THE DISPLAY POSTSCRIPT® SYSTEM: SINGLE OPERATOR CALLS VS. WRAPS and the LINE DRAW APPLICATION

Technical Note #5052

March 5, 1990
PostScript® Developer Support Group

Adobe Systems Incorporated
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400

PN LPS5052

Copyright © 1990 by Adobe Systems Incorporated. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher. Any software referred to herein is furnished under license and may only be used or copied in accordance with the terms of such license.

PostScript, Display PostScript, Adobe and the PostScript logo are registered trademarks of Adobe Systems Incorporated. NeXT, the NeXT logo, Application Kit, Digital Librarian, Interface Builder and Workspace Manager are trademarks and NextStep is a registered trademark of NeXT, Inc. Objective-C is a registered trademark of The Stepstone Corp.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes and noninfringment of third party rights.

THE DISPLAY POSTSCRIPT® SYSTEM: SINGLE OPERATOR CALLS VS. WRAPS and the LINE DRAW APPLICATION

Technical Note #5052

March 5, 1990
PostScript® Developer Support Group
(415) 961-4111

1. INTRODUCTION

In this technical note, the use of s*ingle operator calls* from the Client Library and the use of *wraps* created with the **pswrap** translator facility are explored. The Client Library contains the necessary procedures and calls to interface with the PostScript interpreter. One subset of the Client Library is a collection of C-language function calls that interface with single PostScript language operators. An example is the call, **PSmoveto( ).** This function takes an x and a y coordinate and makes it the current point in the graphic state.

The **pswrap** translator, on the other hand, provides the ability to define a *custom* sequence of PostScript language operations that are callable as a C-language procedure. An example of a wrap is shown below. This wrap takes 2 coordinate points as arguments and performs a moveto and a lineto operation and then strokes the line.

This is the definition of the wrap that goes to the translator:

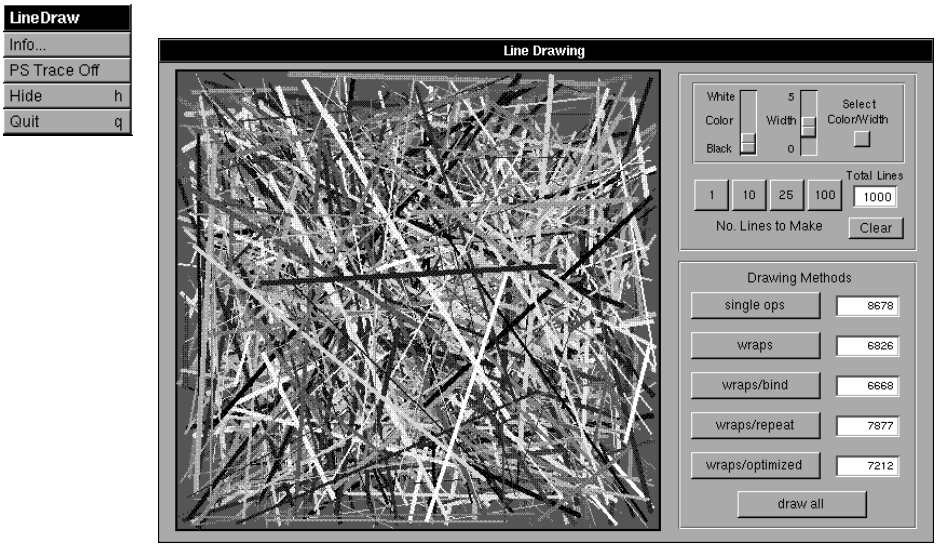
```
/* Wrap definition – examlewraps.psw */
defineps PSWDrawLine(float x1, y1, x2, y2)
  x1 y1 moveto
  x2 y2 lineto
  stroke
endps
```

This is the C-language procedure call:

```
/* Wrap call in a C–language program */
#import <dpsclient/wraps.h>
#include "examplewraps.h"

main()
{
. . .
  PSWDrawLine(100.0, 100.0, 200.0, 200.0);
. . .
}
```

©1990 Adobe Systems Incorporated. All rights reserved.

The **LineDraw** application will be used for examples. In this application, we show several ways to draw lines. Timings are provided which allow for comparison between methods. Each method uses either the single op calls or wraps.



As the times show, wraps are over 20% faster than the single op calls (6826 milliseconds versus 8678 milliseconds in the example shown above). Binding procedures within the wraps and performing much of the data presentation on the client side are also considered. One of the largest impacts on display time is to delay stroking until it becomes necessary, either through a change in a line attribute or the completion of all the paths. In the best case scenario with all the lines the same width and color, this technique can improve display time by 40% over a wrap that strokes every line and 60% over single op calls.

In addition, line widths equivalent to one pixel in device space have been special-cased within the Display PostScript® system and as a result will print and display considerably faster than other line widths. A line width setting of 0.15 is suggested to obtain the benefits of this feature.

In the course of introducing single op calls and wraps, a little groundwork is provided on the *client-server* architecture of the Display PostScript system and its use of *contexts*.

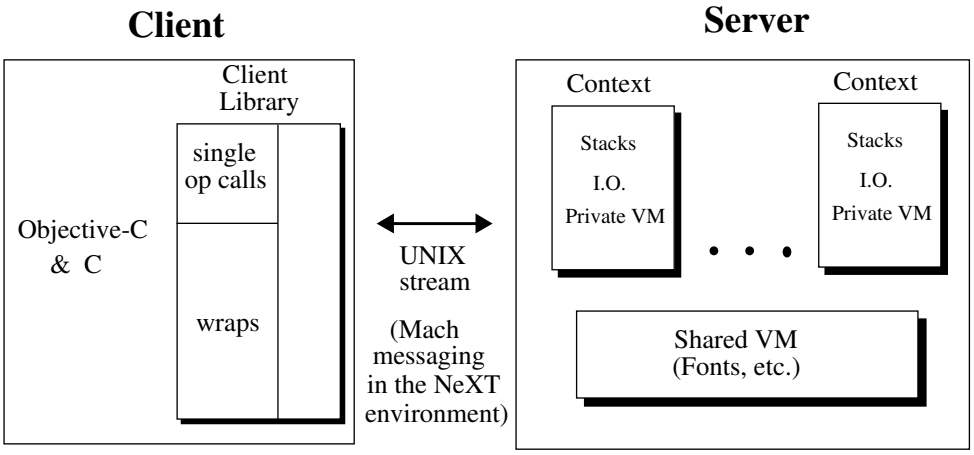
©1990 Adobe Systems Incorporated. All rights reserved.

2. CLIENT-SERVER MODEL

The Display PostScript system makes use of a client-server network architecture. The interpreter is the server and each application is a client. The client and server are most often separate processes on the same machine, but the client can also exist on a remote machine. The client sends PostScript language code to the server either through single operator calls or through wraps. Arguments can also be returned from the server. The Client Library implements the client-server communication transparently with respect to the low level communication protocols used by different windowing systems.

Each client within the Display PostScript system operates within a *context*. A context can be thought of as a virtual printer. It has its own set of stacks, input/output facilities and memory space. Separate contexts allow multiple applications to share one PostScript interpreter running as a process in the server.

Although the Display PostScript system supports multiple contexts per client (application), one context is usually sufficient for all drawing within an application and greatly simplifies programming overhead. A single context can handle many **Views**, the Application Kit object specifically designed for drawing on the screen. An example when a separate context might be warranted is when importing EPS files. A separate context for EPS files will simplify error recovery if the included EPS files contain PostScript language errors.



The server handles the scheduling associated with executing contexts in time slices. Each client has access to a separate and private portion of *virtual memory space* within its context. An additional portion of memory is shared among all clients. This portion is called, surprisingly enough, shared VM and holds system fonts among other items. (Private VM can also hold fonts but they are private to the client.)

The structure of a context is the same across platforms. The creation and management of a context, however, can be different. In the case of the NeXT computer, the **Application** object within the Application Kit manages the handling of contexts. For the most part, context operations are transparent to the developer. The main concern for the developer is the execution of PostScript language operators within a context.

©1990 Adobe Systems Incorporated. All rights reserved.

3. SINGLE OP CALLS

The Client Library manages the communication channel with the server. A major portion of the library are two sets of calls that invoke single PostScript language operators. These two sets are almost identical in function. The only difference is that the first group (found in **dpsops.h**) takes as its first argument an explicit context in which to execute the operator. Those in the second group (found in **psops.h**) do not take a context paramter because the operators execute within the *current* context. Since most applications in the NeXT environment only use one context, the second group of functions is usually used. The two sets are identical in respect to number and purpose of the function calls, the only difference is the context paramter that is passed in the **dpsops.h** set. Each PostScript language operator has a corresponding single operator call in both **dpsops.h** and **psops.h**. Examples of each for the PostScript language operator **moveto** are:

```
DPSmoveto(DPSContext ctxt; float x, y;);

PSmoveto(float x, y;);
```

A number of single operator calls require operands but do not take arguments. An example is **PSadd( )**. Two operands are needed but no arguments are taken.These types of operators use operands from the operand stack. In cases such as these, the operands can be placed on the stack either in a wrap that appears before the function call or through an operation that has left data on the stack. Another way to place data on the stack in the NeXT implementation is through the use of the **PSsend...** set of single operator calls: **PSsendboolean( )**, **PSsendchararray( )**, **PSsendfloat( )**, **PSsendfloatarray( )**, **PSsendint( )**, **PSsendintarray( )** and **PSsendstring( )**. Each of these functions take appropriate arguments and place them on the stack. Other calls can then be used to perform operations with the arguments on the stack. A corresponding set of **PSget...** calls can be used to retrieve objects from the operand stack.

Single operators are easy to use and can be used for such simple drawing as filling in a rectangle or drawing a few lines. Anything more complex should use wraps designed specifically for the operation. Each wrap call incurs a certain amount of processing overhead withing the system, so combining operations from single op wraps into larger wraps reduces the total execution time.

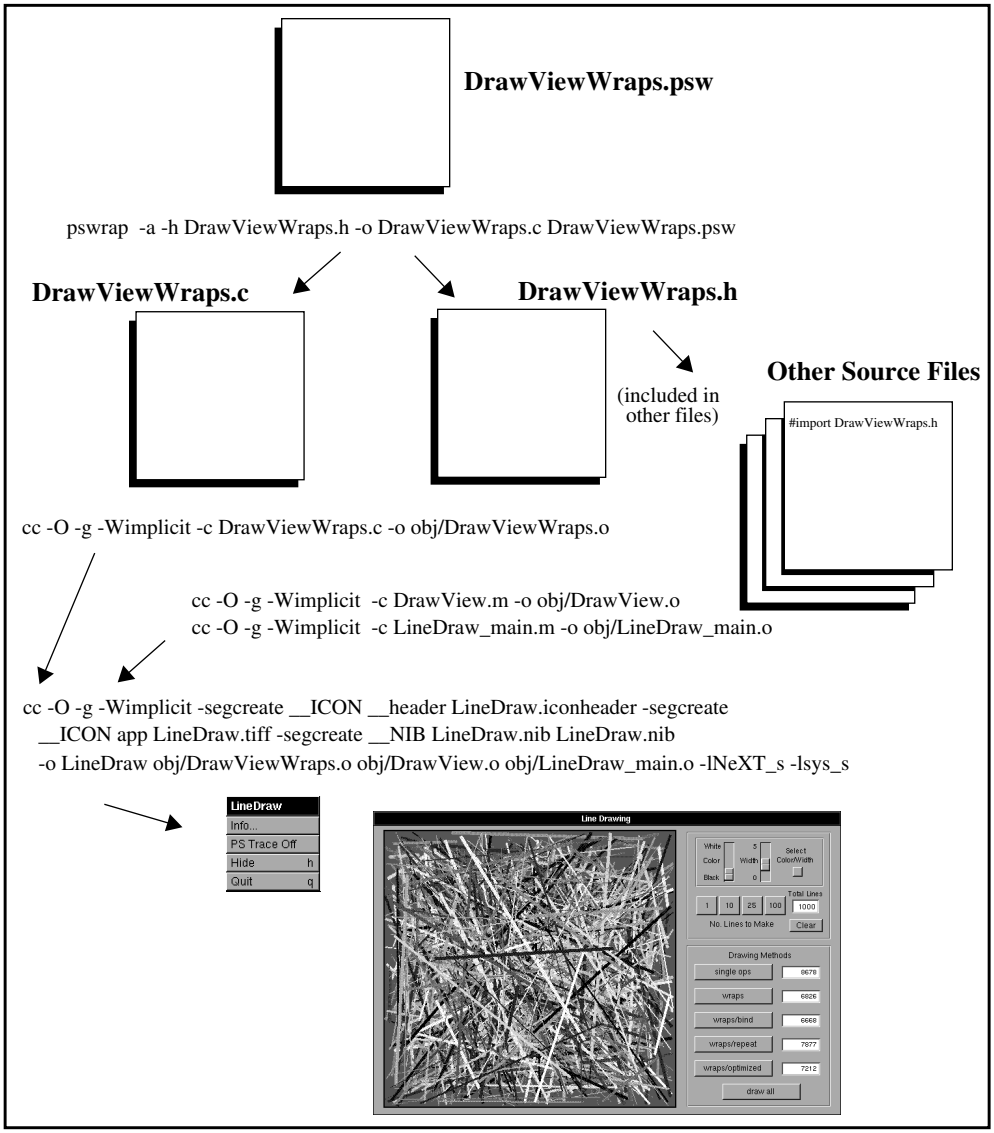
For the most part, single operator calls, or even wraps for that matter, should not be used if the same operation can be done entirely within the client. For example, to obtain the angle formed by a given x and y value, the UNIX function, **atan( )**, should be used instead of the **PSatan( )** operator in the Client Library. Although **PSatan( )** will work, the UNIX call is more efficient. The PostScript language is better suited for *imaging* than for *processing*.

4. WRAPS

The Display PostScript system provides a translator called **pswrap** that takes a set of PostScript language instructions and produces a C-language procedure call interface. In other words, a group of PostScript language operators can be grouped together into one procedure call which then can be used within an application. This approach is similar to the idea of providing a C-language procedure call interface to a FORTRAN Library. Arguments can be passed to and received from the server. The single op calls found in the Client Library are a special set of pre-defined wraps.

©1990 Adobe Systems Incorporated. All rights reserved.

The **pswrap** translator is a stand-alone program that works like a compiler. The translator takes the text representation of the PostScript language code and converts it into a C-language source file of calls to the Client Library routines. The Client Library will handle formatting the PostScript language operators into a binary format (a more efficiently executed representation) and send them to the server. The **pswrap** translator produces a source-level C-language file and an include file containing the function calls and the external declarations. Once this C-language file has been compiled and linked within an application, calls to the functions will send the binary representation to the interpreter.



©1990 Adobe Systems Incorporated. All rights reserved.

The **pswrap** translator has the following invocation in the shell:

```
pswrap [-ar] [-o outputCfile] [-h outputHfile] [-s length] [inputFile]

-a uses ANSI C procedure syntax
-o specifies the outputCfile (default is standard out)
-h generates the external declarations in the header file specified
-s Max length of string input in PS program
-r Wrap can be used in a shared lib, generates extra code, use only if necessary
outputCfile is the source file name desired (the .c extension should be included)
outputHfile is the header file name desired (the .h extension should be included)
inputFile is the .psw or .pswm file (default is standard in)
```

The makefile generated by Interface Builder will invoke **pswrap** provided the wrap file is included in the **Project Inspector**. The header (**.h**) and source (**.c**) files do not need to be included in the **Project Inspector**. The wrap file is sufficient. The header file, however, should be declared as an import in the files that call the wraps. In the LineDraw application, **WorldView.m** imports **DrawViewWraps.h**, the header file for **DrawViewWraps.psw**.

4.1 Wrap Declarations

This section briefly describes the wrap syntax used in the examples. The **pswrap** documentation provided by Adobe Systems Incorporated. contains a more comprehensive explanation of the wrap syntax. This documentation should be referred to should any questions arise.

Each definition contains three parts:

- **defineps / endps** pairing
- C-language procedure call definition (including argument types)
- PostScript language code

defineps/endps Pairing

Each wrap is enclosed between a **defineps** and **endps** pairing. All the text included between the pairing, except for the procedure call definition, should be PostScript language operators. The text is processed by the translator and converted into binary objects. All text not included between the pairings is passed to the source level C-language file untouched. As a result, comments inside each **defineps / endps** should be a PostScript language comment (%). Comments outside each **defineps / endps** should be C-language comments (/* ... */).

Procedure Call Definition

The call definition contains the procedure name and argument list in parenthesis. The procedure is of type **void**. No other return type is possible. By default the procedure names are **external**, but they can be declared **static**. Arguments can be passed to the server and returned from the server.

©1990 Adobe Systems Incorporated. All rights reserved.

PostScript Language Code

All text included between a **defineps** and an **endps** should be PostScript language operators or arguments declared in the procedure call definition. The **pswrap** translator performs the conversion of the text into binary objects but does not validate the operations. Errors in the body of a wrap will not be caught during translation but will instead cause exceptions at run-time.

5. LINE DRAW EXAMPLES

The purpose of the **LineDraw** application is to draw lines and then time the results. This application contains one example of using single operator calls and four using variations of wraps. The wraps can be found in **DrawViewWraps.psw**; the invocations in **DrawView.m**.

Six arrays of floating point numbers store values that are used to draw the lines – **X**, **Y**, **X1**, **Y1**, **C** and **W**. The arrays store the starting x and y points, the ending x and y points and the color and line width for each line. These arrays are populated when the user selects certain controls in the interface. The starting x and y point and the ending x an y point for each line is generated with the **rand( )** function and modified to fall with the bounding rectangle of the view in the application. The color and line width can be set to specific values for each set of lines or generated randomly, again modified to fall within certain ranges. The ability to display sets of lines with uniform color and line widths as well as randomly generated color and line widths provides the best and worst cases for display times.

Each of the five examples of displaying lines has been placed in a separate method. These methods are messaged by the **drawSelf::** method if the menu item for the specific drawing operation has been turned on. The recommended steps for drawing in a view is to place the drawing in **drawSelf::** (or have **drawSelf::** call procedures or message other methods to perform the drawing). When the view is to be displayed, the **display** method defined in the **View** class should be messaged. The **display** method will in turn message **drawSelf::**. In this application, the **Draw** button messages a method in the **DrawView** class which in turn messages **display**.

5.1 PSWMarkTime( ) and PSWReturnTime( )

PSWMarkTime( ) and **PSWReturnTime( )** are used in all the methods of drawing to obtain the processing time for each example. **PSWMarkTime( )** defines **StartTime** and assigns it the value of the PostScript language operator, **realtime**, at the time of the wrap invocation. **PSWReturnTime( )** gets the current realtime value and subtracts **StartTime**. The difference is returned from the server to the client in the output argument, **ElapsedTime**. Output arguments appear after the input arguments in the wrap declaration and are separated from the input arguments by a vertical slash (|). (A call to **NXPing( )** is inserted immediately after the **PSWMarkTime( )** in order to flush the buffer to the server. This ensures the immediate execution of the **realtime** operation. An **NXPing( )** is not necessary after the **PSWReturnTime( )** because the inclusion of a return argument automatically flushes the buffer.)

©1990 Adobe Systems Incorporated. All rights reserved.

The PostScript language wrap definitions:

```
/* Wrap definitions – DrawViewWraps.psw */
defineps PSWMarkTime()
  /StartTime realtime def
endps

defineps PSWReturnTime( | int *ElapsedTime)
  realtime StartTime sub
  ElapsedTime
endps
```

And their invocations from the C-language world:

```
/* Wrap calls – DrawView.m */
int ElapsedTime;
. . .
PSWMarkTime();  NXPing();
. . .
PSWReturnTime(&ElapsedTime);
```

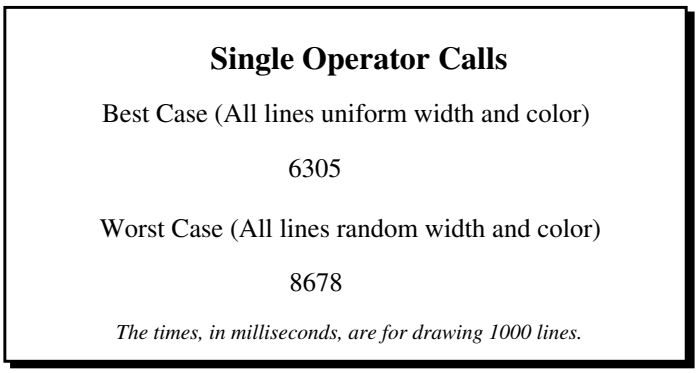
Whenever an output argument is encountered within a wrap, the topmost value on the operand stack is removed and placed in the address that the output argument provides. Any attempt by a subsequent PostScript language operator to use the value will either use the wrong value or produce a stack underflow error message. If an output argument is used multiple times within a wrap, its value upon return will be the last value assigned.

©1990 Adobe Systems Incorporated. All rights reserved.

5.2 Single Op Calls

The single operator calls in the Client Library are used just like other C-language function calls. The calls are contained in the **drawSingleOps** method. This method is messaged within the **drawSelf::** method.

```
-drawSingleOps
{
  int ElapsedTime, counter;

  PSWMarkTime(); NXPing();

  PSsetgray(BGCOLOR);
  PSrectfill(
    0.0, 0.0, bounds.size.width, bounds.size.height);
  PSsetgray(BGSTRCOLOR);
  PSsetlinewidth(BGSTRWIDTH);
  PSrectstroke(
    0.0, 0.0, bounds.size.width, bounds.size.height);

  for (counter = 0; counter < TotalLines; counter++)
  {
    PSsetlinewidth(W[counter]);
    PSsetgray(C[counter]);
    PSmoveto(X[counter], Y[counter]);
    PSlineto(X1[counter], Y1[counter]);
    PSstroke();
  }

  PSWReturnTime(&ElapsedTime);
  [displaySingleOpTime setIntValue:ElapsedTime];

  return self;
}
```



PSWMarkTime( ) and P**SWReturnTime( )** are wraps used for timing. The others calls are single operators calls from the Client Library, more specifically **psops.h**. The view is first cleared by drawing over the previous drawing with the background color and then stroking the border. The names in all capitals are pre-defined literals. The values passed to **PSrectfill( )** and **PSrectstroke( )** are the origin and dimensions for the bounding rectangle for the view.

©1990 Adobe Systems Incorporated. All rights reserved.

The **for** loop draws one line each iteration. Some of the functions in the loop take arguments, such as **PSsetgray( )**. Others do not such as **PSstroke( )**. The number and type of arguments for each call can be found in the include files for the single op calls.

5.3 Simple Wraps

The second way of drawing lines uses very simple wrap definitions. The wrap definitions appear first followed by their invocation in the **DrawView** class method, **drawWraps**. The first wrap clears the screen and the second creates a line path and then strokes it.

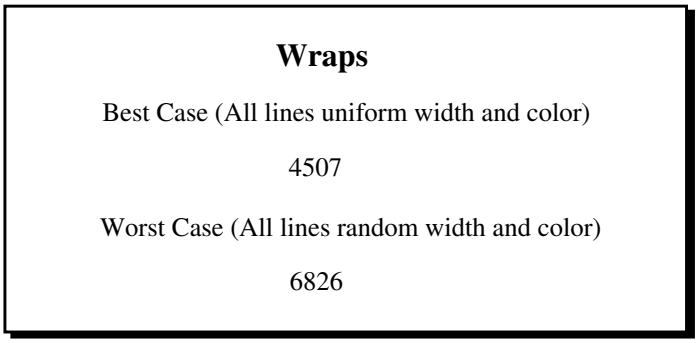
PostScript language wraps:

```
/* Wrap definitions – DrawViewWraps.psw */
  defineps PSWEraseView(
      float BGColor, BGStrColor, BGStrWidth, BGrect[4])
  BGColor setgray
  BGrect rectfill
  BGStrColor setgray
  BGStrWidth setlinewidth
  BGrect rectstroke
endps

defineps PSWDrawLine(float LineWidth, LineColor, X, Y, X1, Y1)
  LineWidth setlinewidth
  LineColor setgray
  X Y moveto X1 Y1 lineto stroke
endps
```

C-language invocations:

```
/* Wrap call – DrawView.m*/
–drawWraps
{
  int ElapsedTime, counter;
  floatViewRect[4];

  PSWMarkTime(); NXPing();
  ViewRect[0] = ViewRect[1] = 0.0;
  ViewRect[2] = bounds.size.width;
  ViewRect[3] = bounds.size.height;
  PSWEraseView(BGCOLOR, BGSTRCOLOR, BGSTRWIDTH, ViewRect);

  for (counter = 0; counter < TotalLines; counter++)
  {
    PSWDrawLine(
        W[counter], C[counter], X[counter], Y[counter],
        X1[counter], Y1[counter]);
  }

  PSWReturnTime(&ElapsedTime);
  [displayWrapTime setIntValue:ElapsedTime];

  return self;
}
```

©1990 Adobe Systems Incorporated. All rights reserved.



The float array, **ViewRect**, is used to pass the bounding rectangle of the view to the wrap in one structure rather than as four individual values, simplifying the call. In the wrap declaration, the array is specified as a float array of size 4. Another example will show the declaration and use of a dynamically sized array. The individual elements in an array can be used within a wrap by specifying an index or the entire array by not including an index. In **PSWEraseView( )**, the entire array is placed on the operand stack.

As in the single operator example, the **for** loop draws one line per iteration. The coordinate points for the line and the color and line width for each line are passed to the wrap. The wrap sets the color and line width in the graphic state of the PostScript interpreter, creates the path and then strokes it.

5.4 Wraps with Binding

This example is quite similar to the previous example except that PostScript language *procedures* are called within the wrap instead of PostScript language *operators*. The necessary PostScript language operators will appear in the procedures. A wrap called **PSWDefs( )** defines the procedures. It is called in the **newFrame:** method of the **DrawView** class so that the definitions will exist in the server when the wraps invoke them. These procedures are created so that they can be *bound* using the **bind** operator. Binding replaces each executable operator name with its value. During execution of the procedures, the interpreter will encounter pointers to the executable code that implements the operators rather than the names of the operators.

Whenever the interpreter encounters a name, it looks the name up in the dictionaries on the dictionary stack. Name lookups are fast but they still represent noticeable processing overhead. Binding forces the look up of the names of the operators at definition time instead of every time the operators are executed. Binding serves the most use for frequently called sequences of code. (In this case, the view is not erased frequently so the savings is negligible for the PostScript language procedure, **EVB**. The savings is much larger for **DLB**, however, since this sequence is invoked many more times.)

The two procedures below in **PSWDefs( )**, **EVB** and **DLB**, have been assigned short names. In print drivers, the reduction in data communications overhead from using short names can reduce processing time by up to 30%. The overhead of name length is insignificant here due to **pswrap** name optimizations and the binary encoding scheme used by the Display PostScript system. When this stream is converted to ASCII and sent to a printer, short names for procedures are a worthwhile convention. Note that the *wrap* names

©1990 Adobe Systems Incorporated. All rights reserved.

have not been reduced. The wrap is a C-language function rather than an executable PostScript language name. Long wrap names are desirable because they make the application code easier to read and have no performance impact.

PostScript language wraps:

```
/* Wrap definitions - DrawViewWraps.psw */
defineps PSWDefs()
. . .
  /EVB { % BGrect BGStrWidth BGStrColor BGColor
    setgray 2 index rectfill
    setgray setlinewidth rectstroke
  } bind def

  /DLB { % X1 Y1 X Y LineColor LineWidth
    setlinewidth setgray
    moveto lineto stroke
  } bind def
  . . .
endps

defineps PSWEraseViewBind(
    float BGColor, BGStrColor, BGStrWidth, BGrect[4])
  BGrect BGStrWidth BGStrColor BGColor EVB
endps

defineps PSWDrawLineBind(float LineWidth, LineColor, X, Y, X1, Y1)
  X1 Y1 X Y LineColor LineWidth DLB
endps
```

C-language invocations:

```
/* Wrap call - DrawView.m */
-drawWrapsBind
{
  int ElapsedTime, counter;
  floatViewRect[4];

  PSWMarkTime(); NXPing();
  ViewRect[0] = ViewRect[1] = 0.0;
  ViewRect[2] = bounds.size.width;
  ViewRect[3] = bounds.size.height;
  PSWEraseViewBind(BGCOLOR, BGSTRCOLOR, BGSTRWIDTH, ViewRect);

  for (counter = 0; counter < TotalLines; counter++)
  {
    PSWDrawLineBind(
        W[counter], C[counter], X[counter], Y[counter],
        X1[counter], Y1[counter]);
  }

  PSWReturnTime(&ElapsedTime);
  [displayWrapBindTime setIntValue:ElapsedTime];

  return self;
}
```

©1990 Adobe Systems Incorporated. All rights reserved.

Wraps with Binding

Best Case (All lines uniform width and color)

4420

Worst Case (All lines random width and color)

6668

5.5 Wraps with Interpreter Loop

The fourth example performs the drawing of the lines in a loop within the interpreter. As the **DLRB** procedure shows, many stack operations such as **roll**'s, **dup**'s and **exch**'s are necessary. The resulting drawing time shows that this method is fairly slow when compared to some of the other methods. In most circumstances, extensive programming with the PostScript language should be avoided not only because of the slower performance but also because of the unnecessary complexity added. The client should handle as much of the presentation of the data as possible.

PostScript language wraps:

```
/* Wrap definitions – DrawViewWraps.m */
defineps PSWDefs()
. . .
  /DLRB { % i – number of times to loop
    0 1 3 -1 roll 1 sub { % for
      dup PSW exch get setlinewidth
      dup PSC exch get setgray
      dup PSX exch get PSY 2 index get moveto
      dup PSX1 exch get PSY1 2 index get lineto
      stroke pop
    } for
  } bind def
  . . .
endps

defineps PSWDrawLineRepeatBind(
    float W[i], C[i], X[i], Y[i], X1[i], Y1[i]; int i)
  /PSW W def
  /PSC C def
  /PSX X def /PSY Y def
  /PSX1 X1 def /PSY1 Y1 def
  i  DLRB
endps
```

©1990 Adobe Systems Incorporated. All rights reserved.

C-language invocations:

```
/* Wrap call - DrawView.m */
-drawWrapsRepeat
{
  int ElapsedTime;
  floatViewRect[4];

  PSWMarkTime(); NXPing();
  ViewRect[0] = ViewRect[1] = 0.0;
  ViewRect[2] = bounds.size.width;
  ViewRect[3] = bounds.size.height;
  PSWEraseViewBind(BGCOLOR, BGSTRCOLOR, BGSTRWIDTH, ViewRect);
  PSWDrawLineRepeatBind(W, C, X, Y, X1, Y1, TotalLines);

  PSWReturnTime(&ElapsedTime);
  [displayWrapRepeatTime setIntValue:ElapsedTime];

  return self;
}
```

Wraps with Interpreter Loop

Best Case (All lines uniform width and color)

5551

Worst Case (All lines random width and color)

7877

This example erases the view with a wrap borrowed from the previous example. In the wrap that displays the lines, **PSWDrawLineRepeatBind( )**, the six arrays that define the lines are passed to the server which then performs the loop for each line. This case shows the use of dynamically sized arrays within wraps. The entire arrays and their size are passed as arguments. In the wrap declaration, the array size variable is used to specify the size of the arrays.

```
defineps PSWDrawLineRepeatBind(
    float W[i], C[i], X[i], Y[i], X1[i], Y1[i]; int i)
```

This capability provides a good deal of freedom when passing data to the server. Other technical notes will provide examples where passing large, dynamically sized arrays can provide significant time savings. In this example, the benefit is lost because of the stack manipulations necessary to display the lines.

©1990 Adobe Systems Incorporated. All rights reserved.

5.6 Wraps with Optimized Stroking

This method of drawing lines is similar to the *Wraps with Binding* example except that the stroking of the lines takes place only after the line width or color has changed. The path is extended (with **moveto's** and **lineto's**) without stroking as long as the width and color are the same. If the width or color changes or the number of lines end, the resulting path is stroked and then the path begun anew. The PostScript language operator, **stroke**, paints a line over the current path in the current color using the current line attributes (width, line cap, etc.) in the graphic state. As long as the color or line attributes do not change, significant stroke overhead can be eliminated by collecting a series of disconnected sub paths and then performing a single stroke operation on the entire series.

A random ordering of the lines types does not show any advantage over stroking every line. When the lines are grouped according to the same line attributes, however, this method is the fastest of all the methods explored in this paper. This method of drawing may be useful when displaying a grid or other drawings with lines of uniform color and line attributes. It also means that there may be some advantage to ordering lines within an application's data structures according to similar line attributes.

PostScript language wraps:

```
/* Wrap definitions – DrawViewWraps.psw */
defineps PSWDefs()
  . . .
/MLB { % X1 Y1 X Y
    moveto
    lineto
  } bind def

  /SLB { % LineColor LineWidth
    setlinewidth
    setgray
    stroke
  } bind def
  . . .
endps

defineps PSWMakeLineBind(float X, Y, X1, Y1)
  X1 Y1 X Y MLB
endps

defineps PSWStrokeLineBind(float LineWidth, LineColor)
  LineColor LineWidth SLB
endps
```

©1990 Adobe Systems Incorporated. All rights reserved.

C-language calls:

```
/* Wrap call - DrawView.m */
-drawOptimizedStroke
{
  int ElapsedTime,counter;
  float ViewRect[4];

  PSWMarkTime(); NXPing();
  ViewRect[0] = ViewRect[1] = 0.0;
  ViewRect[2] = bounds.size.width;
  ViewRect[3] = bounds.size.height;
  PSWEraseViewBind(BGCOLOR, BGSTRCOLOR, BGSTRWIDTH, ViewRect);

  for (counter = 0; counter < TotalLines; counter++)
  {
    PSWMakeLineBind(
        X[counter], Y[counter], X1[counter], Y1[counter]);

    if (!((counter != TotalLines -1) &&
        (C[counter] == C[counter + 1]) &&
        (W[counter] == W[counter+1])))
    {
      PSWStrokeLineBind(W[counter], C[counter]);
    }
  }

PSWReturnTime(&ElapsedTime);
  [displayOptimizedStroke setIntValue:ElapsedTime];

  return self;
}
```

Wraps with Optimized Stroking

Best Case (All lines uniform width and color)

2755

Worst Case (All lines random width and color)

7212

Note: *The PostScript interpreters in most printers have a 1500 point path length limit. For printer compatibility, it may be wise to place a ceiling on the number of points allowed in a path when using this stroke delaying technique. The limit is larger for the first release of the Display PostScript system.*

Also Note: *This optimization technique should not be used when filling objects. The performance is not improved and the results are visually unpredictable due to the interactions between the PostScript language fill rule and overlapping subpaths. It is best to draw and fill each object individually.*

©1990 Adobe Systems Incorporated. All rights reserved.

6. SMALL WIDTH LINES

Line widths equivalent to one pixel in device space display considerably faster than other line widths because the interpreter contains optimizations for these lines. A line width setting of 0.15 is suggested to obtain the benefits of this feature without the absolute device dependence of a zero line width. (A zero line width is, by definition, one pixel on all devices.) A line width setting of 0.15 will produce a one pixel line on devices with resolutions of 400 dpi or less but will still be perceptible on higher resolution devices.

Small line widths are desirable for such uses as displaying a wire frame drawing of an image. In this instance, a line width setting of 0.15 will display much faster than one that does not use the optimization feature. The table below gives two times, in milliseconds, for drawing 1000 lines - once with a 0.15 point line width, the another with a two point line width.

Optimized Stroke Method (Uniform line color)

.15 Point Line Width	1779
2.0 Point Line Width	2755

©1990 Adobe Systems Incorporated. All rights reserved.

7. SUMMARY

The Display PostScript system uses a client-server architecture. The clients send drawing instructions to the server. The server maintains a context for each client. The server contains the memory and stacks for each context. Although a client can have multiple contexts, one context per client or application is recommended except for unusual circumstances. Drawing instructions can be sent from the client to the server either through single operator calls or through wraps.

The single operator calls are an easy way to send drawing instructions to the server. Anything but the simplest drawing should probably make use of wraps, though. The **pswrap** translator takes PostScript language instructions and produces a C-language procedure call interface which can then be used in an application. Creating a custom wrap can produce significant improvements over the single operator calls. Additional improvements can be gained by binding procedures used in wraps and by performing most of the data presentation on the client side. Delaying stroking until necessary can also produce large gains when the paths share the same line attributes. The table below summarizes times, in milliseconds, obtained from the **LineDraw** application for drawing 1000 lines.

Best Case (All lines uniform width and color)

Wraps with Optimized Stroking	2755
Wraps with Binding	4420
Wraps	4507
Wraps with Interpreter Loop	5551
Single Operator Calls	6305

Worst Case (All lines random width and color)

Wraps with Binding	6668
Wraps	6826
Wraps with Optimized Stroking	7212
Wraps with Interpreter Loop	7877
Single Operator Calls	8678

©1990 Adobe Systems Incorporated. All rights reserved.