTEX/Linux see the NTG web site or the article in this MAPS issue.

keywords

pdf, inline graphics, eps inclusion, GeX, VTeX

Why GeX?

VTEX's PDF backend includes an integrated PostScript-compatible processor (GeX).

GeX makes it possible to do easy one-pass handling of:

- Encapsulated PostScript files (`.eps`)
- PStricks, PSFrag, and Seminar code
- Other inline PostScript code with optional feed-back of information from GeX to the TEX processor

While `.eps` inclusion has been previously supported in VTEX via GhostScript library calls, GeX offers much better performance and output quality.

The `.eps` inclusion is likely to be the main *initial* application of GeX. However, in our view it is the inline PostScript which could lead to new and interesting applications.

Why call it GeX?

The GeX name [pronounced `g-e-k-s`] stands for Graphics EXtensions.

While the current extensions are generally compatible with the PostScript language, GeX is intended to be a TEX-resident extension, not an Acrobat clone. Even in the current implementation there are facilities for communication between TEX and PostScript, as well as appoximately a dozen of new operators; these facilities are likely to be further developed. While it is our intention to stay PostScript-compatible to the degree needed for `.eps` and inline PostScript support, we envision further enhancing GeX with features that are decisively non-PostScript.

Enabling GeX

The current implementation does not enable GeX by default. This is because the initialization of the PostScript machinery takes 1-2 seconds and GeX is not needed for documents that contain only text and bitmapped images.

To enable GeX, use the `"-ox"` switch on the VTEX command line.

Notice that GeX works only in the PDF backend mode; in other modes the `"-ox"` switch has no effect.

Syntax

Supported PostScript operators

GeX currently supports a large subset of PostScript, including most of Level I and a few Level II operators. A full list of supported operators appears in the GeX reference documentation which is part of the distribution.

Since not the PostScript operator set is supported in its entirety, it is possible (and even easy) to write a valid PostScript code which will be rejected by GeX; on the other hand, the supported subset includes all the "practically" useful operators, so GeX would handle correctly essentially every `.eps` that appears in real life. Testing of GeX on a large random set of `.eps` files downloaded from the Internet shows that GeX correctly handles more than 99% of them.

Supported additional operators

GeX also understands a number of extra operators, of which the most important ones are listed below.

⟨**int**⟩ **.autofontload** If the integer argument is non-zero, GeX will query the `type1.rc` file when the `findfont` operator cannot resolve a font name. The default is *not to load* fonts implicitly and substitute Helvetica. This operator is useful for processing MetaPost-generated code; see the explanation at the end of this article.

⟨**string**⟩ **.loadfont** Loads a Type 1 font into the interpreter. The argument should be a string containing a font name. Only fonts listed in `type1.rc` can be loaded.

⟨**int**⟩ **.setdigits** This command sets the number of emitted fractional digits in the generated PDF output to an integer argument. The default value is two, which is the best value for most applications and devices. Only high-end applications may benefit from a larger value and only non-printable web-only files can do with a lower value.

.extend Will be explained below, in the section about extending GeX.

⟨**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-dependant and device-independant color manipulations. The first usage is more common and is essentially for minor color adjustments. In such situations the best strategy for producing device-independant `.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-independant adjustments. An example of such an adjustment would be the invertion of 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 process this inversion. In the case of Distiller, the override is a global Job option which will apply to all parts of a document; GeX allows one to override the handling of only an individual image. This is accomplished with the extention operator `.enabletransfer`. With an argument of zero, `.enabletransfer` disables processing of the `settransfer` code; a non-zero argument enables `settransfer` processing. Figure 1 is an example of a small `.eps` file that uses transfer code.

.tkwrite .tkread .tklength These operators are explained below.

Using GeX

If you are going to use GeX only for inclusion of ready-made `.eps` files, you should use a high-level package like `graphics` rather than VTEX's `\special`'s and disregard the rest of this section and paper. The same applies if you intend to use GeX with supported inline PS packages like PStricks, PSfrag or Seminar. In all these cases, the configuration files tuned up for GeX are supplied and knowledge of the low-level details in unneeded.

However, these details provide the framework for additional power to be realized in future packages for mixing text and graphics.

The GeX engine is invoked from VTEX with the following two `\special`'s:

- `\special{ps: ...}` is used to pass a file to GeX
- `\special{pS: ...}` is used to pass commands to GeX

With GeX enabled, VTEX allows you to precede a `\special` with the `\immediate` command. `\immediate \special`'s are passed to GeX right away, while TEX is still doing formatting.

The PDF code generated in *immediate* mode can be re-used, see the section below on re-using pdf code. Of course the immediate mode can also be used to do calculations.

TEX-GeX interface

The communication interface between TEX and GeX consists of three additional operators:

- `.tkread` to read the contents of a TEX `\toks` register
- `.tkwrite` to write to a TEX `\toks` register
- `.tklength` to find out the length of a TEX `\toks` register

The syntax of these operators is as follows:

- ⟨int⟩ ⟨string⟩ `.tkread` ⇒ ⟨int⟩ ⟨string⟩
  where the ⟨int⟩ parameter should be in the range 0 through 255 and designate a TEX token register; 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.
- ⟨int⟩ `.tklength` ⇒ ⟨int⟩
  where the ⟨int⟩ parameter should be in the range 0 through 255 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 defermines if the data should be appended to the `\toks` contents (`true`) or overwrite it (`false`); the ⟨int⟩ parameter should be in the range 0 through 255 and designate a TEX token register; the contents of the ⟨string⟩ parameter will be placed into the specified `\toks` register.

Note: 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` opera-

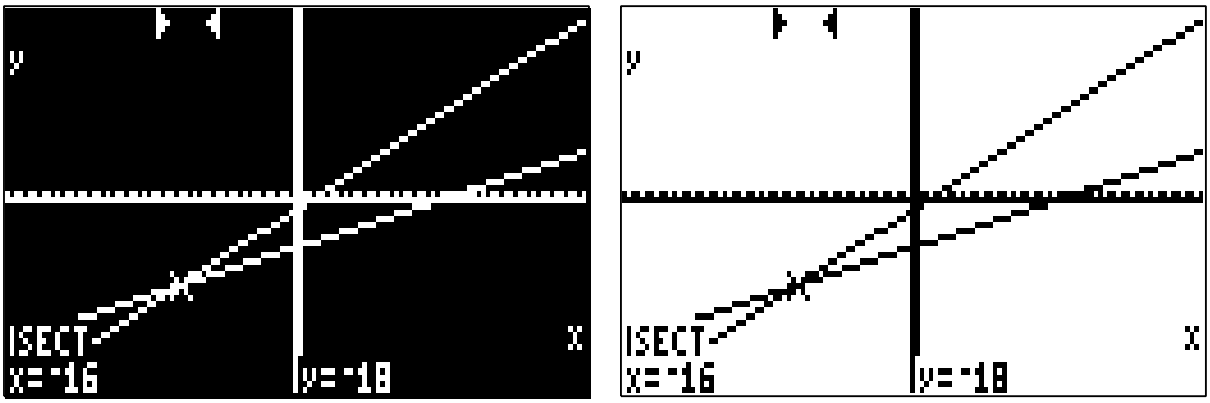


Figure 1. On the left the figure included with default settings. On the right the figure as it would appear after enabling `settransfer`.

tor to find out how big the receiving string should be before allocating it.

Note: Control sequence tokens withing TEX token strings are converted into spaces during `.tkread`; they are counted as single characters in `.tklength`.

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

Re-using pdf code

VTEX allows you to re-use the pdf code that is generated by the `\immediate` form of the `\special{pS:...}` operator.

During the `\immediate` output, the pdf code is written to a temporary stream. Any `\immediate\special{pS:...}` operator opens such a stream (unless it is already opened by another operator); the currently opened stream, if it exists, is destroyed at the moment of `shipout`. Thus, by default, everything written to immediate streams is lost.

To preserve the contents of an `immediate` stream, use the `\special{ice}` command. This command closes the immediate stream; the new TEX count register `\pdflaststream` can be used to retrieve the handle to the closed stream that was just closed. The command

```
\special{!stream \the\pdflaststream}
```

can be used to re-insert the frozen stream into the TEX machinery; it will be emitted during the normal shipout.

Notice that the `\special{ice}` command must be issued in the `\immediate` mode (otherwise, there will be no stream to freeze by the time it gets processed); on the other hand, `\special{!stream ...}` must be deterred till the `\shipout`.

If you generate pdf code during the `\immediate` mode, you should realize that the positioning of your code will not be known until the time of the `\shipout`. Thus, the PostScript `currentpoint` is not really defined. The way to overcome this problem is to initialize the current point to $(0,0)$ by executing `0 0 moveto` at the beginning of the `\immediate` stream.

The data inserted in the output during the `\special{!stream...}` processing is offset by the `currentpoint` as computed during the `\shipout`.

Extending GeX

The major new feature added in VTEX 6.3 is GeXX. The second "X" stands for eXtensible. With GeXX you can supplement the existing set of operators with new constructs, implemented in C or C++.

In the Linux version, the extension libraries are standard shared objects (.so), implemented with gcc/g++. While the GeX API seems C++ at first sight, it is actually standard C. The GeX API is portable, so the same extension can be compiled for both Linux and Windows (under Windows it would become a .dll). At this moment the only supported compiler under Windows is BC, but it is likely that other compilers and languages can be used on both platforms.

The libraries are not referenced from the executable, so you can create new ones as you desire; a single library can implement multiple extentions, and multiple libraries can be loaded in the same job.

To enrich GeX with new operators, you should

- Implement them within a C-language library
- Call the `.extend` operator to load the new language extensions
- Provide additional TEX and/or PostScript code for easy access to the new operators

Note: To be visible to the compiler, the extension library should be placed into the `vtex/bin/gex` subdirectory.

An extention library is only required to export three functions:

1. A function that returns the version of the interface as defined in `gexi.h` (If the version returned by the extention library does not match the version needed for the VTEX compiler, the library is not loaded)
2. A function that returns the number of extensions that are implemented
3. A function that returns the PostScript names of the new extensions and a pointer to the function that should be called when a specific extension is encountered

If you decide to make your extentions publicly available, you should make sure that it is clear for which version of VTEX the extention is designed. This means either supplying the extention in the source form (recommended), or at least mentioning the version number in a readme file. Public distributions of VTEX/Linux will be glad to host your extentions.

Writing an extention operator

An extension operator should be declared as an `int` function; its solo argument is the GeX interface structure, `GEXI`.

The majority of the methods provided by `GEXI` correspond one-to-one to either PostScript operators with the same names, or GeX extention operators (`.tkread`, for example). The only exceptions to this at the present time are the methods that deal with the PostScript operand stack; these methods are used to retrieve (and, later, pop) the arguments provided on the operand stack.

Loading an extension library

To load an extension library, execute the `.extend` operator.

The syntax is: ⟨string⟩ `.extend` ⇒ ⟨int⟩

where the ⟨string⟩ argument contains the name of the library file with the language extentions (without the file extension) and the returned ⟨int⟩ is the number of extention operators loaded.

The `.extend` operator will fail with an error if:

- the specified DLL cannot be found
- the specified DLL does not export all three required functions (`Count()`, `Version()`, `Names()`)
- the version returned by the DLL does not match the version of the VTEX compiler

In all three cases, `.extend` will cause a PostScript `fileerror`.

PieChart

PieChart is a real-life example of using GeXX, which implements MS Word-like PieCharts in TEX. The implementation consists of

- `piechart.[dll|so]`, the extension library.
- `piechart.Sty`, a LATEX2E style for using PieChart.

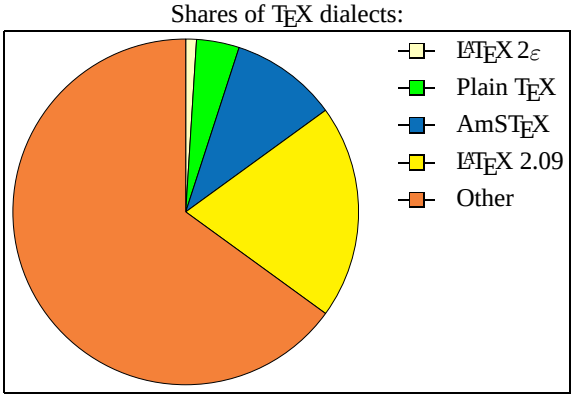
Here is some sample code using PieChart:

```
%% 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{navyblue}{cmyk}{0.94,0.54,0,0}

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

and a sample PieChart produced by this extension:

Shares of TEX dialects:



The PieChart package has been written by Alex Kostin at MicroPress.

Bugs

Being a pilot implementation with source of about 15000 lines of code, GeX undoubtly has many bugs. More than a hundred of them were fixed since the happy moment in July when we thought it more-or-less worked (and were proven wrong on testing of huge set of *real-life* `.eps`'s from diverse sources).

In the aggravation of fixing what we thought was a working program, we discovered that the bugs came in three flavors:

- Our bugs
- Peculiarities (often undocumented) of the PostScript language
- Bugs (or problems) in Adobe Software

Bugs of our implementation (important for us for sentimental reasons) are not worth discussing here; but some of the other bugs are definitly worthwhile mentioning.

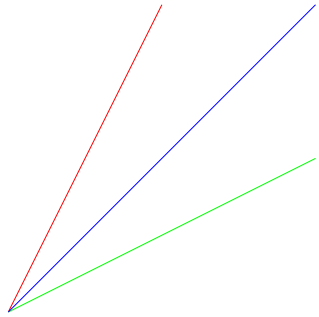
Degenerate matrices

Near-degenerate matrix transforms cause a serious problem with the Acrobat's 16-bit computational limit. It can be shown that the problem is not solvable correctly in general; and Adobe Acrobat Distiller fails 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
gsave 200 200 lineto 0 0 1 setrgbcolor
   [0.186718 -0.565306 0.873838 -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 abberation: they tend to be generated by some common software, especially CorelDraw. The particular set of numbers in the source above came from a Corel example.

While GeX does the work correctly in all cases, some distortion in the line widths is possible and is not avoidable.

Level 1 strokeadjust

Some graphics programs (Freehand is one) output Level I PostScript code which fits the coordinates to an integer grid. This code, if executed literally, will produce rather disasterous results with GeX.

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.

An example of this effect is provided in the VTEX/Linux distribution.

Font name collision bug

There seems to be a bug in many versions of Acrobat which results in (different) fonts with names starting with `|------...` being treated as a single font. To avoid this problem, we replace such names with `|xxxxxx....`

Encoding bug

Under Windows, the Acrobat Reader seems to ignore the `/StandardEncoding` specification and uses the `WinAnsiEncoding` instead. This may lead to incorrect character substitution for some codes in the 2nd half of the ASCII set.

To overcome this problem, VTEX always includes the encoding vector, even if the font is not reencoded.

Dirty Tricks and examples

show redefinition

In order to accomodate packages such as PStricks and PSfrag, VTEX keeps track of redefinition of the `show` PostScript primitive within the GeX engine. In addition to supporting the mentioned packages, this allows rather nice font effects to be implemented with very simple inline code.

Simple outline The examples below were produced with

```
\def\outl#1{\special{pS: save /show{false
   charpath stroke}def}#1\special{pS: restore}}
```

This is a test.

Wider outline with color The macro

```
\def\outla#1{\special{pS: save /show{false
```

```
  3 setlinewidth 1 0 0 setrgbcolor charpath
  stroke}def} #1\special{pS: restore}}
```

produces

Filled letter with outline

```
\def\outlb#1{\special{pS: save /show{false
 charpath gsave 2 setlinewidth 1 0 0 setrgbcolor
 stroke grestore 0 1 0 setrgbcolor fill}def}
 #1\special{pS: restore}}
```

produces

Charpath shown

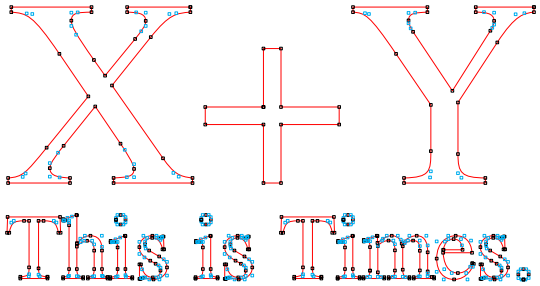
We can also get inside the character representation (something which PostScript would not do on Type 1 fonts):

```
\def\outlc#1{\special{pS: save

  /rct{
   newpath 0.5 add exch 0.5 add exch moveto
   currentpoint exch -1 add exch lineto
   currentpoint -1 add lineto
   currentpoint exch 1 add exch lineto
   currentpoint 1 add lineto stroke} def

  /show{
  false charpath gsave
  0 setlinewidth 1 0 0 setrgbcolor
  stroke grestore
  0 setlinewidth 0 0 0 setrgbcolor
  {rct}
  {rct}
  {rct 1 0 0 0 setcmykcolor
  rct rct 0 0 0 setrgbcolor}
  {} pathforall }def}
  #1\special{pS: restore}}
```

to obtain:



Fragment repositioning

Several examples in PSTricks use the PostScript commands to move the text around in order to land it in an appropriate place on a drawing.

VTEX keeps track of PostScript attempts to group the TEX output; when such activity is detected, VTEX generates PostScript code rather than PDF and feeds this code into the GeX engine.

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 in EPS, MetaPost merely includes declarations like:

```
/cmr10 /cmr10 def
```

and expects post-processing to find and substitute the fonts. Instead of such post-postprocessing, GeX ignores (processes, which is the same really) this declaration, but requires either explicit loading of needed fonts via the `.loadfont` extension:

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

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

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

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

Acknowledgements

The authors wish to express thanks to:

- Alex Kostin for extremely heavy testing of preliminary versions of GeX and finding a few dozen glitches.
- Denis Girou and Timothy van Zandt for cooperation and help in cleaning bugs in PStricks and Seminar which made their use with GeX possible.