Improving Drawing Performance

In most applications, a large chunk of time is consumed by drawing. Indeed, the optimization of those sections of your application that perform drawing should probably one of your first priorities, since often relatively small changes to your code can result in dramatic improvements in the real and perceived performance of your application. This discussion is broken down into three topics: General rules to follow in a client-server world, efficient focusing, and specific techniques for optimizing the performance of your drawing code.

General Rules in a Client-Server World

NEXTSTEP® is based on a client-server model, which has certain implications when you're writing your application, as discussed below.

Write Your Program in Objective C, not in PostScript

Underlying the design of NEXTSTEP is the assumption that the center of gravity for a NEXTSTEP application should be on the application side rather than the Server side, that is, applications in the NEXTSTEP environment should be written in Objective C®, and not in PostScript. This was based on performance considerations, as well as on the view that given the characteristics of the respective languages and the state of the respective development and debugging environments, it was much easier to write and debug applications in Objective C than it was to do the same things in PostScript code.

The bulk of your algorithms and data structures should be written in Objective C and should avoid maintaining

state in the Window Server. This point about state is important for a number of reasons:

- If your application maintains state or other data structures in the Server, you must either provide PostScript procedures to manipulate that state data, or you need to make calls to the Server whenever your algorithms on the application side need to access or modify that data. On the other hand, if you maintain two copies of the state data, one on the application side and one on the Server side, you'll need to ensure that they're always in synch. In either case, you'll require communication with the Window Server, and that implies interprocess communication. You're far better off to avoid the problem entirely by not maintaining state information in the Server, and doing all the work on the application side. If you have PostScript wraps that require state information, pass the state information as parameters to the wraps.
- Another side benefit of avoiding a lot of functions and state information in the Server comes when you print. As you know, printing occurs in a different PostScript context than that of your running application. As a result, if you have any PostScript procedures or state variables that are in the Server, you must ensure that they're downloaded to the printing context as well. While this certainly can be done, it's often easier to avoid the problem altogether.

It should be noted that the above discussion shouldn't be construed as an argument against using wraps. In fact, replacing calls to multiple single-operator C functions with a single call to a wrap may result in a performance gain. Adobe recommends that while single-operator C functions are easy to use, wraps should be used for all but the simplest drawing. However, avoid using wraps for any operation that can be done entirely within the client application. It almost always pays to perform calculations on the application side and pass the results as parameters, as opposed to performing the calculations in the Server. Similarly, pass required state information as parameters to the wrap. Note also that there's little or no performance gain from downloading a wrap that contains a procedure definition and invoking the downloaded procedure from another wrap when needed, as opposed to downloading the procedure each time it's to be executed. Last, and perhaps most important, is being smart about the drawing your application does and your choice of techniques to perform that drawing. This will probably have a much more dramatic effect on overall performance than the decision whether to use single-operator C functions rather than custom wraps.

Avoid Round Trips

Any PostScript function or wrap that has a return value in its parameter list will cause a round trip to the Window Server, and should be avoided if possible. The data an application sends to the Server is buffered to minimize communication overhead and context switches. Executing wraps that return values defeats this buffering, since the data must be flushed to the Server before an answer can be returned. This is always going to be slower than inspecting a variable in your application, or calling a method or function within your application. Thus, be judicious when making calls to the Server; consider whether by caching data or state on the application side you can avoid making the call altogether.

Use the Application Kit's Font Mechanism

The Application Kit provides a number of classes (Font, FontManager, and FontPanel) that can perform many font operations without interacting with the Server, thereby reducing the number of round trips. Wherever possible, you should use the font mechanism built into the Application Kit rather than interact directly with the Server. For example, the font mechanism caches font metric information on the application side. Hence, you can query the Font object for font metrics without involving communication with the Window Server. In fact, it's almost twice as fast to use the **getStringWidth:** method provided by the Font class than to use **PSstringwidth()** to get the width of strings before drawing.

Avoid Unnecessary Calls

Below is an example of making a call to the Server that's silly and inefficient. In this case you're better off just setting the gray, rather than querying the Server first for the current gray.

```
float currentGray;
PScurrentgray(&currentGray);
if(currentGray != desiredGray)
PSsetgray(desiredGray);
```

In general, you should avoid making unnecessary calls to the Window Server. Try to structure your drawing code so that you avoid redundant or unnecessary calls to **PSsetgray()**, **PSsetlinewidth()**, **PSgsave()**, and so on.

Boxcarring

The Display PostScript language has added a number of imaging operations that don't alter the underlying imaging model, but which cover frequently performed operations and provide highly optimized execution. These operations include:

user paths — A number of operators are provided that allow you to download a very compact representation of a path. The operator builds a special kind of path from it and performs some operation like stroke or fill on the path. (The use of user paths is discussed at length below.)

rectangles — A number of operators are provided to build a path made up of one or more rectangles, and perform some operation on the resulting path (fill, stroke, or clip). The operators can take either a single rectangle or an encoded array of rectangles. If you're doing any rectangle operations, you're almost always better off using these operators than **moveto**, **lineto**, and so on. Note the that Application Kit provides a number of functions (**NXRectClip()**, **NXRectClipList()**, **NXRectFill()**, **NXRectFillList()**, **NXEraseRect()**, and so on) that use the underlying rectangle operators provided by the Display PostScript language. You should use them whenever performing rectangle operations.

text drawing — A number of operators are provided that take an encoded array of coordinates and a character string, and use the coordinates to place and show the characters in the character string. (The use of **xyshow** is discussed below.)

We call the use of these operators ªboxcarringº because they allow you to download one large chunk of data to the Server where it's operated on efficiently in bulk, instead of sending many smaller chunks of data that are processed individually. In fact, these operators can be several times faster than the equivalent single-operator C functions or wraps.

Needless to say, you should become familiar with these operators and use them wherever appropriate. Note also that the rectangle, **xyshow** and user path operators above are simulated in the default print packages, so you can use them when printing without modification. Because of their importance, user paths are discussed at length in a later section.

Efficient Focusing and gstate Objects

Before an application draws in a view, the view must become the focus for drawing; otherwise the drawing will appear in some other view that was previously the focus. When you draw within the context of **drawSelf::**, focusing is performed by the view's **display** method. If you draw outside the context of **drawSelf::**, you manage focusing explicitly by invoking **lockFocus** and **unlockFocus**.

Ensuring that your view has the focus before drawing is a necessary step in drawing, but you should avoid unnecessary focusing. When the view is focused on, an appropriate transform matrix must be built to map the view's coordinate system back to the window's coordinate system, an appropriate clipping path must be built, and a **PSgsave()** is performed. While care has been taken to make this process as efficient as possible, it nonetheless takes time and is worth paying attention to. Here are some suggestions:

- The fastest focus is the one you don't need to do. Try to structure your code so that you aren't doing unnecessary focusing. For example, if you're drawing within the context of a modal loop, put the **lockFocus** and **unlockFocus** outside the loop.
- Turn off clipping if not needed with **setClipping:**. Building the clipping path is usually the slowest part of focusing, so you can usually save a good deal of time by turning clipping off for the view if you don't need it. You don't need clipping if you're sure you'll never attempt to draw outside the bounds of your view.
- Use **gstate** objects. (More on this below.)

If you have a view that's repeatedly being focused on (for example, the gauge on an instrument panel of a flight simulator), you should consider allocating a **gstate** object for the view by using View's **allocateGState:** method. A **gstate** object is a PostScript object (not an Objective C object) that contains all the information contained in a graphics state, and is a Display PostScript extension designed to make it much more efficient to switch graphics states. The very first time a **lockFocus** or **display** is sent to a view for which a **gstate** object has been allocated, the view machinery focuses on the view in the normal manner, has the Window Server take a snapshot of the graphics state, and then sticks it in a **gstate** object. Thereafter, whenever focusing occurs, the view machinery first checks to ensure the **gstate** object is still valid (that is, the coordinate system of the view or its superviews has not been permanently changed) and, assuming it is, has the Window Server copy the **gstate** to the current graphics state.

Note: View has a handy method called **initGState** that's called automatically whenever the **gstate** is created or modified. You can override **initGState** to initialize other aspects of the graphics state, such as line width or gray level.

The use of **gstate** objects effectively avoids all the work normally associated with focusing, and subsequently reduces the time required for focusing. However, they don't come for free, and thus should be used judiciously. Since a **gstate** object contains all the information contained in a graphics state, including the current clipping path and current path, they consume hundreds of bytes within the Window Server at a minimum, and can be

much larger depending on the nature of the **clippath** and **currentpath**.

If your application *repeatedly* focuses on and draws in a view, it may be a good candidate for a **gstate** object. On the other hand, if you have a view that your application focuses on infrequently, it may not make sense to allocate a **gstate** object for it, regardless of the amount of drawing you actually do in that view. Allocating a **gstate** affects only the focusing time, as it has no effect on the drawing time once the graphics state has been set up.

You'll see less of a gain from using **gstate** objects if clipping has been turned off for the view, because building the clipping path is normally the most time-consuming part of focusing.

In general, you don't need to allocate a **gstate** object for a control (e.g., a slider or scroller), because any control with a modal loop will typically focus once at the beginning of the loop and unfocus at the end. When the action message of the control is sent to the control's target, the target will typically do some drawing in response and bracket its drawing by a **lockFocus** and **unlockFocus**. However, the focus is still on the control when the action message returns (courtesy of the target's **unlockFocus**). Thus, while it may make sense for the target of the control to have a **gstate** allocated for it, it's probably not worth allocating a **gstate** for the control itself.

Lastly, since every window has a **gstate** object allocated for it by default, there is little to be gained from allocating a **gstate** for the **contentView** of a window.

The key point here is that the use of **gstate** objects is a convenient and powerful way to reduce focusing time, but you should use them judiciously. As CPU's speed increases, there are fewer cases where the perceived improvement from using gstates is worth the memory they consume.

Efficient Drawing

In many cases, your application's overall performance will be determined by its ability to perform the minimal

amount of drawing necessary and to use the most efficient means of performing that drawing. This section first describes how to avoid unnecessary drawing, and then discusses a number of the NeXT and Display PostScript extensions that can radically improve drawing performance.

Avoid Unnecessary Drawing

The fastest drawing you'll ever do is the drawing you don't do. You should always look for ways to avoid doing drawing that's unnecessary. It's almost always worth the effort to do the work necessary to cull out drawing that's not visible, although you shouldn't take this to the extreme and implement your own two-dimensional clipping. Limit your drawing to the area within your view that's contained within the update rectangles passed to **drawSelf::**Ðthey'll tell you the area within your view's coordinate space that's visible, based on the intersection of the view's frame and the frames of the views above it in the hierarchy. In the case of scrolling, they'll also tell you what area of your view needs to be redrawn. Even a simple approach of intersecting the bounding boxes of objects with the update rectangles, and drawing only those that intersect, will get you a long way. Of course, if you know that your view is wholly contained within its superviews and that your view is never going to be placed within a ScrollView, and you don't have graphical objects that live outside the frame of the view, then this is less important.

A related point is to avoid drawing in off-screen windows or panels until it's absolutely necessary. For example, if you have a panel that contains things that need to be updated as the user performs actions in another window, don't update the items unless the panel is visible or about to become visible. You'll note that menu updating is a common instance of this type of problem. In fact, the Application Kit provides a mechanism (discussed in a later section) to easily handle the updating of menus, panels, and windows. It should be noted that a consequence of this approach is that the time to bring a panel to the front will probably be longer than if the contents are updated as you go along. Whether this is an acceptable tradeoff depends on the specific situation.

Look for unnecessary drawing by running your application with the command line arguments

-NXAllWindowsRetained YES. This forces all windows in an application to be retained, so that you can watch drawing as it is happening. Look for controls and other items being drawn redundantly.

To examine the quality of the PostScript your application's custom views are producing, run the application under **gdb** and set breakpoints around methods that draw (e.g., the view's **drawSelf::** method). Before drawing, use the debugger macro **showps** to turn on a dump of the PostScript code produced. **shownops** can be used to turn off the dump, to avoid seeing PostScript emitted by other objects.

Compositing

Compositing is probably the single most powerful mechanism for improving the apparent drawing performance, but it should be used wisely. Whenever you have a complicated image that must be redrawn repeatedly, you should consider creating an **NXImage** object, rendering the image into the object, and thereafter use compositing to put the image wherever you need it. If it's a complicated image, this approach can be many, many times faster than redrawing the image each time. This is the standard approach used for most types of animation and for dragging images.

Like most good things, compositing doesn't come for free, and the cost is the memory required for the **NXImage** (especially the off-screen window used to hold the cached image). While the **NXImage** object tries to reduce the overhead, the fact remains that 2 bits/pixel without alpha or 4 bits/pixel with alpha is required. For example, a 3"-by-3" image with alpha represents almost 40K. The space required is multiplied by 4 for 12 bit color iamges, and by 8 for 24 bit images.

In many instances the performance benefit far outweighs the cost, but you should be aware of the cost. Also keep in mind:

- If you don't need transparency, don't use it. Having transparency doubles the storage requirements for black

and white images and has a negative impact on performance for all images as well. An occasional mistake is to use the **NX_COPY** compositing mode instead of **NX_SOVER**. **NX_COPY** will transfer the alpha of the source image to the destination (a bad idea if you are compositing to a large on-screen window), whereas **NX_SOVER** draws the image on top of the destination, heeding any alpha in the source by not transfering it to the destination. Another mistake is to use the **NX_CLEAR** compositing operator to erase the background of the image rather than **NXEraseRect()**. If you ªclearº the background, you're actually making it transparent with the attendant creation of an alpha channel.

- Consider using user paths with **DPSDoUserPath()** as an alternative to compositing. User paths may be a good alternative if you have an image that's large but pretty simple, stroked, and intended to be put on top of another image. This last qualification implies that if you use compositing, you'll need to use transparency and the **sover** operator. In certain cases, user paths will be not only more efficient from a memory standpoint, but faster as well. Unfortunately there's no hard and fast rule here, but keep user paths in mind as an alternative.

- Lastly, compositing isn't a supported operation during printing. If you are using **NXImage** to optimize some PostScript that you drew into the image's cache, you should reexecute the PostScript when printing instead of using the image (this will produce a better looking, device independent result as well). An alternative is to use **NXEPSImageRep** to hold the EPS version of an image, and let NXImage decide when to use the EPS representation. In comparison, user paths are a supported operation during printing, and so you need not be concerned with whether the drawing is going to the screen or to the printer.

DPSDoUserPath()

NeXT provides a function called **DPSDoUserPath()** that facilitates the creation of a user path. Using **DPSDoUserPath()** can be several times faster than using the equivalent single-operator C functions or wraps to draw even simple shapes. As a result, whenever you need to build a path of more than a few points (as few as

four points or so), you may want to consider using **DPSDoUserPath()**.

DPSDoUserPath() relies on a facility provided by the Display PostScript system called, not surprisingly, user paths. A user path is a completely self-contained description (that is, containing only path construction operators and literal number operands) of a path in user space. Coupled with the fact that you're required to specify the bounding box of the user path as part of the user path, the interpreter can build a user path much more efficiently than a normal path. Since path creation by the Server is one of the more time-consuming tasks involved in drawing, this can be a very big gain. In addition, Display PostScript provides a mechanism to allow you to cache user paths to avoid redundant interpretation of the same path definition.

DPSDoUserPath() takes seven arguments:

- A pointer to an array of coordinates
- The number of coordinates in the array
- The numeric type of the coordinates
- A pointer to an array of path-building operators
- The number of operators
- A pointer to an array of coordinates that represents the bounding box for the user path
- An operator to be applied to the user path

DPSDoUserPath() takes these arguments and sends an encoded user path down to the Server, where it's interpreted.

Here's an example of using **DPSDoUserPath()** to build a user path consisting of a series of vertical lines and then have the resulting user path stroked. It also asks the Server to cache the resulting user path so it can be reused.

```
-drawLines2
{
    float   a;
    float  *cArray;
```

```
        char   *oArray;
        float   bbox[4];

        int oi=0,ci=0;
        int size =(int) (bounds.size.width/(interval));

        /* fill cArray and oArray */
        cArray = (float *)malloc(4*numLines*sizeof(float));
        oArray = (char *)malloc(2+2*numLines*sizeof(char));
        oArray[oi++] = dps_ucache;
        oArray[oi++] = dps_setbbox;
        for(a=0.0;a<bounds.size.width;a+=interval){
            cArray[ci++] = a;
            cArray[ci++] = 0.0;
            oArray[oi++] = dps_moveto;
            cArray[ci++] = 0.0;
            cArray[ci++] = 150.0;
            oArray[oi++] = dps_rlineto;
        }
        /* fill bbox */
        bbox[0] = 0.0;
        bbox[1] = 0.0;
        bbox[2] = bounds.size.width;
        bbox[3] = 150.0;

        DPSDoUserPath(cArray,ci,dps_float,oArray,oi,bbox, dps_ustroke);
        free(cArray);
        free(oArray);
    }
```

To give you a sense of the power of user paths, the following table shows a comparison of using single-operator C functions versus **DPSDoUserPath()** to draw a varying number of vertical, zero-width black lines. In each

case, the time given represents the cumulative wall time for doing 100 iterations.

# Lines	Case 1	Case 2	Case 3	Case 4
5	3.57	2.95	2.41	2.19
10	4.46	3.85	2.64	2.37
20	6.99	6.39	3.13	2.70
40	11.34	9.99	4.09	3.36

Case 1 Draw lines using **PSlineto()** and **PSmoveto()**. **lockFocus** and **unlockFocus** inside loop.

Case 2 Same as Case 1 except **lockFocus** and **unlockFocus** taken out of loop.

Case 3 Draw lines using **DPSDoUserPath()**, arrays **malloc**'ed, initialized, and freed inside loop. **lockFocus** and **unlockFocus** outside loop.

Case 4 Same as Case 3 except that user path is cached using **ucache**.

User paths are beneficial even for a small number of points, and the benefit becomes even greater as the number of points goes up. Note also how taking focusing out of the loop has a noticeable impact on performance.

In some cases, **DPSDoUserPath()** may be a good alternative to compositing. For example, if you have what amounts to a stroked path that you want to lay on top of other drawing, using **DPSDoUserPath()** may be faster than rendering the path into an **NXImage** object with a transparent background and using NX_SOVER to composite the image into the destination. This will be particularly true if the path is relatively simple and the number of pixels touched by the path is small in comparison to the number of pixels.

The knowledge and use of compositing and user paths is essential to fast drawing performance on our system.

Be sure to read the relevant descriptions in the *NEXTSTEP General Reference* manual as well as the Adobe documentation on Display PostScript extensions, and start using these techniques if you aren't already using them.

PSxyshow()

If you need to do drawing of the form

```
x y moveto
(a) show
x1 y1 moveto
(b) show
```

(that is, a sequence of **moveto**s and single-character shows), you should consider using **PSxyshow()**, **PSxshow()**, or **PSyshow()**. While not as generally useful as user paths, it can provide the same type of improvement in performance. **PSxyshow()** is the single-operator function for **xyshow**, which is a Display PostScript operator. These operators take a character string and an array of coordinates, go through the character string rendering a character, and then use the next one or two coordinates (depending on the function) to explicitly position the next character.

Those of you drawing rulers should look at **PSxyshow()** or one of its relatives.

Other Drawing Tips

There are a number of other things to keep in mind that will help with drawing performance.

Zero-Width Lines

Use zero-width lines whenever you need top line-drawing performance. Zero-width lines are special-cased by the interpreter and are the fastest way to draw lines. Note that when printing, the line width should be set to some other size since zero-width lines can ªdisappearº on high-resolution printers like the Linotronic®. Adobe recommends using a line width of 0.15 units, which will be treated as zero-width for the purposes of drawing on the screen, but will result in lines that are 0.15 units wide on a printer.

Dashed Lines

Don't use these. Dashed lines using **PSsetdash()** are extremely time consuming. Consider using user paths or compositing as an alternative.

PSgsave() and PSgrestore()

Avoid unnecessary **gsave**s and **grestore**s. In particular:

- Since part of the built-in focusing mechanism using either **display** or **lockFocus** and **unlockFocus** is to do a **gsave** and **grestore**, your drawing code in either **drawSelf::** or within a **lockFocus**/**unlockFocus** is already bracketed by a **gsave** and **grestore**, so there's no need for you to explicitly do it.
- When stroking or filling a path, don't put a **gsave** or **grestore** around the stroke or fill if you aren't planning to use the path again. As Glenn Reid points out in *PostScript: Language Program Design*, the following construction is particularly inefficient:

```
0 0 moveto 100 100 lineto
gsave stroke grestore newpath
```

- Also, if you're clipping to a path, remember that the clip doesn't affect either the path or the current point. Avoid the following construction:

```
gsave clip grestore
```