Emulation of the rectclip, rectfill, and rectstroke Operators

Adobe Developer Support

Technical Note #5123

31 March 1992

Adobe Systems Incorporated

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

PN LPS5123

Contents

Emulation of the rectclip, rectfill, and rectstroke Operators

1 Introduction

Applications programs frequently draw rectangles. Many applications programmers define rectangle procedures for doing this, because they can use speed enhancing optimizations when they know that the figure being drawn is a rectangle.

PostScript™ Level 2 and Display PostScript™ systems provide rectangle operators both for the convenience of the applications programmer and to gain the performance improvements available when drawing rectangles. Since these optimized PostScript language operators execute much more rapidly than defined procedures that perform the same tasks, use the operators whenever possible. These new operators support clipped, filled, and stroked paths.

It is easy to define rectangle procedures for Level 1 interpreters and provide performance gains on those systems as well. If designed properly, the rectangle procedures can be replaced by the corresponding Level 2 operators for the additional speed gain possible when printing on Level 2 devices. Examples of such rectangle procedures appear in the following sections.

In cases where your application cannot determine whether a PostScript language file will be executed on a Level 1 or Level 2 interpreter, replacement can be done automatically on the printer side using *self-configuring code*. An example of self-configuring code also appears below. In actual use, the code shown is more likely to be a part of a larger emulations package installed in a similar manner, than to be installed separately (and into **userdict**) as shown here.

2 Emulating the Level 2 Rectangle Operators

There are several features of the new Level 2 rectangle operators that might differ from rectangle procedures applications programmers write for themselves. Like the Level 2 user path operators, these rectangle operators combine path construction with painting, so that each operator draws and *clips*, draws and *fills*, or draws and *strokes* a number of rectangles.

These operators also support three forms of operands documented in section 4.6.5 of the *PostScript Language Reference Manual, Second Edition* and under the entries for each operator in Chapter 8, "Operator Details." Briefly, they are

- four numbers *x*, *y*, *width*, and *height* that describe a single rectangle. The rectangle's sides are parallel to the user space axes. Its corners are located at the coordinates (*x*, *y*), (*x* + *width*, *y*), (*x* + *width*, *y* + *height*), and (*x*, *y* + *height*). *Width* and *height* can be negative.

- *numarray*, an arbitrarily long sequence of numbers represented as an array. This sequence must contain a multiple of four numbers, each group of four interpreted as one of the rectangles described above. The effect produced is the equivalent of specifying all the indicated rectangles as subpaths of a single combined path, which is then acted upon by a single **clip**, **fill**, or **stroke** operation.

- *numstring*, an arbitrarily long sequence of numbers similar to *numarray*, but represented as an encoded number string.

The first two operand forms are readily emulated for Level 1 devices. The third is difficult to emulate efficiently in terms of Level 1 features. For an application to remain compatible with Level 1 systems, the *numstring* form should not be used.

2.1 One Rectangle

Both the **rectclip** and **rectfill** operator emulations require the same operations for the drawing of a single rectangle. To be fully compatible with those operators, it is necessary to guarantee that all rectangles will be drawn counterclockwise in user space, regardless of the sign of *width* and *height*.

As stated previously, when multiple rectangles are specified in the *numarray* form of operand, all rectangles are treated as subpaths of a single path, acted on by a single **clip**, **fill**, or **stroke**. The "inside" of the single path is considered the union of the insides of each of the multiple rectangles specified. All the subpaths must be drawn in the same direction if the clipping and filling done by the procedures are to show the same results as the operators they emulate.

Two ways guarantee the “always counterclockwise” behavior using the emulations. One way requires an application to restrict the *width*, *height* values it passes to the defined procedures. If both are positive or both are negative, the following code for a basic rectangle will draw counterclockwise:

```
/Rectangle { % x y w h Rectangle - rectangle path
   4 -2 roll moveto        % low left (+) or upper right (-) corner
   exch dup 0.0 rlineto    % to low right (+) or up left (-)
   exch 0.0 exch rlineto   % to up right (+) or low left (-)
   neg 0.0 rlineto         % to up left (+) or low right (-)
   closepath               % close the rectangle
} bind def
```

Another way to get the desired behavior involves having the rectangle path procedure check the signs of *width*, *height* and then use the method that will draw counterclockwise for those values.

```
/RectCC {   % x y w h RectCC - draws counterclockwise for all values
   4 -2 roll moveto              % use x, y coordinates
   2 copy 0.0 lt exch 0.0 lt xor  % check for one arg only being neg
   { dup 0.0 exch rlineto        % do height first
      exch 0.0 rlineto
      neg 0.0 exch rlineto
   }{ exch dup 0.0 rlineto       % do width first
      exch 0.0 exch rlineto
      neg 0.0 rlineto
   } ifelse
   closepath
} bind def
```

The emulations shown here use the second procedure because it is more general. Applications willing to restrict themselves to avoid the overhead of sign testing should use the first procedure, as does the **rectstroke** emulation below. Insideness, and therefore direction, does not matter when stroking paths, and so there is no need for sign testing.

2.2 Emulating rectclip

The behavior of **rectclip** is described in the *PostScript Language Reference Manual, Second Edition*. The effect of **rectclip** is to intersect the inside of the current clipping path with the inside of the rectangles its operands describe (see section 2.1). After computing this new clipping path, **rectclip** resets the current path to empty, as if by performing **newpath**.

If only a single rectangle at a time was to be specified, the **rectclip** procedure would be equivalent to:

```
newpath RectCC clip newpath
```

Since the emulation will also support the *numarray* form of operand, a little extra code is necessary to read from that array, if supplied.

```
/Rc {    % x y w h -or- numarray Rc - rectclip: clip to rectangles
   newpath
   dup type /arraytype eq              % is it an array operand?
   { aload length 4 idiv { RectCC } repeat }
   { RectCC }
   ifelse
   clip newpath
} bind def
```

Note *Level 2 devices do not have the hard limit on operand stack size that Level 1 devices have. It is possible to form an array that is fine on Level 2 but would cause* ***stackoverflow*** *errors on some Level 1 devices.*

The ***aload*** *operator generates such an error if it is passed an array that is too large for the operand stack. It is possible to avoid using* ***aload*** *in a manner that would cause this error for overly large arrays on Level 1 devices (for instance, one might use* ***getinterval*** *to load subarrays of the large array). That has not been done here because it seems reasonable that such an array would have caused an error when it was built, before being handed to the* ***rectclip*** *emulation procedure.*

Building a very large array in a manner that would not cause an error on Level 1 devices is likely to have performance penalties outweighing that of simply keeping to arrays of a known safe size for cases where the language level of the target system is not known to be Level 2.

2.3 Emulating rectfill

The effect of **rectfill** is to fill the one or more rectangles it is passed, rather than to alter the clipping path, otherwise, it is very similar to **rectclip**. Like **rectclip**, its rectangle subpaths must be drawn in a single consistent direction (counterclockwise) so the rectangles will be rendered correctly. Because **rectfill** neither reads nor alters the current path in the graphics state, the emulation procedure's operations are enclosed in a **gsave**/**grestore** pair.

```
/Rf {    % x y w h -or- numarray Rf - rectfill: fill
         % rectangles
   gsave newpath
   dup type /arraytype eq              % is it an array operand?
   { aload length 4 idiv { RectCC } repeat }
   { RectCC }
   ifelse
   fill grestore
} bind def
```

The warning in section 2.2 about **aload** and array size also applies here.

2.4 Emulating rectstroke

Unlike clipping and filling, correct stroking does not depend on the rectangle paths being drawn in one consistent direction. Since there is no reason to test for the sign of *width*, *height*, the first rectangle path procedure shown above is used.

```
/Rs {    % x y w h -or- numarray Rs rectstroke: stroke rectangles
   gsave newpath
   dup type /arraytype eq               % is it an array operand?
   { aload length 4 idiv { Rectangle } repeat }
   { Rectangle }
   ifelse
   stroke grestore
} bind def
```

The warning in section 2.2 about **aload** and array size also applies here.

There are additional points of interest having to do with **rectstroke**. This operator also has argument forms that include a matrix operand. This operand is concatenated to the current transformation matrix after the path is defined, but before it is stroked, so that the matrix applies to the line width and dash pattern, if any, but not to the path itself. We have not implemented those argument forms here, however, this is readily done. Consult the *PostScript Language Reference Manual, Second Edition* for more information.

Another topic related to **rectstroke** is stroke adjustment. This is covered in Technical Note #5111, “Emulation of the **setstrokeadjust** Operator,” and has to do with forcing a uniform device width on lines that have been specified as having the same width in user space. Level 2 devices and Display PostScript systems both provide automatic stroke adjustment, which can be emulated on Level 1 devices.

A rectangle procedure in both plain and stroke adjusting form is shown in Technical Note #5111. Although that procedure can be replaced by **rectstroke** when available, this Rs procedure is a more fully realized version than the one shown in the other technical note, and is supplied in the unified emulations prolog provided in the *PostScript Language Software Development Kit*. The corresponding stroke adjusting form of Rs looks like this:

```
/Mt {    % x y Mt - stroke adjusted moveto: see technical note #5111
   snaptopixel moveto
} bind def

/Rectangle {      % x y w h Rectangle - stroke adj rectangle path
   4 -2 roll Mt          % low left (+) or up right (-) corner
   dtransform round exch          % round w, h: see tech note #5111
   round exch idtransform
   exch dup 0.0 rlineto           % to low right (+) or up left (-)
   exch 0.0 exch rlineto          % to up right (+) or low left (-)
   neg 0.0 rlineto          % to up left (+) or low right (-)
   closepath         % close the rectangle
} bind def

/Rs {    % x y w h -or- numarray Rs rectstroke: stroke adj rectangles
   gsave newpath
   dup type /arraytype eq            % is it an array operand?
   { aload length 4 idiv { Rectangle } repeat }
   { Rectangle }
   ifelse
   stroke grestore
} bind def
```

Technical Note #5111 demonstrates a strategy for emulating the action of the **setstrokeadjust** operator, so that automatic stroke adjustment can be used on Level 1 devices in a manner similar to that available on Level 2 devices. It also discusses different strategies for using Level 2 operator emulations, including self-configuring code for cases in which the target system is unknown, as will also be shown in this technical note.

3 Performance

As stated in the introduction, applications defined rectangle procedures for their own use before Level 2 rectangle operators were available. Aside from the convenience of being able to call a rectangle procedure, performance gains were also seen, chiefly due to a large reduction in transmitted data.

The rectangle operator emulation procedures shown in this technical note do not provide the fastest possible way to draw their respective rectangles, especially if used only in their single-rectangle operand forms. This is due to the overhead of the state saving and graphics operations done by these procedures on each call. When used in their number array forms, these procedures approach the speed of the more basic rectangle procedures, and have the additional advantage of interchangeability with the distinctly faster Level 2 rectangle operators.

For more about performance issues relating to drawing rectangles, see Technical Note #5126, “PostScript Language Code Optimization: Rectangles.” For more on how to exploit this interchangeability, see section 4.

4 Conditional Installation of Emulations

Although this subject is covered more extensively elsewhere (in Appendix D, “Compatibility Strategies,” in the *PostScript Language Reference Manual, Second Edition*; Technical Notes #5111, “Emulation of the **setstrokeadjust** Operator;” #5112, “Emulation of the **makepattern** and **setpattern** Operators;” and #5113, “Emulation of the **execform** Operator”), a brief summary is given in this section.

If a convenient short name interface to the desired Level 2 functionality is defined, those names can be used to perform the same tasks on either Level 1 or Level 2 systems, to the extent that emulations have been created for the tasks. For instance, an Rf procedure was defined above to partially emulate the functionality provided by the Level 2 operator **rectfill**.

If an application restricts itself to using **rectfill** features that have been emulated, it can load the **rectfill** operator into the name Rf on Level 2 systems. It can also load the emulation procedure into that same name Rf on Level 1 devices, allowing the script that the application produces to use the same name for the same task regardless of the level of the device on which the script is executed.

If the application knows which device will execute the script it produces , it can define whichever procedure set is appropriate (the operators or their emulations). If the target device is unknown at the time the procedures must be

defined, both sets can be downloaded with code that will perform installation of the appropriate set over on the device side. The following code performs such an installation.

```
/Level2|DPS systemdict /rectfill known def % is there a rectfill?

Level2|DPS not { save } if    % prepare to remove if not needed
Level2|DPS {      % we do want the following definitions if L2 or
DPS
   /Rf rectfill load def
   /Rc rectclip load def
      ... and so on ...
} if
Level2|DPS not { restore } if  % get rid of the above if not used

Level2|DPS { save } if        % prepare to get rid of the following
Level2|DPS not { % use if not L2 or DPS
   /Rf {
      ... the emulation code ...
   } bind def
   /Rc {
      ... more emulation code ...
   } bind def
      ... and so on ...
} if
Level2|DPS { restore } if     % if L2 or DPS, don't need emulations
```

A complete example of conditional installation of the rectangle operator emulation procedures is given in Appendix A. In reality, these procedures would be part of a larger set of emulations, possibly using some different set of installation conventions. For instance, it is unlikely that such an emulation procedure set would be defined directly into **userdict**.

5 Summary

Rectangle procedures have been used for a long time by applications programmers. Rectangle operations are now supported directly by Level 2 devices and Display PostScript systems. Not only are there speed gains to be realized on Level 2 devices through the use of rectangle operators, emulations and self-configuring code allow the same script to execute on either Level 1 or Level 2 devices without prior knowledge of the device and with good performance characteristics on both language levels.

Appendix A: Self-Configuring Rectangle Code

```
/Level2|DPS systemdict /rectfill known def% is there a rectfill

Level2|DPS not { save } if     % prepare to remove if not needed
Level2|DPS {   % we do want the following definitions if L2 or DPS
   /Rc rectclip load def
   /Rf rectfill load def
   /Rs rectstroke load def
} if
Level2|DPS not { restore } if  % get rid of the above if not used

Level2|DPS { save } if         % prepare to get rid of the following
Level2|DPS not { % use if not L2 or DPS
   /Rectangle { % x y w h Rectangle - rectangle path
      4 -2 roll moveto      % low left (+) or upper right (-) corner
      exch dup 0.0 rlineto  % to low right (+) or up left (-)
      exch 0.0 exch rlineto% to up right (+) or low left (-)
      neg 0.0 rlineto % to up left (+) or low right (-)
      closepath % close the rectangle
   } bind def
   /RectCC {% x y w h RectCC - draws counterclockwise for all values
      4 -2 roll moveto            % use x, y coordinates
      2 copy 0.0 lt exch 0.0 lt xor% check for one arg only being
neg
      {  dup 0.0 exch rlineto % do height first
         exch 0.0 rlineto
         neg 0.0 exch rlineto
      }{ exch dup 0.0 rlineto % do width first
         exch 0.0 exch rlineto
         neg 0.0 rlineto
      } ifelse
      closepath
   } bind def
   /Rc {    % x y w h -or- numarray Rc - rectclip: clip to rectangles
      newpath
      dup type /arraytype eq % is it an array operand?
      {  aload length 4 idiv { RectCC } repeat }
      {  RectCC }
      ifelse
      clip newpath
   } bind def
```

```
/Rf {    % x y w h -or- numarray Rf - rectfill: fill rectangles
      gsave newpath
      dup type /arraytype eq % is it an array operand?
      {  aload length 4 idiv { RectCC } repeat }
      {  RectCC }
      ifelse
      fill grestore
   } bind def

   /Rs {    % x y w h -or- numarray Rs rectstroke: stroke rectangles
      gsave newpath
      dup type /arraytype eq % is it an array operand?
      {  aload length 4 idiv { Rectangle } repeat }
      {  Rectangle }
      ifelse
      stroke grestore
   } bind def
} if
Level2|DPS { restore } if     % if L2 or DPS, don't need emulations
```

Appendix B: Changes Since Earlier Versions

Changes since August 9, 1991 version

- Document was reformatted in the new document layout and minor editorial changes were made.

Index