Fast Text Rendering
Adobe Developer Support

Technical Note #5136

April 13, 1993

Adobe Systems Incorporated
Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/
Contents

Fast Text Rendering 5

1 Introduction 5

2 Current Text Rendering Methods 7

3 Fast Text Rendering 8
   show String Length 8
   Computation Tasks for Text Rendering 9

4 Optimization Case Study 10
   Using show Operator Variants 10
   PostScript Language Procedures for Vertical Text Rendering 13
   Level 1 show Operator Variants 14
   QuickDraw Strategy 14
   “Space Hack” 14

5 Conclusion 15

Index 17
Fast Text Rendering

1 Introduction

Font rasterizing in the previous generation of PostScript™ output devices was the major bottleneck for printing text. Besides Roman text, this was also true for text rendering of non-Roman text, such as text in the Japanese language. Now, a new generation of PostScript Level 2 output devices with RISC processors are shipping in the market. These new output devices are often shipped with the Type 1 Coprocessor (T1C), Adobe Systems’ new font accelerator technology, which renders characters much faster.

The total amount of time needed for execution of a PostScript language program is divided into the time needed to interpret the program and the time necessary to rasterize the resulting graphics. As a result of the hardware improvements in font rasterizing, the efficiency of the PostScript language code has become a significantly more important factor in printing performance. In fact, inefficient PostScript language code can significantly degrade printing throughput. A key to getting good performance in rendering text is to properly use the show variants available in PostScript Level 1 and Level 2 language. The new hardware environment challenges software developers to improve the way they generate efficient PostScript language code.

Certain actions within application software determine the behavior and performance of the resulting PostScript language code. When the same PostScript printer driver, the same printer with the same controller and engine, and the same text to be rendered are used in performance testing, the results of the PostScript language programs vary. We have analyzed PostScript language programs that have been generated by some application software and PostScript printer drivers, including Adobe™ Printer Driver, Macintosh® version 8.0 (PSPrinter). Many of these programs were found to be inefficient in text rendering.

Figure 1 shows how PostScript language programs generated from a document of an application program performed text rendering operations with different font renderers. Adobe Type Manager™ (ATM™) is software, T1C is the hardware co-processor and RAM cache is the second pass of the rendering operation. The dark bars indicate the performance of the original Post-
Script language program with one-character shows. The shaded bars indicate the performance of the modified PostScript language code with multiple-character shows.

Figure 1 Renderers and show String Variation with Application A

![Application Evaluation Example](image)

Note that regardless of the renderer used to render the text, the absolute time differences between three PostScript language programs are constant. When ATM is used, the difference between the performance of the program generated by PSPrinter and that of edited EPS (Encapsulated PostScript) language file with longer show strings may not seem significant. However, when a faster renderer is used, the difference relative to the overall performance time grows. As the graph indicates, when the RAM cache is used, the performance difference is as large as half the entire execution time. Evaluations with other application software resulted in similar results, and the performance differences stated above indicated the same tendency.

What these evaluation results should indicate to application software developers is that the faster the renderers become, the greater the significance of the quality of PostScript language program generated by application software and PostScript printer drivers.

This document describes how simple optimizations of PostScript language code can enable application software to take advantage of the new text printing environment and improve their performance. Although all the information in this document is true for Roman text printing, developers who publish software for the global market where multi-directional text is common may...
find this information especially useful. The descriptions about fast text rendering and the PostScript language examples in this document are platform-independent; however, the recommendations are specifically tailored for Macintosh application software. Since the importance of efficient PostScript language programming is more apparent in non-Roman text handling, Japanese text is used in the samples used in this document.

2 Current Text Rendering Methods

Many PostScript language programs were generated on the Macintosh from various combinations of application software and PostScript language printer drivers, Apple® LaserWriter® and Adobe PSPrinter. The performance of these programs was evaluated and alternative text rendering methods were recommended to the software developers. The characteristics observed in the evaluation are often typical among the application software capable of handling non-Roman, multi-byte encoded glyphs.

In the tests and analyses of these PostScript language programs, the following text rendering techniques were observed:

- Much of the computation for glyph placement is not performed on the host, but the computational tasks are delegated to PostScript language programs. Such tasks are often performed between moveto/show pair.

- Of those evaluated, a large number of application programs use the show operator only. Few use ashow and awidthshow.

- Almost all application software use show passing a one-character long character string as the primary means of text rendering. If each character placement corresponds to one call of the show operator, rendering one line of text can be a slow operation.

- The majority of application software position glyphs, usually one by one, with repeated pairs of moveto (or rmoveto) and show. This is true even when the text is aligned to the same coordinates along the writing direction.

- No applications use the PostScript Level 2 show operator variants, even if the checking of /languagelevel is performed.

Optimizing each of the above methods can improve the performance of PostScript language programs. Section 4, “Optimization Case Study” shows examples showing comparisons between current text rendering methods and optimized PostScript language code.
3 Fast Text Rendering

Familiarity with the behavior of the show operator variants, PostScript printer drivers, and availability of data associated with rendering of text in documents present optimization opportunities.

3.1 show String Length

Based on the evaluation of PostScript language programs, certain show operator variants’ behavior can be observed. For a given amount of text, it is more efficient when fewer show variants are called, because the overhead incurred with multiple calls to the show variant operators is smaller when a single call with a long string is used. Therefore, more text should be passed to the show variant operators to obtain better rendering performance.

The graph in Figure 2 was created with data obtained in our program evaluation. The x-axis is the number of characters per show string. The numbers on the y-axis represent the number of characters rasterized per second. Only roman characters are used in the evaluation. There is a correlation between the number of characters passed to a show operator variant call and the rate of performance improvement. The major factor contributing to this improvement is the overhead incurred by each call to one of the show operator variants.
3.2 Computation Tasks for Text Rendering

Application software, especially the ones which produce their own PostScript language code, tend to include computational tasks within their PostScript language output.

The overhead caused by this method is two-fold: the number of PostScript language procedures that were included in the resulting file increases and the overall execution time to execute these procedures increases. These computational tasks belong in the application software on the host. Computations for precise character positions in the rendering space should be performed by applications on the host. Their results should then be reflected in the PostScript language program that describes the document.
4 Optimization Case Study

4.1 Using show Operator Variants

There already are a number of PostScript Level 2 output devices shipping worldwide, and many will follow. The parameters passed to the xshow and yshow operators are usually available from data in application software and are suitable for efficient text string and font operations. The following are simple examples to show how a text line can be rendered using show operator variants.

Example 1 shows rendering of a Japanese text line aligned to the left. The code renders each Japanese glyphs on a character-by-character basis. Character spacing uses the default character width.

**Example 1: Horizontal Text Rendering**

```
40 100 moveto
<82b1>show
<82ea>show
<82cd>show
<93fa>show
<967b>show
<8cea>show
<8365>show
<834c>show
<8358>show
<834c>show
<8367>show
<82cc>show
<89a1>show
<8f91>show
```

The first and the most important step in optimizing the code is to make show strings longer. The code shown in Example 2 is much more efficient than the one in Example 1.

**Example 2: Horizontal Text Rendering**

```
40 100 moveto
<82b1 82ea 82cd 93fa 967b 8cea 8365 834c 8358 8367 82cc 89a1 8f91> show
```

When the same text is justified, the generated PostScript language code might appear as in Example 3. The code now contains moveto for the extra character spacing required by the justified text.
Example 3: \textit{Horizontally Justified Text Rendering}

\begin{verbatim}
40 150 moveto
<82b1>show
77 150 moveto
<82ea>show
114 150 moveto
<82cd>show
151 150 moveto
<93fa>show
188 150 moveto
<967b>show
225 150 moveto
<8cea>show
262 150 moveto
<8365>show
299 150 moveto
<834c>show
336 150 moveto
<8358>show
373 150 moveto
<8367>show
410 150 moveto
<82cc>show
447 150 moveto
<89a1>show
484 150 moveto
<8f91>show
\end{verbatim}

Note that the y-coordinate values for \texttt{moveto} calls in the above example are constant. If a constant delta of the pen movement is expected, the same code can become more efficient by using \texttt{ashow}, \texttt{widthshow} and \texttt{awidthshow}, with longer show strings used. \texttt{widthshow} and \texttt{awidthshow} are suitable for languages that separate words by a space or other characters. \texttt{ashow} is an efficient method for continuous text such as Japanese. Example 4 demonstrates how a number of \texttt{moveto/show} pairs can be consolidated into one pair for the same text.

Example 4: \textit{Horizontal Justified Text Rendering Using ashow}

\begin{verbatim}
40 100 moveto
37 0
<82b1 82ea 82cd 93fa 967b 8cea 8365 834c 8358 8367 82cc 89a1 8f91> ashow
\end{verbatim}

\texttt{xshow}, \texttt{yshow}, and \texttt{xyshow} are capable of effectively specifying the widths for each character in a string. If the text line formatting requires more manipulation of character spacing, these operators allow positioning of each character without breaking up the string. Example 5 shows how to set the equal character spacing for the same justified text using \texttt{xshow}. 
Example 5: Use of xshow in Horizontal Text Rendering

40 550 moveto
<82b1 82ea 82cd 93fa 967b 8cea 8365 834c 8358 8367 82cc 89a1 8f91>
[37 37 37 37 37 37 37 37 37 37 37 37 37] xshow

PSPrinter produces slightly more efficient code than that in Example 1, when xshow is not used. If the y-coordinates of glyphs are the same, it attempts to concatenate all the characters and create a longer show string. Even though PSPrinter attempts to create a longer show string, there is still a better alternative to generating more efficient PostScript language code. This is possible by calling Macintosh QuickDraw™ functions with longer character strings. The Macintosh QuickDraw strategy is discussed in the Section 4.4.

For vertical writing text, yshow can be used when the x-value of the coordinate widths for each character is zero. The PostScript language code generated by some application software shows each character positioned by an explicit moveto for each character coordinate. This is true, even through the x-values of the character coordinates are unchanged for each character on one vertical text line. When the origin of a string is known, using the yshow operator would optimize the PostScript language code for a string of more than one character. An array of y-values, which is one of the parameters for the yshow operator, can be determined with values defined by the user, application software and current font size. Application software already have these values but typically do not use them with yshow. See the examples 6 and 7 below.

Example 6: An Example of Vertical Text Rendering

258.36 105.81 moveto <82b7>show
258.36 128.47 moveto <82d7>show
258.36 151.12 moveto <82c4>show
258.36 173.77 moveto <82cc>show
258.36 196.42 moveto <8f91>show
258.36 219.08 moveto <95a8>show
258.36 241.73 moveto <82cd>show
258.36 264.38 moveto <eb41>show
258.36 287.03 moveto <82a0>show
258.36 309.69 moveto <82e9>show
258.36 332.34 moveto <82c5>show
258.36 354.99 moveto <82a0>show
258.36 377.64 moveto <82e9>show
258.36 400.3 moveto <82c6>show
258.36 422.95 moveto <82a2>show
258.36 445.6 moveto <82a4>show
258.36 468.25 moveto <82be>show
258.36 490.9 moveto <82af>show
258.36 513.56 moveto <82c5>show
258.36 536.21 moveto <82b7>show
258.36 558.86 moveto <82c5>show
258.36 581.51 moveto <82c9>show
258.36 604.17 moveto <9581>show
258.36 626.82 moveto <95d5>show
258.36 649.47 moveto <9349>show
With \texttt{yshow}, the same text can be rendered as follows:

\textbf{Example 7: Use of yshow in Vertical Text Rendering}

\begin{verbatim}
514.93 77.47 moveto
<82b7 82d7 82c4 82cc 8f91 95a8 82cd eb41 8f91 95a8 82c5 82a0 82e9
  82c6 82a2 82a4 82be 8af 82c5 82c7 82c5 82c9 9581 95d5 9349
  82c5 82a0 82e9 eb42>
[22.65 22.65 22.66 22.65 22.65 22.66 22.65 22.66 22.65 22.65 22.65
  22.66 22.65 22.65 22.66 22.65 22.65 22.65 22.65 22.65 22.66
  22.65 22.65 22.66 22.65 22.65 22.66 22.65 22.65 22.66 22.66]
\end{verbatim}

Comparison of the above two examples shows that the latter executes with at least 300\% performance improvement. Please note that \texttt{xshow}, \texttt{yshow} and \texttt{xyshow} allow rendering of text in any writing direction. This is useful to provide international support in your application software, as operating systems now provide such opportunities.

\section*{4.2 PostScript Language Procedures for Vertical Text Rendering}

Here is an example of an optimization that can be implemented in a PostScript language program generated when vertical text is used in a document created. Examples of functions that can be defined in these procedures can be summarized as follows:

\textbf{Example 8: Summary of Efficient Text Rendering Procedure}

\begin{verbatim}
Define "global variables" for coordinates
...
Evaluate writing direction
If horizontal
  Show string at coordinate using a show operator variant
If vertical
  Determine character orientation of the text
  If the orientation is the same as horizontal text
    Show string at coordinate using a show operator variant (yshow)
  If the orientation should be rotated in vertical text
    Rotate -90 degrees
    Show string at coordinate using a show operator variant
    Rotate 90 degrees
...
\end{verbatim}

The key is to use the longest \texttt{show} strings possible, because of the overhead associated with the \texttt{show} operators. This method also provides isolation of text placement functions regardless of writing directions within a few proce-
dures. Once the application obtains enough data with character spacing and coordinates of the text, these procedures can be invoked. As a result, ashow, awidthshow, xshow, yshow, or xyshow can be invoked with a longer string.

4.3 Level 1 show Operator Variants

There are Level 1 PostScript show operator variants that may suit your requirements in character positioning. For Level 1 PostScript language output devices, ashow and awidthshow allow character string rendering and placement of each character in the string with an interval of an equal delta value. These operators are more effective when a character string contains more than one character.

Many text-handling applications allow their users to control character spacing and text formatting, such as justification. awidthshow can be used to handle these situations, so that each text line in a formatted document does not have to be treated on a glyph-by-glyph basis.

4.4 QuickDraw Strategy

For most Macintosh application software, QuickDraw calls to draw glyphs can be made for a string, not for one character at a time, to make the PostScript language code more efficient. Character-by-character drawing results in generation of PostScript language code that renders text by each character, if the current version of the Apple LaserWriter is used. Adobe PSPrinter attempts to find characters that are drawn one at a time on the same y-coordinates in one text line and appends them to a character string, then uses the string as a parameter to the show operator. In order to let any PostScript printer driver generate more efficient PostScript language code, a line of text can be drawn by calling such QuickDraw functions as DrawString() or DrawText() in conjunction with SpaceExtra() and CharExtra(). This sequence of QuickDraw calls will invoke either ashow, widthshow, or awidthshow in the resulting PostScript language program.

4.5 “Space Hack”

Some applications attempt to turn off the printer driver’s text layout by doing a “space character hack”. It does so by intentionally imaging a space character at other coordinates before the next character is drawn by calling QuickDraw functions. As a result, the PostScript language code generated by both the Apple LaserWriter and Adobe PSPrinter will include moveto/show pairs to move the pen to a coordinate outside of the rendering space, (–10, –10) for example, and draw a space character of a specified font. Doing so for every character in an entire document will cause major degradation of performance of both the application software and the PostScript language program generated by the printer drivers. Examine the case shown below.
Example 9: *PostScript Language Code with “Space Hack”*

28 48 moveto <82b7>show
-10 -10 moveto ( )show
53 48 moveto <82d7>show
-10 -10 moveto ( )show
78 48 moveto <82cd>show
-10 -10 moveto ( )show
102 48 moveto <8f91>show
-10 -10 moveto ( )show
127 48 moveto <95a8>show
-10 -10 moveto ( )show
152 48 moveto <95a8>show
-10 -10 moveto ( )show
177 48 moveto <82cd>show
-10 -10 moveto ( )show
202 48 moveto <8141>show
-10 -10 moveto ( )show
227 48 moveto <8f91>show
-10 -10 moveto ( )show
251 48 moveto <95a8>show

Instead, if an application wishes to turn off line layout, it should use the StringBegin and StringEnd PicComments as outlined in Apple’s Macintosh Technical Note “Picture Comments – The Real Deal”, an updated version of Apple’s old technical note #91, published in 1992. The performance improvement by the method suggested by Apple over Example 9 is up to 60%. This rate can be higher for documents with a larger number of characters.

5 Conclusion

Optimization for fast text rendering can be simply achieved in application software for Macintosh and other platforms, whether or not it overrides the resident PostScript printer drivers by sending its own PostScript language code. The following are platform-independent guidelines to achieving optimum performance improvements.

- Longer *show* strings achieve better performance. A larger number of calls to the PostScript language *show* operator variants for a given amount of text will cause larger overhead associated with the operators.

- Perform computations to determine lines and glyph placements on the host whenever possible. PostScript language programs that include computation to determine text lines and glyph placement are less efficient. At the time PostScript language code for glyph drawing is generated, applications should have enough data to construct a set of parameters for a single call to one of the *show* operator variants (*ashow*, *xshow*, etc.).
• Simpler operators are faster than complex ones. For example, `ashow` is faster than `awidthshow`, and `awidthshow` is faster than drawing one character at a time with pairs of `moveto` and `show`.

• PostScript operators are always faster than their emulations.

• Less data achieves faster execution and transmission. For example, if something can be done with either `ashow` or `xshow`, `ashow` will be better.
Index

A
Adobe Type Manager (ATM) 5

C
character positioning 7
character positioning
  computation for 7

L
LaserWriter 14
longer show string 12, 13

M
multi-directional text 13

P
PSPrinter 5, 12, 14

Q
QuickDraw functions 12, 14

S
show operator variants
  ashow 7, 14, 15
  awidthshow 7, 14, 16
  xshow 10, 12, 13, 14, 15
  xyshow 13, 14, 16
  yshow 10, 12, 13, 14, 16
  space character hack 14

T
text rendering techniques 7
Type 5

V
vertical writing text 12

Type 1 Coprocessor 5