POSTSCRIPT®

THE DISPLAY POSTSCRIPT® SYSTEM: IMAGE CONSISTENCY at SMALL SIZES and the CONTROL POINT APPLICATION

Technical Note #5056

July 3, 1990
PostScript® Developer Support Group

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

PN LPS5056

THE DISPLAY POSTSCRIPT® SYSTEM: IMAGE CONSISTENCY at SMALL SIZES and the CONTROL POINT APPLICATION

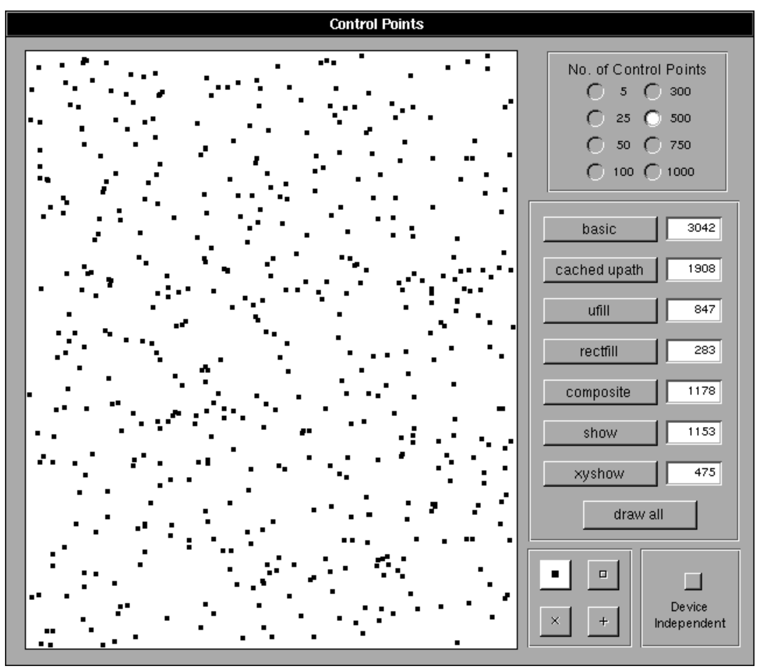
Technical Note #5056

July 3, 1990
PostScript® Developer Support Group
(415) 961-4111

1. INTRODUCTION

Previous technical notes have looked at variations of drawing lines either randomly or around a dial. Each variation in these notes is timed and then the results discussed and compared. This technical note takes a similar approach by using the **ControlPoint** application to look at and compare seven different ways to display control points (or any arbitrary shape).

In this case, however, the issue is not only the display time for each method, but also the consistency between each of the control points and the device independence of each method. Control points are typically very small; the ones used in this application are five points across. At such small sizes, a one pixel variation in how a path maps from user to device space can have a significant impact on appearance. Steps can be taken to ensure that control points appear the same but some of these are device dependent. The goal of this document is to introduce techniques that will not only display a fairly large set of small images efficiently but also consistently and device independently.



This technical note looks at seven variations of displaying up to 1000 randomly selected control points. The control points can be displayed in four different shapes - a filled square, an open square, a cross and an x. The areas highlighted for each method include the display time, the algorithm to display *x* number of points, the control point description, the consistency between two or more points and the device independence of the point.

The seven approaches used to draw the control points are listed below:

- *basic drawing* — uses **rmoveto**'s and **rlineto**'s within wraps
- *rect operations* — passes an array of rectangles to the **rectfill** and **rectstroke** operators (only for rectangular control points)
- *user paths with cache* — represents each control point as a cached user path, translating to each point location before rendering the path
- *user paths* — places a set of user path subpaths (one for each control point) into a large array and then executes a single **ufill** or **ustroke**
- *compositing a bit map* — draws the control point into a bit map and then composites the image at each control point location
- *show* — takes advantage of the font caching mechanism by turning the control points into a Type 3 font program and using the **show** operator to display the points
- *xyshow* — same as the previous method except that the **xyshow** operator is used instead of **show** operator

The fastest, simplest to use and most consistent methods are the rect operations and the **xyshow** operator. Both methods are significantly faster (by at least a factor of two) than the other methods. Drawing 500 squares takes approximately 200 milliseconds using **rectfill**, 400 milliseconds using **xyshow** and 1000 milliseconds using the next closest approach, compositing. The calling sequence for these methods is much simpler and the ability to display consistent control points across different device resolutions far surpasses the other methods. For drawing rectangles, the **rectfill** and **rectstroke** operators are recommended. For drawing any other shapes, the **xyshow** operator is recommended. (The **rectstroke** operator is slightly slower than **xyshow** for a stroked rectangle but the convenience of **rectstroke** outweighs the minor performance difference.)

Both methods have a few drawbacks. The rect operators can only be used for rectangular control points. Circles, stars, crosses and x's cannot be represented. In addition, both methods can display only a one color image at a time. If multi-colored control points are desired, the methods must be repeated with a different color and shape. Compositing is more than twice as slow as the rect and **xyshow** operators but allows for multi-colored and hand-tuned control points. But in doing so, different bitmap images must be used for different display resolutions.

The other four methods shown, basic drawing wraps, user paths caching, user path descriptions in large arrays and **show** operations, are not particularly suited for displaying a large number of control points. They are poorer in performance and require greater complexity than the three highlighted in the paragraph above. In addition, these methods also have problems producing suitable images at extremely small point sizes.

2. CONSISTENCY AND DEVICE INDEPENDENCE

Consistency of rendering and device independence are closely related. Consistency means that successive instances of the same control point or character or similar object appear the same regardless of the actual point location in device space. Device independence means that the same description will produce the same results on different resolution displays. At low resolutions and at small point sizes, these two issues are paramount. The Type 1 font format implements a hinting mechanism to produce consistent rasterizations in these instances. Other techniques outside the Type 1 font algorithms have a similar but reduced capability for simple renderings.

Lack of consistency in drawing is readily noticeable since it can be seen on a single display. Lack of device independence is more subtle since it takes two devices with different resolutions to become apparent. Although device independence in regards to display images receives little attention at the moment, it is bound to become more and more important in the future as different displays become available.

Inconsistency
(*same display, different locations*)

Device Dependence
(different displays, same locations)

The same description may produce different results because of the paths fall at different locations within a pixel in device space

The different mappings from user space to device space turns on different pixels.

In the **ControlPoint** application, an array is created at initialization time that contains 1000 point locations. These locations are selected to randomly fall at different locations within a pixel. Since the control points are 5 points or less across, some care must be taken to ensure that the points are displayed consistently.

In the case of the NeXT MegaPixel display, one way to have the points appear consistently is to round the control point locations to the nearest integer. This step will work for the MegaPixel display because the one-to-one mapping from the default user space to device space. This one-to-one correspondence means that a point location in user space maps to the same location in device space. Relying on this default mapping, though, is discouraged. In the first place, scaling or other transformations of the user space will disrupt this mapping. In the second, other displays, both current and future, will have different resolutions and different mappings. Casting the drawing to one type of display will produce problems when the drawing is performed on a different resolution display.

Some PostScript language operators perform a rounding adjustment automatically but round to device space and not to user space. The rect operators and the character show operators are two such examples. This rounding to device space produces consistent rectangles and letters regardless of where the current point falls within a pixel. In the case of characters, the letters are consistent but not necessarily of good form. The path locations may turn on undesired pixels. Type 1 font programs employ hints to constrain the paths and produce readable characters. Type 3 font programs can incorporate minor adjustments to produce a similar hinting capability.

The user path operators also round to device space but round the entire drawing and not each subpath. User path operators such as **ufill** and **ustroke** perform a temporary adjustment to the current transformation matrix by rounding the origin to the nearest device boundary. This adjustment ensures that multiple renderings of the same path will appear the same when the paths are placed at different locations by translating the user space. The method in this technical note that uses the **ucache** operator to cache a single control point description benefits from this adjustment. The other user path method, which puts the control point descriptions into a large array, does not. In this second instance, the change in position is done explicitly and not through translation. As a result, the subpaths appear in their original locations.

By the nature of its operation, compositing a bit map aligns the image to pixel boundaries producing a consistent image. This image, though, is tied to one resolution and is not device independent.

Another technique to produce an acceptable control point for a single display is to hand tune the points in the path construction until an acceptable looking control point is produced. Adjusting the numbers by fractions of points can produce the desired control point. Problems arise, though, when these numbers are used on a different display with a different mapping to device space. The numbers for one device may not match another device and so they produce inexact renderings.

A better solution where possible is to round to device space so as to always align the paths in the proper place with respect to pixel boundaries. The basic drawing and font methods show some techniques for doing this. Note that these methods use an algorithmic solution that is not tied to a particular device. In the case of basic drawing, the rounding is very expensive — in fact, it doubles the display time. In the case of using a font, the description is only executed once, and the font machinery caches the resulting image. As a result, the roundings performed in the character description have no impact on performance. Rounding to device space, though, should only be used when it is necessary. A case where this is warranted is for extremely small and exacting images. Larger and infrequently displayed images will not benefit as much from this exact positioning to device space.

3. THE CONTROLPOINT APPLICATION

The **ControlPoint** application demonstrates various techniques for drawing small control point shapes, and measures the speed of each technique. The rest of this note examines how these techniques were implemented in Objective-C™ on the NeXT computer.

The application creates an array of random locations at which to draw control points at initialization time. The user can display and time the drawing of a set of 5 to 1000 points. Four different types of control points can be displayed. Each method uses a different way to describe the individual control points as well as a different algorithm for displaying the points. For each of the seven methods addressed, the control point drawing instructions, the display algorithm and the PostScript language trace (the operations performed by the server) are listed. A good corollary to using time as an indication of the performance of a method is to look at the complexity of the source and the size of the trace. The three more favorable methods have the simplest calling sequence as well as the simplest and smallest traces. Reducing the amount of information sent to the server reduces both the transmission time and the processing time.

There are several data structures used by all the methods. The first variable, **XYArray**, is an array that holds 2000 coordinate values (randomly selected to fall within the **ControlView**, the view that displays the points). Each consecutive pair of numbers comprise a control point location. It is these values that are sent in the wraps or placed in the arrays as the point locations for the methods below. An instance variable, **indexOfPoints**, provides the starting index into the array. The instance variable, **numberOfPoints**, contains the number of points to be displayed. This value is obtained from the tag of the selected radio button for the *No. of Control Points* control in the interface.

Two other arrays are used by some of the methods to send large amounts of data to the server. The array, **XYBuffer**, holds any float numbers sent to the server. **XYBuffer** is used for the user path methods, the rect method and the **xyshow** method to send the control point locations. Another array, **OpsBuffer**, holds the path construction operators for the user path method as well as the character values for the **xyshow** method.

3.1 Basic Drawing

This method displays the control points by making calls to wraps. One call is made for each point. Four different PostScript language procedures are used, each one describing a different type of control point. Only the procedure for the current shape is called by the method.

Below lies the wraps for the control points as well as two procedures to adjust the point locations to device space, **SA** and **RSA**. The **PSWSetIndependent( )** and **PSWSetDependent( )** wraps activate and deactivate these two procedures. The wraps themselves are invoked when the device independent button is toggled in the ControlPoint interface. When the button is selected, the procedures are executed. When the button is not selected, the dummy procedure, **NOP**, is executed.

PostScript language code:

```
/* This wrap is called in the +new method to define and bind the
 * procedures. The procedures are invoked within another wrap.
 * The first 3 procedures, NOP,SA and RSA, are used to adjust
 * the point positions to consistent locations in device space.
 */
defineps PSWDefsContPts ()
  /NOP { } def

  /SA { % x y  sa  x' y'
    transform
    0.25 sub round 0.25 add exch
    0.25 sub round 0.25 add exch
    itransform
  } bind def

  /RSA { %dx dy  rsa  dx' dy'
    dtransform
    round exch
    round exch
    idtransform
  } bind def

  /BRF{ %X Y
    sa moveto -1.5 -1.5 rsa rmoveto  0 3 rsa rlineto
    3 0 rsa rlineto 0 -3 rsa rlineto -3 0 rsa rlineto
    closepath
  } bind def

  /BRS{ %X Y
    sa moveto -2 -2 rsa rmoveto  0 4 rsa rlineto
    4 0 rsa rlineto 0 -4 rsa rlineto -4 0 rsa rlineto
    closepath
  } bind def

  /BX { % X Y
    sa moveto -2 -2 rsa rmoveto 4 4 rsa rlineto
    0 -4 rsa rmoveto -4 4 rsa rlineto
  } bind def

  /BC{ % X Y
    sa moveto
    0 2 rsa rmoveto  0 -4 rsa rlineto
    -2 2 rsa rmoveto  4 0 rsa rlineto
  }  bind def
endps

defineps PSWSetIndependent ( )
  /sa /SA load def
  /rsa /RSA load def
endps

defineps PSWSetDependent ( )
  /sa /NOP load def
  /rsa /NOP load def
endps
```

Below lies the algorithm used to display the points for this method. This algorithm makes use of a wrap call, **PSWBasic( )**, that is passed the name of one of the four procedures above as well as either a **fill** or **stroke** painting operator. The procedure name and operator to use are obtained with the **getBasicProc** and **getBasicOp** methods. These methods return pointers to the character strings of the procedure and the painting operators. The local variable **basicProc** will point to either “BRF”, “BRS”, “BX” or “BC”. The local variable **basicOp** will point to either “fill” or “stroke”. (Other ways to invoke these procedures are possible and probably preferred but are not used here in order to use a single approach across all methods.)

PostScript language code:

```
defineps PSWBasic(float X, Y; char *Figure, *Op)
   X Y Figure Op
endps
```

C-language code:

```
/* The drawing will center around the point passed in. */
- drawBasic:(int)cell
{
  int   i;

  char *basicProc, *basicOp;

  basicProc = [controlPoint  getBasicProc];
  basicOp = [controlPoint  getBasicOp];
  . . .
    PSWEraseView();
    PSsetlinewidth(0.15);

    for (i = indexOfPoints; i < indexOfPoints + (numberOfPoints*2);
         i = i+2)
      PSWBasic(XYPoints[i], XYPoints[i+1], basicProc, basicOp);
  . . .

  return self;
}
```

PostScript language trace for 25 points:
(These are the PostScript language operations that are sent to and executed by the server.)

```
113.72  382.64 BRF fill
335.57  94.79 BRF fill
176.63  84.93 BRF fill
76.77  56.66 BRF fill
71.36  135.61 BRF fill
24.02  82.15 BRF fill
63.11  356.97 BRF fill
215.25  206.47 BRF fill
161.55  30.73 BRF fill
361.98  166.83 BRF fill
54.32  322.76 BRF fill
412.14  395.39 BRF fill
107.63  125.97 BRF fill
333.61  322.91 BRF fill
222.55  154.92 BRF fill
282.94  466.14 BRF fill
400.16  320.25 BRF fill
176.22  403.91 BRF fill
156.44  490.60 BRF fill
61.06  303.79 BRF fill
378.83  421.52 BRF fill
22.02  379.82 BRF fill
318.11  456.69 BRF fill
329.33  125.83 BRF fill
128.72  204.49 BRF fill
```

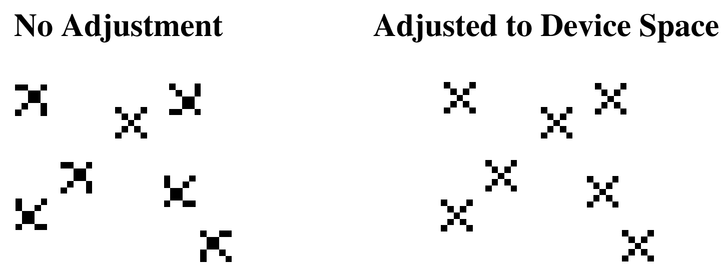
Conventional path construction operators are not really recommended for this type of drawing. As a comparison of the times will show, this is the slowest of all the methods. There are two reasons for this: each wrap incurs overhead, and the wraps themselves contain many PostScript operations. These two features combined generate a lot of unnecessary setup, data formatting and operator execution.

Basic Drawing

	No Adjustment	Adjusted
■ Filled Rectangle	2896	5218
□ Open Rectangle	2217	4541
+ Cross	1797	3767
× X	1852	5188

(500 points)

In addition, this method has difficulty producing consistent images unless some expensive adjustment procedures are performed. Without the adjustments, the random location of the current point within a pixel causes pixel variations across images. The procedures used to eliminate this problem by rounding to device space produce a consistent and device independent solution, but it is at the expense of performance. Rounding to device space doubles the display time, a significant factor since this method is already the slowest of the ones explored.



3.2 User Paths

Although user paths draw much of the focus and recommendation in Technical Note #5054, *Path Construction & Rendering and the Dial Application*, they do not perform as well here as other some of the other methods for drawing control points. User paths are suited for large, existing paths as well as dynamically created paths. The paths can be retained in a static array and passed to directly to the server in the first case or translated from the application data structure to user paths on the fly in the second case. Both instances are very straight-forward and efficient uses of user paths.

In this note, we look at two cases of user paths. The first makes use of the user path cache by taking a single description and translating it to each control point location before rendering. The second places a set of user path descriptions for control points into a large array and performs a single **ufill** or **ustroke** operation (for a large number of control points several waves are necessary). Both instances fail to provide an acceptable method for drawing control points both from a performance and an image quality standpoint. Drawing control points requires that the same description be drawn in a number of places. Translating to each control point location or stuffing a description for each control point into an array is not an efficient approach. The amount of data that is sent to the server and the limited ability to adjust the small sizes to device space makes these two methods less attractive than some of the others.

User Path Descriptions

Both cases use the same userpath descriptions for the points. These descriptions are stored as static float and character arrays in one of the application files. The first number in each array is the number of entries contained in the array. The information from these arrays is supplemented with other information and placed into other arrays before transmission to the server. (The supplemental information establishes a current point for these descriptions.)

C-language code:

```
static floatptsRectfill[] = {8, -2, -2, 0, 4, 4, 0, 0, -4};
static charopsRectfill[] = {5, dps_rmoveto, dps_rlineto,
    dps_rlineto, dps_rlineto, dps_closepath};

static floatptsRectstroke[] = {8, -2, -2, 0, 4, 4, 0, 0, -4};
static charopsRectstroke[] = {5, dps_rmoveto, dps_rlineto,
    dps_rlineto, dps_rlineto, dps_closepath};

static floatptsX[] = {8,  -2, -2, 4, 4, 0, -4, -4, 4};
static charopsX[] = {4, dps_rmoveto, dps_rlineto, dps_rmoveto,
    dps_rlineto};

static floatptsCross[] = {8, 0, 2, 0, -4, -2, 2, 4, 0};
static charopsCross[] = {4, dps_rmoveto, dps_rlineto, dps_rmoveto,
    dps_rlineto};
```

User Path Cache

In this method, a single generic control point is created that is centered around the location (0,0). The user space origin is then translated to each control point location before rendering. The operands for the generic user path description – the bounding box, current point and path values – are placed in **XYBuffer**. The literals for the **ucache**, **setbbox**, **moveto** and path construction operators for the same point are placed in **OpsBuffer**.

In the code segment below, the **getUserPtsArray**, **getUserOpsArray** and **getUserOp** methods return the addresses of the user path description and the user path rendering operator. The variable **userPtsArray** holds the address of one of the four float arrays above. The variable **userOpsArray** holds the address of one of the four character arrays above. The variable **userOp** points to the character string “ufill” or “ustroke”. **FIGURESIZE** is the width and height of the largest control point, 8 points.

The user space is translated by performing a translate that is relative to the last translate performed. This relative translate is faster than an absolute translate encapsulated within a **gsave/grestore** nesting.

PostScript language code:

```
defineps PSWUserPath (float Pts[Tot_Pts]; int Tot_Pts;
     char Ops[Tot_Ops]; int Tot_Ops; char *Op)
  [Pts (Ops)] Op
endps
```

C-language code:

```
- drawUserCache:(int)cell
{
  int    i, i_pt, i_op, j;
  char   *userOp, *userOpsArray;
  float  *userPtsArray;

  userPtsArray = [controlPoint  getUserPtsArray];
  userOpsArray = [controlPoint  getUserOpsArray];
  userOp = [controlPoint  getUserOp];

  . . .
  /* Places a user path description for a generic control
   * point into the OpsBuffer and XYBuffer.  */
  i_op = i_pt = 0;
  OpsBuffer[i_op++] = dps_ucache;

  XYBuffer[i_pt++] = -FIGURESIZE/2;
  XYBuffer[i_pt++] = -FIGURESIZE/2;
  XYBuffer[i_pt++] = FIGURESIZE/2;
  XYBuffer[i_pt++] = FIGURESIZE/2;
  OpsBuffer[i_op++] = dps_setbbox;

  XYBuffer[i_pt++] = 0;  XYBuffer[i_pt++] = 0;
  OpsBuffer[i_op++] = dps_moveto;

  for (j = 1; j <= userPtsArray[0]; j++)
    XYBuffer[i_pt++] = userPtsArray[j];

  for (j = 1; j <= (int) userOpsArray[0]; j++)
    OpsBuffer[i_op++] = userOpsArray[j];

  . . .
  /* Performs an initial translate to the first location
   * and then performs a relative translation thereafter.  */
  PSgsave();
  PStranslate(XYPoints[indexOfPoints], XYPoints[indexOfPoints+1]);

  for (i = indexOfPoints; i < indexOfPoints + (numberOfPoints*2);
      i = i+2)
  {
    PSWUserPath(XYBuffer, i_pt, OpsBuffer, i_op, userOp);
    PStranslate(XYPoints[i+2] - XYPoints[i],
        XYPoints[i+3] - XYPoints[i+1]);
  }
  PSgrestore();
  . . .
  return self;
}
```

PostScript language trace:

```
113.72  382.64 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
221.85  -287.85 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-158.93  -9.86 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-99.86  -28.26 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-5.41  78.94 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-47.34  -53.45 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
39.09  274.82 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
152.13  -150.5 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-53.69  -175.74 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
200.42  136.10 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-307.66  155.93 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
357.82  72.62 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-304.5  -269.42 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
225.97  196.94 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-111.05  -167.98 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
60.38 311.22 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
117.22 -145.89 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-223.94 83.66 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-19.77 86.69 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-95.38 -186.81 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
317.77 117.73 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-356.82 -41.70 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
296.10 76.86 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
11.22 -330.85 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
-200.61 78.66 translate
[[-4 -4 4 4 0 0 -2 -2 0 4 4 0 0 -4] <0b0001020404040a>] ufill
150.57 131.46 translate
```

The benefit of the user path cache is offset by performing a translate operation and then sending a user path description for each control point. As in the basic drawing case above, a couple server operations are necessary for each control point.

User Path with Cache

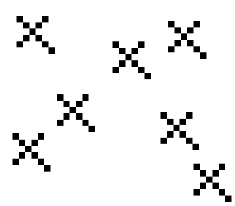
■	Filled Rectangle	1953
□	Open Rectangle	1903
+	Cross	1911
×	X	1918

(500 points)

A consistent image can be produced because the translation before each point adjusts the current transformation matrix uniformly. Unfortunately, device independent adjustments cannot be made to the user path description to produce acceptable images at small sizes. The user path format limits the description to literal numbers and path constructions operators. This restricted format precludes any attempt to round the points to device space with procedures like those used in the basic drawing method above.

No Adjustment

(The user paths start in the same place in device space because of the adjustment to the CTM before each user path rendering operator.)



(The individual path elements, though cannot be adjusted because of the restricted user path format.)

A Single Large User Path

In the second case, putting a set of control points as user paths into a large array requires first placing a **moveto** operation into the array followed by the path description for the point. Not only does this method require two large arrays, one for the points and another for the operators, but it also calls for replicating the same user path descriptions many times in the arrays. The source code and the PostScript language trace are shown below. The source is tortuous and the trace large. These two factors alone signal that it is not an acceptable approach.

The same wrap, **PSWUserPath( )**, used above is used to send the description and rendering operator to the server. The local variables **userPtsArray**, **userOpsArray** and **userOp** also point to the same user path descriptions and operators. The bounding box of the user path is set to the size of the **ControlView**'s bounds and not to the size of a single control point because all of the points will be drawn at their absolute locations. A **moveto** operation is placed into the user path description for each control point and followed by the description of the control point. This process is repeated for every control point.

PostScript language code:

```
defineps PSWUserPath (float Pts[Tot_Pts]; int Tot_Pts;
     char Ops[Tot_Ops]; int Tot_Ops; char *Op)
  [Pts (Ops)] Op
endps
```

C-language code:

```
- drawUserPath:(int)cell
{
  int    i,  i_pt, i_op, j;
  char   *userOp, *userOpsArray;
  float  *userPtsArray;

  userPtsArray = [controlPoint  getUserPtsArray];
  userOpsArray = [controlPoint  getUserOpsArray];
  userOp = [controlPoint  getUserOp];

  . . .
  XYBuffer[0] = bounds.origin.x;
  XYBuffer[1] = bounds.origin.y;
  XYBuffer[2] = bounds.origin.x + bounds.size.width;
  XYBuffer[3] = bounds.origin.y + bounds.size.height;
    psBuffer[0] = dps_setbbox;

  i = 0; i_pt = 4; i_op = 1;
  while (i < numberOfPoints * 2)
  {
    /*
     * This check sends the array to the server if the array
      * limit has been reached.
     */
    if ((i_pt + userPtsArray[0] > MAX_UPATHPTS) ||
         (i_op + (int) userOpsArray[0] > MAX_UPATHOPS))
    {
      PSWUserPath(XYBuffer, i_pt, OpsBuffer, i_op, userOp);
      i_pt = 4; i_op = 1;
    }

    XYBuffer[i_pt++] = XYPoints[indexOfPoints + i++];
    XYBuffer[i_pt++] = XYPoints[indexOfPoints + i++];
    OpsBuffer[i_op++] = dps_moveto;

    for (j = 1; j <= userPtsArray[0]; j++, i_pt++)
      XYBuffer[i_pt] = userPtsArray[j];

    for (j = 1; j <= (int) userOpsArray[0]; j++, i_op++)
      OpsBuffer[i_op] = userOpsArray[j];
  }
  PSWUserPath(XYBuffer, i_pt, OpsBuffer, i_op, userOp);
  . . .

  return self;
}
```

PostScript language trace:

```
[[0 0 423 515 113.72 382.64 -2 -2 0 4 4 0 0 -4 335.57 94.79 -2 -2 0
4 4 0 0 -4 176.63 84.93 -2 -2 0 4 4 0 0 -4 76.77 56.66 -2 -2 0 4 4
0 0 -4 71.36 135.61 -2 -2 0 4 4 0 0 -4 24.02 82.15 -2 -2 0 4 4 0 0
-4 63.11 356.97 -2 -2 0 4 4 0 0 -4 215.25 206.47 -2 -2 0 4 4 0 0 -4
161.55 30.73 -2 -2 0 4 4 0 0 -4 361.98 166.83 -2 -2 0 4 4 0 0 -4
54.32 322.76 -2 -2 0 4 4 0 0 -4 412.14 395.3915 -2 -2 0 4 4 0 0 -4
107.63 125.97 -2 -2 0 4 4 0 0 -4 333.61 322.91 -2 -2 0 4 4 0 0 -4
222.55 154.929993 -2 -2 0 4 4 0 0 -4 282.94 466.14 -2 -2 0 4 4 0 0
-4 400.16 320.25 -2 -2 0 4 4 0 0 -4 176.22 403.91 -2 -2 0 4 4 0 0 -
4 156.44 490.60 -2 -2 0 4 4 0 0 -4 61.06 303.79 -2 -2 0 4 4 0 0 -4
378.83 421.52 -2 -2 0 4 4 0 0 -4 22.02 379.82 -2 -2 0 4 4 0 0 -4
318.11 456.69 -2 -2 0 4 4 0 0 -4 329.33 125.83 -2 -2 0 4 4 0 0 -4
128.72 204.49 -2 -2 0 4 4 0 0 -4]
<0001020404040a01020404040a01020404040a01020404040a01020404040a010
20404040a01020404040a01020404040a01020404040a01020404040a010204040
40a01020404040a01020404040a01020404040a01020404040a01020404040a010
20404040a01020404040a01020404040a01020404040a01020404040a010204040
40a01020404040a01020404040a01020404040a>] ufill
```

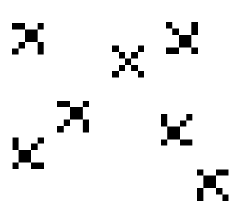
User Path in a Large Array

■	Filled Rectangle	849
□	Open Rectangle	618
+	Cross	531
×	X	607

(500 points)

Another notable drawback of this particular method is that there is no way to adjust the positioning of control points in the array. Unlike the **ucache** method the appearance of the control points can be inconsistent because the individual control points do not start in prescribed locations within device space. The user path rounding is only performed once in this case and not for each point as in the case above. A wrap could be used to round the control point location to device space before placement in the array but since this would add additionally client-server messages (with return arguments) the resulting time would be unacceptable.

No Adjustment



A less significant item about user paths in a large array is that the subpaths (control points) are painted or stroked at that same time, that is, they appear on the screen simultaneously. All the other methods cascade the control points in a continuous procession. This is not a problem when drawing in a buffered window or when the number of operators are less than the recommended number per **ufill** or **ustroke**. When drawing on the screen in a non-buffered window and when multiple rendering operators are necessary, the control points appear in waves. In the **ControlPoint** application, 1000 points will cause three sets of points to be displayed with each set containing a few hundred points. The appearance of the three waves can be disrupting to the eye.

Note*: Arrays containing over 2500 points may exceed the path limit in the first release of the Display PostScript system, resulting in a limitcheck error. As a result, an upper limit of 2500 points should be placed on user paths arrays used for display purposes. Applications that make use of the emulation provided by* ***DPSDoUserPath( )*** *to produce Level 1 code should place the limit at 1500 points, the path limit for Level 1 printers.*

3.3 Rectangle Operations

The Display PostScript language rectangle operators, **rectfill** and **rectstroke**, provide the fastest and simplest method of those studied. The drawback is that this method can only be used for control points that have rectangular shapes. In the **ControlPoint** application, this method is made unavailable whenever the control point is a cross or an x. When rectangles are to be displayed, though, these operations should be used.

The arguments for the rectangle operations can take three forms, four numbers describing a single rectangle or an array or encoded number string describing an arbitrary number of rectangles. Examples are shown below.

```
x y width height  rectfill
numarray   rectfill
numstring  rectfill
```

The first form above performs the equivalent of:

```
gsave
newpath
x y moveto
width 0 rlineto
0 height rlineto
width neg 0 rlineto
closepath
fill
grestore
```

Providing an array of rectangles for the rectangle operations is much faster than calling operators multiple times for much the same reasons that user paths are faster than conventional drawing. Only one single operator call or wrap is necessary for displaying a large number of rectangles minimizing the number of PostScript language operations that are executed. Whenever possible, multiple rectangles, even different sized rectangles, should be combined into a single operation. When the array of rectangles is replaced by individual calls to **rectfill**, the processing time triples from approximately 300 milliseconds to 1000 milliseconds for 500 rectangles. **RECTOFFSET** and **RECTSIZE** have values of 2 and 4 respectively.

PostScript language code:

```
defineps PSWRectDraw (float XYScratch[j]; int j; char *rectOp)
  XYScratch rectOp
endps
```

C-language code:

```
/*
 * Here we have to calculate the offset from the center because
 * rectfill starts drawing at the location passed in.
 */
- drawRectOp:(int)cell
{
  int  i, j;

  char  *rectOp;

  rectOp = [controlPoint  getRectOp];

  . . .
  for (i = indexOfPoints, j = 0;
    i < indexOfPoints + (numberOfPoints*2); i = i+2, j = j+4)
  {
    /* Draw the rectangles if the array limit has been reached */
    if (j+3 >  MAX_RECTPTS)
    {
      PSWRectDraw (XYBuffer, j, rectOp);
      j = 0;
    }

    XYBuffer[j] = XYPoints[i] - RECTOFFSET;
    XYBuffer[j+1] = XYPoints[i+1] - RECTOFFSET;
    XYBuffer[j+2] = RECTSIZE;
    XYBuffer[j+3] = RECTSIZE;
  }
  PSWRectDraw (XYBuffer, j, rectOp);
  . . .

  return self;
}
```

PostScript language trace:

```
[111.72 380.64 4 4 333.57 92.79 4 4 174.63 82.93 4 4 74.77 54.66 4
4 69.36 133.61 4 4 22.02 80.15 4 4 61.11 354.97 4 4 213.25 204.47 4
4 159.55 28.73 4 4 359.98 164.83 4 4 52.32 320.76 4 4 410.14 393.39
4 4 105.63 123.97 4 4 331.61 320.91 4 4 220.55 152.92 4 4 280.94
464.14 4 4 398.16 318.25 4 4 174.22 401.910 4 4 154.44 488.60 4 4
59.06 301.79 4 4 376.83 419.52 4 4 20.02 377.82 4 4 316.11 454.69 4
4 327.33 123.83 4 4 126.72 202.49 4 4] rectfill
```

***Note**: The NextStep C functions, **NXRectFill( )** and **NXRectFillList( )**, are not used in this example because corresponding RectStroke functions are not available.*

Also note: *A limit of 500 rectangles per* ***rectfill*** *or* ***rectstroke*** *invocation is recommended. The* ***rectstroke*** *operator produces limitcheck errors in the first release of the Display PostScript system when passed an array of 600 or more rectangles. In addition, the Client Library generates overflow errors with arrays larger than 8000 entries. As a result, the* ***rectfill*** *operator can handle at most 2000 rectangles. For these reasons, 500 is a sufficiently large but suitable upper bound for both operators.*

Rectangle Operations

■	Filled Rectangle	289
□	Open Rectangle	587
+	Cross	—
×	X	—

(500 points)

3.4 Compositing

The next method shown composites a bitmap of a control point at each control point location. Although more than twice as slow as **xyshow**, compositing provides an advantage that the **xyshow** does not: multi-colored control points can be used without having to render the control point multiple times. The other two methods require a separate invocation for each color.

The **ControlPoint** application uses a **Bitmap** object to hold the image of the control point. When the control point is first drawn in the bitmap, the bitmap is first painted over with a white color and a transparent alpha. The control point is then imaged into the bitmap. The transparent alpha of the bitmap allows the portions of the drawing background covered by the bit map but not covered by control point to show through. The basic drawing method is used to draw the control point in the bitmap. The variable **basicProc** points to the name of the drawing procedure in the server while **basicOp** points to either “fill” or “stroke”.

C-language code:

```
- drawBitMap:bitMapId
{
  [bitMapId lockFocus];
    PSgsave();
      PSsetalpha(0.0);     /* Transparent */
      PSsetgray(NX_WHITE);
      PSrectfill(0, 0, FIGURESIZE, FIGURESIZE);
      PSsetalpha(1.0);     /* Opaque */
      PSsetgray(NX_BLACK);
      PSWBasic(FIGURESIZE/2, FIGURESIZE/2, basicProc, basicOp);
    PSgrestore();
  [bitMapId unlockFocus];

  return self;
}
```

Once the control point has been imaged, the only step required to display a point is to composite the bitmap at the point's location. An offset should be used so that the center and not the lower left or upper left corner of the bitmap lies on the point location. The source and the trace for this method are listed below:

C-language code:

```
- drawComposite:(int)cell
{
  int       i;
  NXPoint   point;

  [controlPoint drawBitMap:bitMap];

  . . .
  for (i = indexOfPoints; i < indexOfPoints + (numberOfPoints*2);
        i = i+2)
  {
    point.x = XYPoints[i] - FIGUREHALFSIZE;
    point.y = XYPoints[i+1] - FIGUREHALFSIZE;
    [bitMap composite:NX_SOVER toPoint:&point];
  }
  . . .

  return self;
}
```

PostScript language trace:

```
0 0 8 8 17 execuserobject 109.72 378.64 2 composite
0 0 8 8 17 execuserobject 331.57 90.79 2 composite
0 0 8 8 17 execuserobject 172.63 80.93 2 composite
0 0 8 8 17 execuserobject 72.77 52.66 2 composite
0 0 8 8 17 execuserobject 67.36 131.610 2 composite
0 0 8 8 17 execuserobject 20.02 78.15 2 composite
0 0 8 8 17 execuserobject 59.11 352.97 2 composite
0 0 8 8 17 execuserobject 211.25 202.47 2 composite
0 0 8 8 17 execuserobject 157.55 26.73 2 composite
0 0 8 8 17 execuserobject 357.98 162.83 2 composite
0 0 8 8 17 execuserobject 50.32 318.76 2 composite
0 0 8 8 17 execuserobject 408.14 391.39 2 composite
0 0 8 8 17 execuserobject 103.63 121.97 2 composite
0 0 8 8 17 execuserobject 329.61 318.91 2 composite
0 0 8 8 17 execuserobject 218.55 150.92 2 composite
0 0 8 8 17 execuserobject 278.94 462.14 2 composite
0 0 8 8 17 execuserobject 396.16 316.25 2 composite
0 0 8 8 17 execuserobject 172.22 399.91 2 composite
0 0 8 8 17 execuserobject 152.44 486.60 2 composite
0 0 8 8 17 execuserobject 57.06 299.79 2 composite
0 0 8 8 17 execuserobject 374.83 417.52 2 composite
0 0 8 8 17 execuserobject 18.02 375.82 2 composite
0 0 8 8 17 execuserobject 314.11 452.69 2 composite
0 0 8 8 17 execuserobject 325.33 121.83 2 composite
0 0 8 8 17 execuserobject 124.72 200.49 2 composite
```

The **composite** operator takes the graphic state object of the image as one of its arguments. In the case above, the graphic state object is stored as a user object. The **17 execuserobject** serves to retrieve the graphic state object and place it on the stack. User objects and graphic state objects are covered more thoroughly in Technical Paper #5057, *User and Graphic State Objects and the Clock Application*.

Since each control point requires a separate single operator call and one PostScript language operator, **composite**, the cumulative overhead of the calls accounts for the reduced performance in comparison to the rect operations and **xyshow**.

The limitation of this method is that it is highly device dependent. A bitmap is only suitable for one resolution. Whereas the other types of control point renderings are infinitely scalable, a bitmap does not scale particularly well, especially when the amount of data is small. As a result, if the size of a control point changes then a new image is necessary. The **ControlPoint** application draws the control point in the bitmap object using the basic drawing technique. This is acceptable since just like the font, the control point is rendered only once. In the case of the font, the control point image is stored in the font cache. In the case of the bitmap, the image is stored in the bitmap.

A **TIFF** file could also be stored in the **.nib** file for use as a control point image. If a **TIFF** file is used then a separate image is necessary for each device resolution. Drawing into the bitmap can reduce the amount of device dependence. While the bitmap is still tied to one resolution, the drawing is not. A multi-colored control point bitmap could be created very quickly with a font or other drawing technique.

Compositing

■	Filled Rectangle	1178
□	Open Rectangle	1166
+	Cross	1167
×	X	1172

(500 points)

3.5 show and xyshow

The next two methods makes use of the font machinery to cache and display points. In each case, the character description is executed once, when the character is first displayed. Each subsequent display uses the description stored in the font cache. The font cache machinery can render characters quickly. Printing a character that is already in the font cache is up to a thousand times faster than scan converting it from the character description in the font.

The font cache does not retain color information. It stores the data as a mask. As a result, the color or gray should not be set in the character descriptions. Changes in color should be produced by changing the current color or gray in the current graphic state before invoking the **show** (or **xyshow**) operator. If multi-colored control points are desired, show operator should be repeated with a change in color or gray appearing between invocations, or a bitmap object should be used with the composite operation.

The font description used for the **show** and **xyshow** operators does not appear in the code segments below but is covered in a separate section that follows. The **getChar** method returns the character that corresponds to the current control point. The **selectFont** method selects and scales the font. (**FONTSIZE** has a value of 5.) The **PSgsave( )/PSgrestore( )** nesting preserves the previous font.

show

The show operator is not recommended because each character requires a separate wrap to execute a **moveto** and a **show** operation. Control points do not appear together and so only a single character can be rendered per **show** operator. The font caching mechanism provides some benefit, but the advantage is offset by the overhead that each wrap incurs as well as the processing time required for each operator.

PostScript language code:

```
defineps PSWShow (float X, Y; char *Char)
   X Y moveto (Char) show
endps
```

C-language code:

```
- drawShow:(int)cell
{
  int   i;
  char fontchar[2];

  fontchar[0] = [controlPoint getChar];
  fontchar[1] = 0;
  [controlPoint selectFont:FONTSIZE];

  . . .
  PSgsave();
    for (i = indexOfPoints; i < indexOfPoints + (numberOfPoints*2);
          i = i+2)
      PSWShow(XYPoints[i], XYPoints[i+1], fontchar);
  PSgrestore();
  . . .

  return self;
}
```

PostScript language trace:

```
113.72  382.64 moveto (a) show
335.57  94.79 moveto (a) show
176.63  84.93 moveto (a) show
76.77  56.66 moveto (a) show
71.36  135.61 moveto (a) show
24.02  82.15 moveto (a) show
63.11  356.97 moveto (a) show
215.25  206.47 moveto (a) show
161.55  30.73 moveto (a) show
361.98  166.83 moveto (a) show
54.32  322.76 moveto (a) show
412.14  395.39 moveto (a) show
107.63  125.97 moveto (a) show
333.61  322.91 moveto (a) show
222.55  154.92 moveto (a) show
282.94  466.14 moveto (a) show
400.16  320.25 moveto (a) show
176.22  403.91 moveto (a) show
156.44  490.60 moveto (a) show
61.06  303.79 moveto (a) show
378.83  421.52 moveto (a) show
22.02  379.82 moveto (a) show
318.11  456.69 moveto (a) show
329.33  125.83 moveto (a) show
128.72  204.49 moveto (a) show
```

Show

		No Adjustment	Adjusted
■	Filled Rectangle	1123	1151
□	Open Rectangle	1124	1120
+	Cross	1131	1166
×	X	1140	1128

(500 points)

xyshow

The **xyshow** operator, however, works very well for displaying control points. This operator takes a string of characters followed by an array or encoded number string of point displacements. The number of operations are minimal; in the case of the **ControlPoint** application, a single **xyshow** operation is used. The size of the data structures necessary for **xyshow** are smaller than those for either the rect operator method or the user path method. Only one character and two points are necessary to describe the control point for **xyshow** whereas the rect operators need four points per control point and the user path description at least two points for each path construction *operator* plus an entry for the operator itself.

The array passed to the **xyshow** operator contains the displacements relative to the previously placed character and not the absolute location of the character. The first two numbers in the array are the positions of the second character offset from the positions of the first. A current point must be established before executing the **xyshow** operator. This relative positioning means that when a control point is moved, not only does the relative position for that point change, but also the relative position for the point immediately following as well.

PostScript language code:

```
defineps PSWXYShow(float X, Y; char *CharString;
       float XYCoords[j]; int j)
  X Y moveto  (CharString) XYCoords xyshow
endps
```

C-language code:

```
- drawXYShow:(int)cell
{
  int    i, j;
  char  fontchar;
  fontchar = [controlPoint getChar];

  /*
   * Place the characters into the character string for xyshow.
   * Terminate the string with a NULL character.
   */
  for (i = 0; i < numberOfPoints; i++)
    OpsBuffer[i] = fontchar;
  OpsBuffer[i] = 0;

  [controlPoint selectFont:FONTSIZE];

  . . .
  PSgsave();
    /* Calculate the displacement from the previous character. */
    for (i = indexOfPoints+2, j = 0;
          i < indexOfPoints + (numberOfPoints*2); i++, j++)
      XYBuffer[j] = XYPoints[i] - XYPoints[i-2];

    /*
     * Provide a dummy set of displacements for the move after
     * the last character has been shown.
     */
    XYBuffer[j++] = 0;
    XYBuffer[j++] = 0;

    /* Establish a current point and they execute the xyshow. */
    PSWXYShow(XYPoints[indexOfPoints], XYPoints[indexOfPoints+1],
      OpsBuffer, XYBuffer, j);
  PSgrestore();
. . .

  return self;
}
```

PostScript language trace:

```
113.720001 382.649994 moveto
(aaaaaaaaaaaaaaaaaaaaaaaaa)
[221.85 -287.85 -158.93 -9.86 -99.86 -28.26 -5.41 78.94 -47.34 -53.45
39.09 274.82 152.13 -150.5 -53.69 -175.74 200.42 136.10 -307.66
155.93 357.82 72.62 -304.5 -269.42 225.97 196.94 -111.05 -167.98
60.38 311.22 117.22 -145.89 -223.94 83.66 -19.77 86.69 -95.38 -186.81
317.77 117.73 -356.82 -41.70 296.10 76.86 11.22 -330.85 -200.61 78.66
0 0] xyshow
```

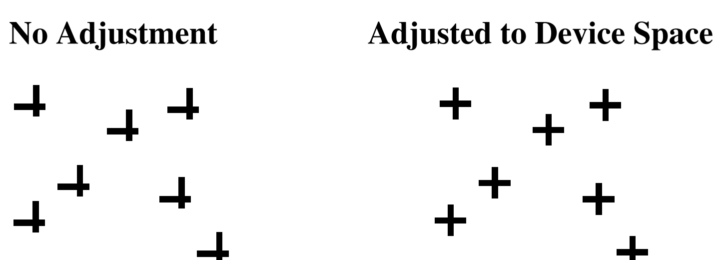
Unlike **rectfill** or **rectstroke**, **xyshow** can display different characters with a single operation. Two different types of control points can be drawn just by including different characters in the string that is passed to **xyshow**.

xyshow

		No Adjustment	Adjusted
■	Filled Rectangle	481	482
□	Open Rectangle	482	483
+	Cross	483	480
×	X	483	488

(500 points)

In both cases, the consistency between images is handled by the font machinery. Each character is positioned at the same point within device space. This ensures that the same image is produced regardless of the user space point chosen. The font in the **ControlPoint** application employs the same procedure used in the basic drawing method to round the path placement to device space. This procedure provides device independence without sacrificing performance (the font cache reduces the number of times the description is executed). This approach is fine for the simple character descriptions that are employed. Applications that use more elaborate characters might want to consider the Type 1 font format. The section below contains details on the character descriptions used.



(Consistency handled by the font machinery.)

4. CREATING A TYPE 3 FONT

The most significant part of this approach is to create a font of control points. The section below shows the font used in the application briefly explaining each section. The PostScript Language Reference Manual (the *Red Book*) and the PostScript Language Tutorial and Cookbook (the *Blue Book*) provides a detailed explanation of font dictionaries as well as several examples of creating or modifying font programs. These books should be referred to for more detailed information in this area. The font used in the example is a Type 3 font. A Type 1 font program can be created, incorporating hints to accommodate small point sizes and low resolutions. A description of this format is available in the *Adobe Type 1 Font Format* specification.

Each font program contains a number of key-value pairs. Some of these are used by the font machinery and must adhere to the correct semantics. Others are optional and user-definable. Each font must have the following keys: **FontMatrix**, **FontType**, **FontBBox** and **Encoding**. In addition, each font must have a procedure called **BuildChar** whose job it is to render the character for the encoding vector passed to it. Below lies the font definition for the four control points.

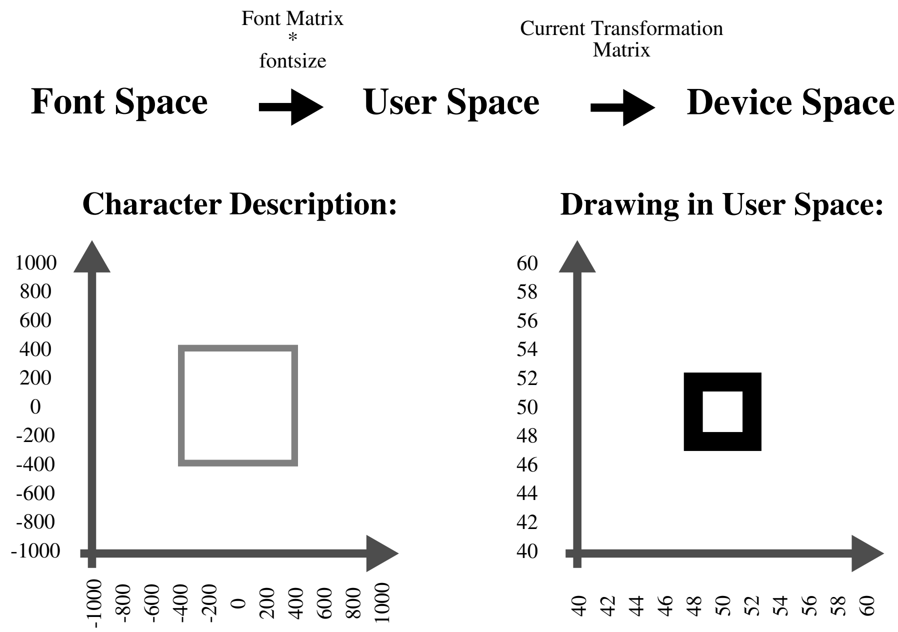
```
defineps PSWDefineFont(char *fontname)
  8 dict dup begin
  /FontName /fontname def
  /FontType 3 def
  /FontMatrix [.001 0 0 .001 0 0] def
  /FontBBox [-500 -500 500 500] def
  /Encoding 256 array def
    0 1 255 {Encoding exch /.notdef put} for
  Encoding
    dup (a) 0 get /Rectfill put  dup (b) 0 get /Rectstroke put
    dup (c) 0 get /Ximage put    (d) 0 get /Crossstroke put

  /CharProcs 5 dict def
  CharProcs begin
    /.notdef { } def
    /Rectfill {
      -300 -300 sa moveto 0 600 rsa rlineto
      600 0 rsa rlineto 0 -600 rsa rlineto
      closepath fill
    } def
    /Rectstroke {
      -400 -400 sa moveto 0 800 rsa rlineto
      800 0 rsa rlineto 0 -800 rsa rlineto
      closepath stroke
    } def
    /Ximage {
      -500 -500 translate
      1000 1000 scale
      5 5 true [5 0 0 5 0 0]
      {<88 50 20 50 88>} imagemask
    } def
    /Crossstroke {
      0 400 sa moveto 0 -800 rsa rlineto
      -400 0 sa moveto800 0 rsa rlineto stroke
    } def
  end
```

```
  /BuildChar { % font dict, char code
    500 0 -500 -500 500 500 setcachedevice
    exch begin
      true setstrokeadjust
      Encoding exch get
      CharProcs exch get
      exec
    end
  } def
  end
  /fontname exch definefont pop
endps
```

4.1 Required Keys

FontMatrix is the matrix that describes the mapping of the character to the user coordinate system. Just as the user space uses a matrix to map to device space, a font program uses a matrix to map to user space. This process avoids having to scale the character descriptions in order to scale the font. If a 10 point font is desired, the matrix is scaled by 10. If 12 point font is desired, the matrix is scaled by 12. (The font matrix is either scaled by the scalar argument passed to **scalefont** or is concatenated by the matrix argument passed to **makefont**.

Most font programs, including the one in the **ControlPoint** application, use a **FontMatrix** of [0.001 0 0 0.001 0 0] and define the characters in terms of a 1000 unit character coordinate system. In other words, a character is drawn with the idea that 1000 points in the character coordinate system will equal 1 point in a 1 point font and 2 points in a two point fonts. In the font for the control points, the characters are centered at the point (0, 0) and extends to the left and bottom as well as to the right and top. This centers the character on the current point when the character is drawn. The use of a 1000 unit coordinate system is part historical and part regimented. The Type 1 font format employs a number encoding scheme to reduce the amount of space necessary to store integer values between -1131 and 1131. Working with a 1000 unit coordinate system allows for a suitable drawing range while still taking advantage of the compact number representation. Type 3 font programs do not employ this encoding scheme but most choose to use the same **FontMatrix** for uniformity purposes. The diagram below shows the mapping that occurs to the ControlPoint font which has a font matrix of [.001 0 0 .001 0 0] and a fontsize of 5.




```
-400 -400 moveto
0 800 rlineto
800 0 rlineto
0 -800 rlineto
closepath
```

```
/ControlPointsFont
5 selectfont
50 50 moveto
(b) show
```

The **FontType** indicates where the information for the character description is to be found and how it is represented. The font type for a user-defined font should be set to 3. The **FontBBox** gives the lower left and upper right coordinates for the bounding box that contains all the characters were they to be imaged at the same point. The font machinery uses this array for setting clipping paths and making caching decisions. The values in the **FontBBox** should be accurate if they are non-zero.

The **Encoding** entry is an array of 256 names that map the character codes to the procedure names. An application or a computer system may change a font’s encoding vector to match the requirements of the application or system. The index into the array is the character code. The entries of the array are procedure names for the characters. Therefore, juggling the entries changes the encodings. In the **ControlPoint** font, the character codes for **a**, **b**, **c** and **d** are used for the control points. The procedure names for the control points are placed in these locations of the encoding vector array. The **/.notdef** procedure name is placed in all the other locations. Each character name has a procedure by the same name in the **CharProcs** dictionary. The **/.notdef** procedure is used as a place holder for unused characters.

Encoding Array
(ControlPoint Font)

95	/.notdef
96	/.notdef
97	/Rectfill
98	/Rectstroke
99	/Ximage
100	/Crossstroke
101	/.notdef

CharProcs Dictionary
(ControlPoint Font)

/.notdef	{ }	
/Rectfill	{. . .}	■
/Rectstroke	{. . .}	□
/Ximage	{. . .}	×
/Crossstroke	{. . .}	+

```
/BuildChar { % font dict, character code
  500 0 -500 -500 500 500 setcachedevice
  exch begin
    true setstrokeadjust
    Encoding exch get
    CharProcs exch get
    exec
  end
} def
```

The elements of the string that are passed to the **show** and **xyshow** operators are treated as character codes. The character codes are used as indices into the **Encoding** array to obtain a procedure name. The procedure name is then looked up in the **CharProcs** dictionary and its value executed. The **setcachedevice** in the **BuildChar** procedure above sets the dimensions of the font cache and the **setstrokeadjust** turns on stroke adjustment. Stroke adjustment is *off* by default when building a Type 3 font character.

4.2 Optional Keys

Optional keys typically used in font software are: **FontName**, **PaintType**, **Metrics**, **StrokeWidth**, **FontInfo**, **UniqueID**, **CharStrings** and **Private**. The **ControlPoint** font only includes the **FontName** entry. This name is separate from the name used to define the font and is provided for information only. The interpreter uses the name passed to it in the **definefont** operation to identify the font.

UniqueID provides a unique identifier for the system to use to identify characters that have already been created and cached. The font machinery uses this ID to operate more efficiently across applications. Each font program that uses a **UniqueID** should have a different value. Font programs that have a **UniqueID** will be cached across jobs while fonts that do not have a **UniqueID** will only be cached for the immediate jobs. **UniqueIDs** that are not unique may inadvertently cause incorrect characters to appear. Adobe Systems Incorporated maintains a registry of **UniqueId** numbers. The font used in the **ControlPoint** application is localized to one application and relatively trivial so a **UniqueID** is not employed.

5. Printing Issues

The focus of this note and the **ControlPoint** application has been the display of control points on the screen. The techniques used here can just as easily be used to display an arbitrary number of objects as part of an application. An example would be to display stars within a star map. Such an application would more than likely want to be able to image to a printer. Only two methods would be able to print without any adjustments, drawing with basic operators and using the show operator. The PostScript language instructions produced by these methods are compatible with all PostScript language interpreters.

User paths, rect operations and the **xyshow** operator need emulation procedures for interpreters that do not contain the Display PostScript extensions. In the case of user paths, the NeXT procedure, **DPSDoUserPaths( )**, automatically provides emulation for printing.

The emulation for the rect operations are included in the NeXT print driver but are shown below for reference purposes.

PostScript language code:

```
/__NXDoRectOp {
  1 index type /arraytype eq {
  exch aload length
  dup 2 add -1 roll exch
  4 idiv}
  { 1 } ifelse
  {5 1 roll
  newpath 4 2 roll moveto 1 index 0 rlineto
  0 exch rlineto neg 0 rlineto closepath
  dup cvx exec
  } repeat pop
} __NXbdef
/rectclip {/clip __NXDoRectOp newpath} __NXbdef
/rectfill {gsave /fill __NXDoRectOp grestore} __NXbdef
/rectstroke {gsave /stroke __NXDoRectOp grestore} __NXbdef
```

The emulation for the **xyshow** operator shown below is not contained in the NeXT print driver and so it must be made a part of the prologue by the application. The Adobe PostScript language compatibility guidelines prescribe that only complete emulations may be assigned the actual operator name. Partial emulations should be given a separate and distinct name. Since this emulation does not handle encoded number strings and as a result is not complete, the name **xys** is used. This name should be used in place of **xyshow**. The procedure assigned to **xys** is selected conditionally. The **xyshow** operator is assigned if the **xyshow** operator exists. The emulation is assigned if **xyshow** does not exist. The code segment below shows this sequence.

PostScript language code:

```
/_xyshowemulation {
  0 1 3 index length 1 sub {
    currentpoint
    2 index 5 index exch 1 getinterval show
    moveto
    2 mul dup 2 index exch get exch 1 add 2 index exch get rmoveto
  } for
  pop pop
} bind def

/xyshow where {
  pop
  /xys /xyshow load def
}{
  /xys /_xyshowemulation load def
} ifelse
```

The compositing technique cannot be used for printing. Although it is possible to use the **image** and **imagemask** operators to transfer a bitmap to the page, these approaches are not recommended. With the **image** operator, the entire bit map will appear as if drawn with opaque ink. Everything that is under the dimensions of the bitmap would be obscured. The **imagemask** will allow transparency but the resulting image will be limited by the amount of sample data. In most cases, the image will be of poorer quality than if constructed as a path.

6. SUMMARY

Small and frequently drawn objects present display problems that larger objects do not share. At sizes less than 10 points, variations in how a path falls within a pixel can have a dramatic effect on the appearance of the object. As a result, a technique for displaying a large number of these objects must not only be fast but must reproduce the objects to be exactly alike. Basic drawing techniques can achieve this consistency but at a high price in terms of performance. User paths are poor candidates because their restricted data format does not allow any positional adjustment to an object. Compositing provides a clean and consistent approach but at the cost of device dependence. Different resolution or different size objects need separate bitmaps.

The two methods that are recommended are the **rectfill** and **rectstroke** operators and the **xyshow** operator. **Rectfill** and **rectstroke** operators are limited to rectangles and **xyshow** requires that the objects be converted into a font. Nevertheless, these operators provide the most efficient and consistent display of small images. **Rectfill** and **rectstroke** require no overhead such as wraps, user path or character descriptions and are very fast, especially when passed arrays of rectangles. **Xyshow** uses the font cache to quickly blit the stored image onto the page or display. Both methods provide a mechanism that automatically adjusts the position of the object thereby producing the same image regardless of the actual mapping to device space.

Summary of Times

(without adjustments)

		Basic Drawing	User Path Cache	Large User Path	Rect Ops	Compositing	Show	xyshow
■	Filled Rectangle	2896	1953	849	**289**	1178	1123	**481**
□	Open Rectangle	2217	1903	618	**587**	1166	1124	**482**
+	Cross	1797	1911	531	—	1167	1131	**483**
×	X	1852	1918	607	—	1172	1140	**483**

(500 points)