POSTSCRIPT®

THE DISPLAY POSTSCRIPT® SYSTEM: PATH CONSTRUCTION & RENDERING and the DIAL APPLICATION

Technical Note #5054

March 6, 1990
PostScript® Developer Support Group

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

PN LPS5054

POSTSCRIPT®

THE DISPLAY POSTSCRIPT® SYSTEM: PATH CONSTRUCTION & RENDERING and the DIAL APPLICATION

Technical Note #5054

March 6, 1990
PostScript® Developer Support Group
(415) 961-4111

1. INTRODUCTION

Technical note #5052 describes the use of the single operator calls and wraps to render paths. This technical note on user paths explores the issue of efficiently rendering paths a little further. *User paths* are Display PostScript® system extensions to the PostScript® language that can be used to render paths. User paths can be invoked by single operator calls or by wraps and can provide advantages that conventional path construction does not. Four of the larger benefits of user paths are 1) they are more efficient to interpret and execute, 2) they provide a convenient way to send an arbitrary path to the server, 3) they are compact and minimize data transmission when the client and server are on different machines and 4) they can be cached.

By definition, a user path is a PostScript language procedure that consists entirely of path construction operators and their corresponding operands expressed as literal numbers. Another way of saying this is that a user path is a combination of a complete and self-contained description of a path within user space. A number of Display PostScript system rendering operators can act on user paths to perform standard PostScript language operations such as stroking or filling a user path. A user path can be described in two ways, either as ASCII text or in an encoded format which consists of an array of two elements—an array of operands and a string of operators.

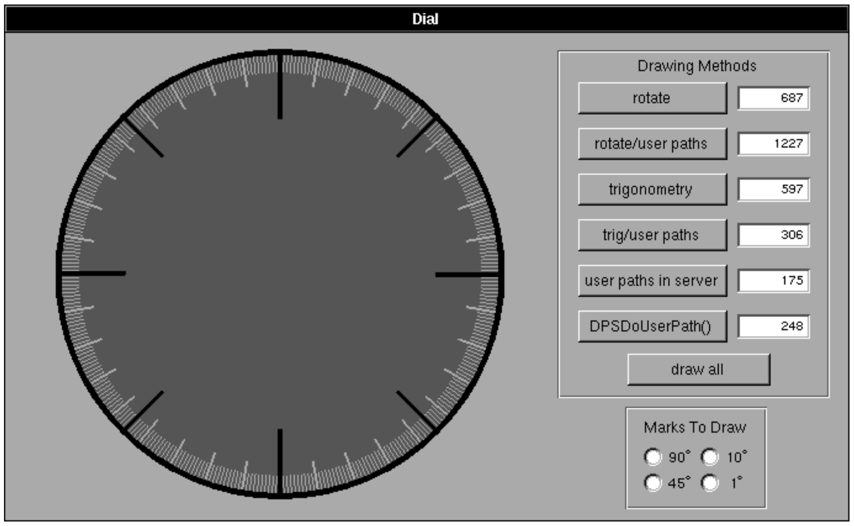
The **Dial** application will be used to show six ways of drawing hash marks around the inner edge of a dial. The first method uses the same line description for all the marks but rotates the user space each time to render the line at a different angle around the dial. The second follows a similar approach except that the line description is sent as a user path and then appended to the current path with the **uappend** operator. The third method performs trigonometric calculations on the client side to find the precise coordinates for **moveto/ lineto** couplings.

The fourth method takes the trigonometric approach one step further by placing the **moveto/lineto** path constructions into a single, large user path and then executing one **ustroke** operation on the entire user path.[1] The fifth approach follows the same steps as the

1. All of the approaches delay the stroking until all of the marks sharing the same characteristics have been constructed. In addition, the PostScript language operators in the first three approaches are placed into a bound procedure to eliminate frequent name lookups. The wrap in the last approach is not called frequently enough to warrant this step. These two lessons are detailed in Technical Note #5052, The Display PostScript System, Single Operator Calls vs. Wraps.

fourth except that instead of sending the user path descriptions to the server when drawing each time, the descriptions are stored in the server at initialization. When drawing the user path, only the name of the user path is passed to the server instead of the entire user path.

The sixth method uses the NeXT procedure, **DPSDoUserPath( )** to send a user path to the server. In some cases the execution time is about 20% faster than sending the user path to the server in a custom wrap (binary encoded numbers are used instead of the normal number representation format). The primary advantage of **DPSDoUserPath( )** is that it automatically provides an emulation of user paths and their rendering operators when printing to devices that do not contain the PostScript language extensions for the Display PostScript system.



Timing results show that the three methods that calculate the positions of the hash marks and then place them into user paths are faster than rotating the user space, using **uappend** or using a wrap for each **moveto** / **lineto** operation. Of the three different ways of drawing with user paths, retaining the user path in the server is the fastest method. This method of drawing avoids the overhead of the trig calculations and the transmission time for each drawing, but this method is only valuable for paths that are drawn frequently and that do not change. Otherwise, **DPSDoUserPath( )** provides an efficient way to transmit user paths to the server. Sections that follow will explain these results and also defend some of the poorer performing methods.

2. USES

A user path can provide an advantage over conventional path construction in a number of instances. One instance is when the path is known ahead of time. Instead of creating a wrap to construct the path, an encoded user path can be used to retain the path in a compact form. When it comes time for rendering, the user path can be invoked with the desired user path rendering operator. An example where a user path might be suited for this purpose would be to retain the descriptions of clock hands. Instead of drawing the hands within a wrap using **moveto**’s, **lineto**’s, **curveto**’s, etc., a user path could be retained in a static array and then sent to the server either in a wrap or through single operator calls.

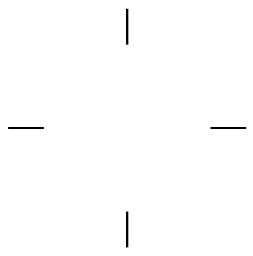
Even if the path is dynamically created, user paths can be a big win. As the **Dial** application shows, it is faster to place a large path in a user path on the fly than it is to send the drawing instructions individually. Graphics applications can make use of this by either retaining a user path definition for each object or by converting the internal path representation to a user path immediately before drawing. Since dynamically sized arrays can be sent to the server, an application can maintain a large buffer for the express purpose of transmitting user paths to the server.

3. USER PATH DEFINITION

A user path is a path description with the operands in a literal format. User path rendering operators such as **ustroke** and **ufill** take a user path as an argument and perform the appropriate rendering operation. The example below shows one of the two possible formats for user path description along with the **ustroke** operator.

User Path with Rendering Operator (ASCII Path Description) **Result**

```
{
0 0 200 200 setbbox
175 100 moveto
200 100 lineto
100 175 moveto
100 200 lineto
25 100 moveto
0 100 lineto
100 25 moveto
100 0 lineto
} ustroke
```



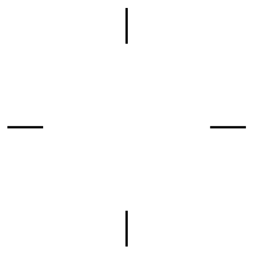
3.1 Path Description

The path description can be represented two ways. The first format, used in the example above, consists of an ASCII path definition enclosed within brackets, ‘{’ and ‘}’. The second format is a two part array containing an array or string of encoded numbers and an string of encoded path construction operators. The operators are executed sequentially with the operands for the operators taken from the encoded number grouping. The encoded format provides a compact representation scheme. Below lies the same user path description as above except that the path description is in an encoded format instead of ASCII format. (The code segment for calculating the trig into a user path at the end of this note provides an example of the encoded format in **C** construct form.)

User Path with Rendering Operator
(Encoded Path Description)

Result

```
[
  [
    0 0 200 200 175 100
    200 100 100 175 100 200
    25 100 0 100 100 25 100 0
  ]
  <000103010301030103>
] ustroke
```



3.2 User Path Construction Operators

The table below provides the allowable path construction operators. The first column lists the operators, the second column the operands for each operator, the third the corresponding encoding value for the operator and the fourth column the NeXT literal definition for the encoding (contained in **dpsNeXT.h**).

User Path Construction Operators and Encodings

Operators	Operands	Encoding	NeXT Encoding
setbbox	*llx lly urx ury*	0	dps_setbbox
moveto	*x y*	1	dps_moveto
rmoveto	*dx dy*	2	dps_rmoveto
lineto	*x y*	3	dps_lineto
rlineto	*dx dy*	4	dps_rlineto
curveto	$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$	5	dps_curveto
rcurveto	$dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$	6	dps_rcurveto
arc	$x\ y\ r\ ang_1\ ang_2$	7	dps_arc
arcn	$x\ y\ r\ ang_1\ ang_2$	8	dps_arcn
arct	$x_1\ y_1\ x_2\ y_2\ r$	9	dps_arct
closepath		10	dps_closepath

All of the path construction operators should be familiar except for **setbbox**, **arct** and **ucache**. The **ucache** operator is optional but should appear as the first operator in the path description if included. The user path cache is analogous to the font cache in that it retains the results of interpreting the path. If the PostScript interpreter encounters a user path that is already in the cache, it substitutes the cached results instead of reinterpreting the path definition. Additional processing is required to place the path in the cache so caching should only be used for paths that are rendered frequently. Although caching works with translations of user paths, it does not work for rotating and scaling user paths. In these instances, the path will have to be reinterpreted and recached.

The **setbbox** operator is required and should immediately follow the **ucache** operator (or appear as the first operator if **ucache** is not used). The **setbbox** operator requires four operands which comprise the bounding box enclosing the entire path. The operands specify the lower left and upper right coordinates of the bounding box and *not* the NeXT representation of a rectangle as an origin and followed by a size. All coordinates specified as operands for successive operators should lie within this bounding box. A **rangecheck** error will be generated if they do not. The inclusion of a bounding box reduces the number of calculations the interpreter must perform and improves performance for rendering the path.

In this particular case, a bounding box that is closer to the actual bounding box performs slightly better than a bounding box that is larger but the difference is not too significant. Increasing the bounding box for the **setbbox** operator by a thousand points in each direction for the fourth method of drawing in the **Dial** application only increased the time of execution by 4-5 milliseconds off a base time of approximately 200 milliseconds. The bounding box values, however, can be used by the interpreter to determine whether an image lies within a clipping region. In these types of cases, a closer approximation to the actual bounding box can mean the difference between needlessly imaging and not imaging at all.

The **arct** operator is a user path replacement operator for **arcto**. The operators are identical except that **arct** does not push any results on the operand stack whereas **arcto** pushes four numbers. Because **arcto** pushes results on the stack it cannot be used in user path definitions.

4. USER PATH RENDERING OPERATORS

The Display PostScript system extensions to the PostScript language include operators that interpret and operate on user paths. These operators are listed below.

Rendering Operators

ufill	inufill
ueofill	inueofill
ustroke	inustroke
ustrokepath	
uappend	

The operators in the second column are used for hit detection and are discussed in Technical Note #5057. The operators in the first column, except for **uappend**, perform the same functions on a user path that their corresponding operators perform on regular paths. For example, the **ufill** operator takes the user path off the operand stack, interprets it and then paints the area enclosed by the user path with the current color.

The **uappend** operator interprets a user path definition and appends the result to the current path in the graphics state. The **uappend** operator has some overhead associated with it as one of the times in the examples shows so it should be used with some consideration.

User path rendering operators make a temporary adjustment to the user space by rounding the origin components to the nearest integer values. This ensures that a single user path description produces uniform results regardless of its position on the page or display due to translation of the user space.

4.1 Clipping with a User Path

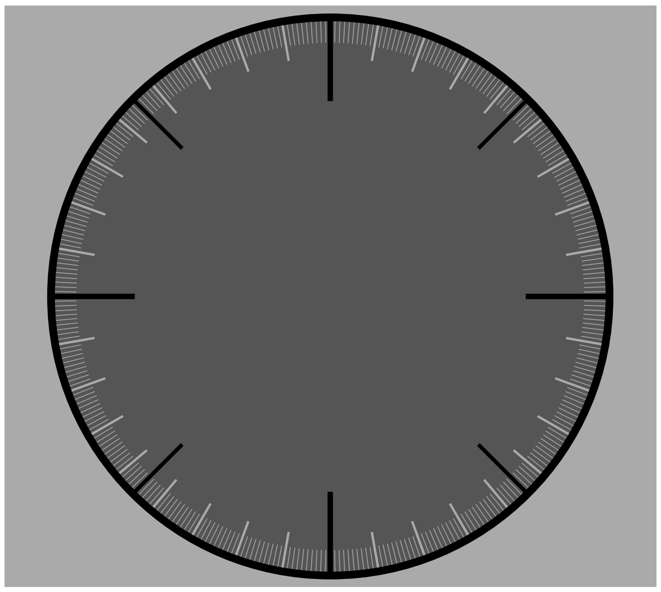
The table of rendering operators does not include a **uclip** operator. Clipping with a user path can be done by using the **uappend** operator to append the path to the current path and then the **clip** operator to clip the current path. The code example below shows this sequence.

```
newpath <user path> uappend clip newpath
```

5. DIAL APPLICATION

The **Dial** application draws a dial with hash marks appearing around its inner edge. Hash marks are optionally drawn at 90, 45, 10 and 1 degree rotations. Six methods are used to draw the marks. Times are available so that comparisons can be made between the types of drawing and the number of hash marks drawn. In the code segments shown, the wraps are shown first followed by their invocations in the **DrawView** class methods.

In these examples, **CLR1** and **WID1** are literals for the line color and line width for the particular type of hash mark highlighted. The values are **NX_LTGRAY** and 0.5, respectively. The variable, **maxdim**, is an instance variable in the **DialView** class that is the diameter of the dial. The literal, **LEN1**, is the scale factor for the hash mark (10.0/11.0) and **DEG1** is the degrees between hash marks (1.0).



5.1 Rotating the User Space

The first method uses the same line description for all the marks but rotates the user space each time to render the line at a different angle around the dial. The static procedure below rotates the user space by the specified degree and then calls the wrap to perform the **moveto/lineto**. After all the lines have been constructed, the path is then stroked by calling the **PSWStrokePath( )** with the color and line width settings.

PostScript language code:

```
defineps PSWDefs( )
. . .
  /RML { % X1 Y1 X0 Y0 Ang
    rotate moveto lineto
  } bind def
. . .
endps

defineps PSWRotate_MakeLine(float Ang, X0, Y0, X1, Y1)
  X1 Y1 X0 Y0 Ang RML
endps
```

C-language code:

```
static void drawRotateLines(clr, wid, startlen, endlen, deg)
  float clr, wid, startlen, endlen, deg;
{
  int angle;

  for (angle = 0; angle < 360; angle += deg)
    PSWRotate_MakeLine(deg, startlen, 0, endlen, 0);

  PSWStrokePath(clr, wid);
}

- drawRotate
{
. . .
   drawRotateLines(CLR1, WID1, maxdim * LEN1, maxdim, DEG1);
. . .
}
```

Rotate

Number of lines	**4**	**8**	**36**	**360**
Display Time	17	26	67	575

An advantage to this type of drawing is that it can be easier to implement than some of the others. One static path description can be used for all the drawings within a set. With only a few paths and rotations, the difference in performance is not significant so it can be an acceptable method of drawing.

5.2 Rotating the User Space and Appending a User Path

The second technique follows a similar approach except that the line description is sent as an encoded user path and then appended to the current path with the **uappend** operator. The **uappend** operator is used instead of **ustroke** because **uappend** only interprets the user path but does not render it. The **ustroke** operator renders the path and so it loses the advantages of delaying the stroking until all the paths have been constructed. In this example, **ustroke** would perform about 60% *worse* than **uappend**.

PostScript language code:

```
defineps PSWDefs()
. . .
  /RUA { % [ Pts (Ops)] Ang
    rotate uappend
  } bind def
. . .
endps

defineps PSWRotate_UAppend(
    float Ang; float Pts[Tot_Pts]; int Tot_Pts;
    char Ops[Tot_Ops]; int Tot_Ops)
  [Pts (Ops)]  Ang RUA
endps
```

C-language code:

```
static void drawRotateUserPathLines(
    pts, ops, clr, wid, startlen, endlen, deg)
  float   pts[];
  char    ops[];
  float   clr, wid, startlen, endlen, deg;
{
  int  angle;

  pts[0] = pts[4] = startlen;
  pts[2] = pts[6] = endlen;
  pts[1] = -wid/2;
  pts[3] = wid/2;
  pts[5] = pts[7] = 0;
  PSsetgray(clr);
  PSsetlinewidth(wid);
  for (angle = 0; angle < 360; angle += deg)
    PSWRotate_UAppend(deg, pts, 8, ops, 3);
  PSstroke();
}

- drawRotateUserPaths
{
  . . .
  ops[0] = dps_setbbox;  ops[1] = dps_moveto;  ops[2] = dps_lineto;
  . . .
  drawRotateUserPathLines(
    pts, ops, CLR1, WID1, maxdim * LEN1, maxdim, DEG1);
  . . .
}
```

Rotate/User Paths

Number of lines	**4**	**8**	**36**	**360**
Display Time	22	40	114	1059

The static procedure, **drawRotateUserPathLines( )**, has the same **for** loop as the first method except that a user path is sent to the wrap instead of two coordinates. The user path description simply consists of the **setbbox** operator followed by the **moveto** and **lineto** operators. The operators are placed in the **ops** array in **drawRotateUserPaths( )** because they will not change for all four invocations of the static procedure. The **pts** array is populated in the static procedure because the bounding box and the coordinates will vary according to the width and length of the particular set of lines drawn.

The times for this method are considerably longer than the previous method. One of the reasons is the overhead that **uappend** requires. Each hash mark must be appended separately because of the rotation. User paths are faster than conventional drawing because they reduce the number of operators that must be interpreted. In this case, the reduction is not sufficient enough to offset the overhead of **uappend**.

Does this example show that rotating a user path is a bad idea? For this case, the answer is yes. For another case the answer might be quite the opposite. For example, let's assume we had some type of intricate dial or clock hand. In this instance, the size of the path and its preexistence might justify placing the description in an encoded user path and then rotating the user space just before rendering the path. In this case, it would not be necessary to append the hand to the current path with the **uappend** operator before stroking or filling. Instead, **ustroke** or **ufill** could be called directly. Technical note #5055 will take a further look at this issue with a description of a clock application that makes use of graphic states, user objects and user paths defined in the server.

5.3 Calculating the Trigonometry

The third method performs trigonometric calculations on the client side to find the precise coordinates for **moveto**/**lineto** couplings.

PostScript language code:

```
defineps PSWDefs()
. . .
  /ML { % X1 Y1 X0 Y0
    moveto lineto
  } bind def
. . .
endps

defineps PSWMakeLine(float X0, Y0, X1, Y1)
  X1 Y1 X0 Y0 ML
endps
```

C-language code:

```
static void drawTrigLines(clr, wid, x, y, startlen, endlen, deg)
  float clr, wid, x, y, startlen, endlen, deg;
{
  int  angle;

  for (angle = 0; angle < 360; angle += deg)
    PSWMakeLine(x + (float) cos(angle * RADIANS) * startlen,
        y + (float) sin(angle * RADIANS) * startlen,
        x + (float) cos(angle * RADIANS) * endlen,
        y + (float) sin(angle * RADIANS) * endlen);

  PSWStrokePath(clr, wid);
}

- drawTrig
{
. . .
    drawTrigLines(CLR1, WID1, viewcenter.x, viewcenter.y,
        maxdim * LEN1, maxdim, DEG1);
. . .
}
```

Trigonometry

Number of lines	**4**	**8**	**36**	**360**
Display Time	13	21	73	500

The times given by this method show little real advantage over the first method, rotation of a static path description, except when rendering a large number of hash marks. This comparison says that rotating the user space has some overhead associated with it but that it is not too significant at low frequencies. A disadvantage of this example is that it can begin to get a bit complex. This test case uses one of the simplest paths possible, a single line segment. A path with multiple line segments and curves and arcs could be difficult to process. Unless the performance advantage outweighs the complexity, rotating the user space might be preferred over calculating the positions on the client side.

5.4 User Paths

The fourth method takes the trigonometric approach one step further by placing the **moveto**/**lineto** path constructions into a single, large user path and then executing **ustroke**. In this case, the array is populated on the fly and then sent to a wrap with a dynamic index size. A static description for each set of hash marks could just as well be retained as a user path in static arrays eliminating the need for the **for** loop and the calculations. The ease of calculating the marks around the dial, however, makes it more convenient to calculate the points in a loop than it does to save the user paths.

PostScript language code:

```
defineps PSWUStroke(
    float Color, Width; float Pts[Tot_Pts]; int Tot_Pts;
  char Ops[Tot_Ops]; int Tot_Ops)
  Color  setgray Width setlinewidth
  [Pts (Ops)] ustroke
endps
```

C-language code:

```
static void drawTrigUserPathLines(
    pts, ops, clr, wid, x, y, startlen, endlen, deg)
  float pts[];
  char ops[];
  float  clr, wid, x, y, startlen, endlen, deg;
{
  int i , j, angle;

  i = 4; j = 1;
  for (angle = 0; angle < 360; angle += deg)
  {
    pts[i++] = x + (float) cos(angle * RADIANS) * startlen;
    pts[i++] = y + (float) sin(angle * RADIANS) * startlen;
    ops[j++] = dps_moveto;

    pts[i++] = x + (float) cos(angle * RADIANS) * endlen;
    pts[i++] = y + (float) sin(angle * RADIANS) * endlen;
    ops[j++] = dps_lineto;
  }
  PSWUStroke(clr, wid, pts, i, ops, j);
}

- drawTrigUserPaths
{
  pts[0] = bounds.origin.x;
  pts[1] = bounds.origin.y;
  pts[2] = bounds.origin.x + bounds.size.width;
  pts[3] = bounds.origin.y + bounds.size.height;
  ops[0] = dps_setbbox;
  . . .
  drawTrigUserPathLines(
      pts, ops, CLR1, WID1, viewcenter.x, viewcenter.y,
      maxdim * LEN1, maxdim, DEG1);
  . . .
}
```

User Paths

Number of lines	**4**	**8**	**36**	**360**
Display Time	12	16	49	245

The resulting times of a user path is significantly faster for drawing as little as 60 lines. The benefit is derived from the elimination of interpreter loop overhead during the path construction. User paths combine a restricted data format with an optimized rendering pipeline to eliminate much of the data manipulation that path construction operators such as **moveto** and **lineto** perform. For drawing 8 hash marks, the total processing time is not large enough to produce a noticable difference between user paths and **moveto**/**lineto** wraps. For 48 hash marks, however, user paths represent a time savings of almost 30% over the other method (100 milliseconds versus 70 milliseconds). Unlike the previous method, using user paths to reduce the number of operators greatly outweighs the complexity of performing the trigonometry.

5.5 User Path Retained in the Server

This method is almost identical to the previous method except that the user paths for each type of hash mark are created and stored in the server when **DialView** is instantiated. When drawing the hash marks, only the names of the user paths are sent to the server instead of the user paths themselves. For drawing 360 hash marks, this method has about a 45% time savings over calculating the trig and sending the user paths each time. Storing user paths in the server is an attractive option for drawing objects that appear frequently.

PostScript language code:

```
defineps PSWDefineUserPath(
    float Pts[Tot_Pts]; int Tot_Pts;
    char Ops[Tot_Ops]; int Tot_Ops; char *str)
  /str [Pts (Ops)] def
endps

defineps PSWDrawUserPath (float Color, Width; char *str)
  Color  setgray Width setlinewidth
  str ustroke
endps
```

C-language code:

```
/* This definition is for the name of the user path array in */
/* the server. A user object could have been used as well. */
static char *upath1 = {"upath1"};

/* Calculate the start and end points and place in user path */
/* format. Send the user path to the server and define it in */
/* the server. */

static void setupTrigUserPath(
    pts, ops, x, y, startlen, endlen, deg, upathname)
  floatpts[]; charops[];
  float x, y, startlen, endlen, deg;
  char*upathname;
{
  inti , j, angle;

  i = 4; j = 1;
  for (angle = 0; angle < 360; angle += deg)
  {
    pts[i++] = x + (float) cos(angle * RADIANS) * startlen;
    pts[i++] = y + (float) sin(angle * RADIANS) * startlen;
    ops[j++] = dps_moveto;
    pts[i++] = x + (float) cos(angle * RADIANS) * endlen;
    pts[i++] = y + (float) sin(angle * RADIANS) * endlen;
    ops[j++] = dps_lineto;
  }
  PSWDefineUserPath(pts, i, ops, j, upathname );
}

/* This method is messaged from the newFrame method at object */
/* instantiation. The user paths are created and then defined */
/* in the server. */

- setupUserPaths
{
  pts[0] = bounds.origin.x;
  pts[1] = bounds.origin.y;
  pts[2] = bounds.origin.x + bounds.size.width;
  pts[3] = bounds.origin.y + bounds.size.height;
  ops[0] = dps_setbbox;
  setupTrigUserPath(pts, ops, viewcenter.x, viewcenter.y,
      maxdim * LEN1, maxdim, DEG1, upath1);
  . . .
  return self;
}

/* This is the method to draw the user path. It simply sends */
/* the line color and width as well as the name of the user */
/* path in the server. */

- drawTrigUserPathsServer
{
    PSWDrawUserPath(CLR1, WID1, upath1);
. . .
}
```

.

User Paths Retained in the Server

Number of lines	**4**	**8**	**36**	**360**
Display Time	10	15	36	128

5.6 DPSDoUserPath( )

In this method of drawing, the user path is sent to the server with the NeXT procedure, **DPSDoUserPath( )**. The calling sequence for this procedure is shown below. (The NeXT documentation should be referred to for the complete specification.)

```
void DPSDoUserPath(
     void *coords, int numCoords, DPSNumberFormat numType,
     char *ops, int numOps, void *bbox, int action)
```

The *coords* argument is a pointer to an array of coordinate points with *numCoords* identifying the number in the array. The *numType* argument specifies the type of the numbers used in the data string – the three identifiers available are **dps_float**, **dp_long** and **dps_short**. The *ops* argument is a pointer to a character array or string containing the user path construction operators with the *numOps* argument identifying the number of operators in the array or string. The *bbox* argument is a pointer to a four element number array that contains the values for the setbbox operator. The *action* argument is an identifier for a user path rendering operator. Examples are **dps_ufill** and **dps_ustroke**.

Note: *The bbox argument should describe the bounding box of the user path by specifying the lower left and the upper right coordinates of the bounding box. This representation is different from the NeXT rectangle scheme of an origin and followed by a size.*

C-language code:

```
/* Calculate the start and end points and place in user path */
/* format. Send the entire user path at once using the */
/*  DPSDoUserPath( ) call. */

static void drawDPSUserPathLines(
    pts, ops, clr, wid, x, y, startlen, endlen, deg, bbox)
  floatpts[];
  charops[];
  float clr, wid, x, y, startlen, endlen, deg;
  floatbbox[4];
{
  inti , j, angle;

  i = j = 0;
  for (angle = 0; angle < 360; angle += deg)
  {
    pts[i++] = x + (float) cos(angle * RADIANS) * startlen;
    pts[i++] = y + (float) sin(angle * RADIANS) * startlen;
    ops[j++] = dps_moveto;

    pts[i++] = x + (float) cos(angle * RADIANS) * endlen;
    pts[i++] = y + (float) sin(angle * RADIANS) * endlen;
    ops[j++] = dps_lineto;
  }

  PSsetgray(clr);
  PSsetlinewidth(wid);

  DPSDoUserPath(pts, i, dps_float, ops, j, bbox, dps_ustroke);
}


- drawDPSUserPaths:(int) cell;
{
  floatbbox[4];
  . . .
    bbox[0] = bounds.origin.x;
    bbox[1] = bounds.origin.y;
    bbox[2] = bounds.origin.x + bounds.size.width;
    bbox[3] = bounds.origin.y + bounds.size.height;
    . . .
    drawDPSUserPathLines(
        pts, ops, CLR1, WID1, viewcenter.x, viewcenter.y,
        maxdim * LEN1, maxdim, DEG1, bbox);
  . . .
. . .
}
```

For drawing 360 hash marks, sending the user path with **DPSDoUserPath( )** is approximately 20% faster than using a custom wrap. The reason for the difference is that **DPSDoUserPath( )** places the coordinate points into binary encoded number arrays. The server can process this number representation more efficiently than a non-encoded array.

In addition, **DPSDoUserPath( )** automatically provides an emulation of user paths. This is necessary for printing to devices that do not contain the PostScript language extensions for the Display PostScript system.

DPSDoUserPath( )

Number of lines	**4**	**8**	**36**	**360**
Display Time	13	18	42	190

6. SUMMARY

The main conclusion that can be drawn from the examples in the **Dial** application is that user paths can have a large performance improvement for certain types of paths. These types of paths can include large paths or preexisting ones. The benefit arises from the elimination of the interpreter from the path construction.

Defining the user path in the server can be an efficient way to draw items that appear frequently. The NeXT AppKit procedure call, **DPSDoUserPath( )**, is useful because it provides an easy and highly efficient way to send user paths to the server and because it provides a user path emulation for printing. This emulation is necessary for printing to devices that do not contain the PostScript language extensions for the Display PostScript system.

The table below displays times, in milliseconds, for drawing hash marks using the six methods in the **Dial** application.

Times from the Dial Application

Method (# lines)	4	8	36	360
Rotate	17	26	67	575
Rotate/Uappend	22	40	114	1059
Trigonometry	13	21	73	500
User Paths	12	16	49	245
User Paths in Server	10	15	36	128
DPSDoUserPath( )	13	18	42	190

The examples that show slower results should not necessarily be discounted. For types of drawing different from that shown with the dial, these methods might be preferred.