POSTSCRIPT®

THE DISPLAY POSTSCRIPT® SYSTEM: USER and GRAPHICS STATE OBJECTS and the CLOCK APPLICATION

Technical Note #5055

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 LPS5055

THE DISPLAY POSTSCRIPT® SYSTEM: USER and GRAPHICS STATE OBJECTS and the CLOCK APPLICATION

Technical Note #5055

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

1. INTRODUCTION

The Display PostScript® system contains a number of features specifically designed to aid drawing in a display environment. This technical note, with the help of the **Clock** application, takes a look at two of these — *user objects* and *graphics state objects*.

A *user object* is an integer identifier of a PostScript language object in the server. User objects provide a way to refer to objects in the server using integer keys rather than name keys. Any PostScript language object, such as arrays, strings, dictionaries and gstates, can be referenced by user objects. User objects are more convenient to store and manipulate in C-language data structures than PostScript language variable names which have to be stored as strings or character arrays. This note will focus on managing graphics state objects and user path descriptions with user objects.

A graphics state, or *gstate*, is a collection of parameters that determine how a path is rendered. A graphics state *object* is a new PostScript language data type that encapsulates the values of the parameters that make up a graphics state. Graphics state objects provide more flexibility than the graphics state stack for setting the current graphics state. A graphics state object can be made the current graphics state by performing a **setgstate** operation on the graphics state object whereas the use of the graphics state stack requires nesting graphics states with **gsaves** and then reestablishing the topmost graphics state with each **grestore**. Graphics state *objects* allow indiscriminate switching between states. Graphic state objects can be a performance win for certain uses such as scrolling. They should be used judiciously, however, since there is some appreciable memory cost associated with them.

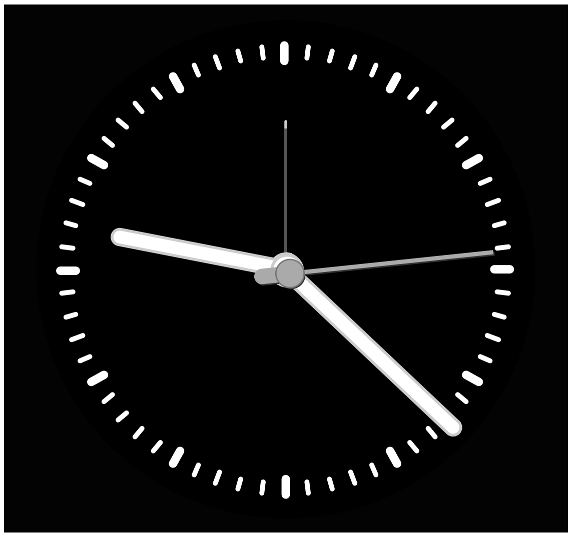
Although these features provide a slight performance advantage over conventional methods, their primary usefulness is in offering a means to manage objects in the server more simply. The NeXT print driver provides user object capability so user objects can be sent to non-Display PostScript interpreters. Graphics state objects are more difficult to emulate for a print environment. The advantage of graphics state objects is for switching between states for frequently drawn objects, which is a requirement for drawing on displays. As a result, graphics state objects should really be used for display purposes and not within page descriptions.

The **Clock** application shows three additional drawing techniques — using an offscreen buffer, storing of the user path descriptions of the clock hands in the server and scaling of the **View** object. An offscreen buffer eliminates having to redraw the clock face every tick. Storing of the user path descriptions in the server eliminates retransmitting the descriptions each time a hand is drawn. Scaling of the **View** object and then redefining the graphics states for the hands scales the clock hands within the view without having to explicitly scale each graphics state or each user path description fragment.

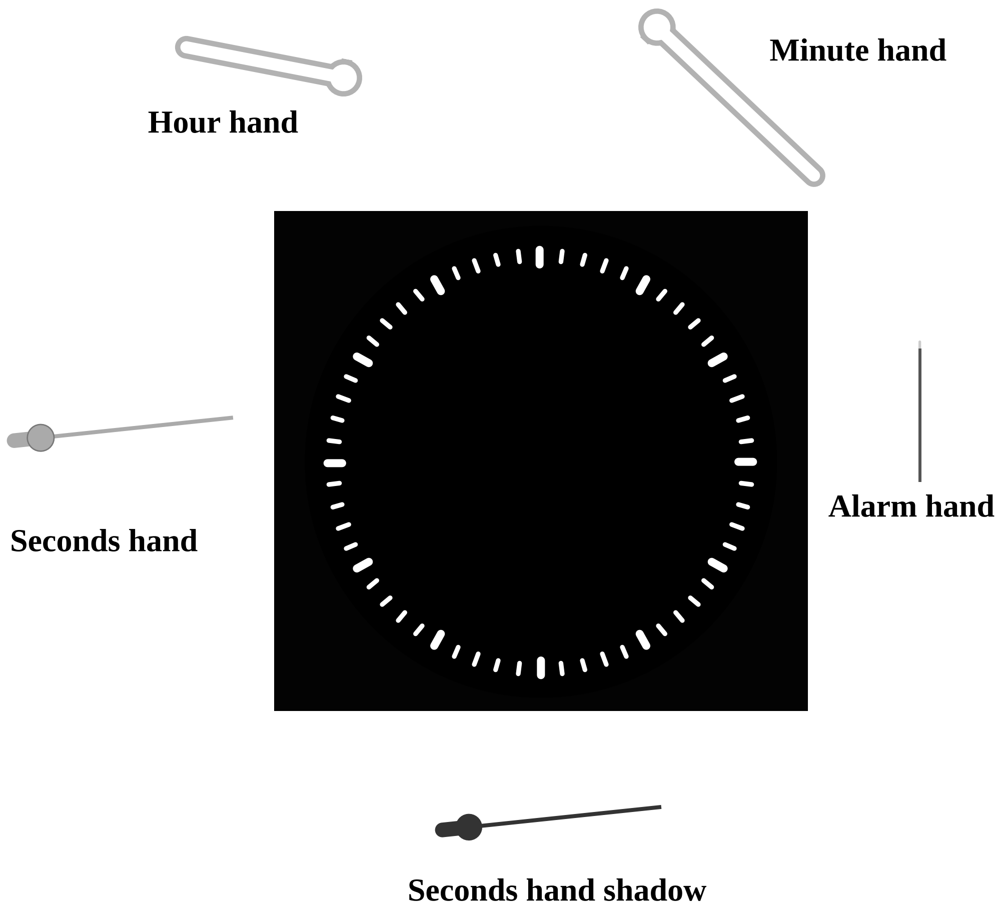
Storing user path descriptions in the server was addressed in Technical Note #5054, *Path Construction & Rendering and the Dial Application*. This technique can improve performance by approximately 20% over retransmitting the user paths with each redrawing. Although the performance difference is not as significant here, the primary reason they are stored in the server is that it simplifies the drawing invocation. Since the description of the clock hands do not change, it makes more sense to store them in the server than to transmit them every time. The **Clock** application differs from the **Dial** Application in that the **Clock** application uses user objects to refer to the user paths in the server rather than static character arrays.

2. CLOCK APPLICATION

A simple clock is used to highlight the several techniques discussed in this technical note.



The clock is composed of a view within a window. The view has six visual components — a clock face, an hour hand, a minute hand, a seconds hand, a seconds hand shadow and an alarm hand. Almost all the hands have a different offset from the center of the clock face and a different color and line width attribute. For these reasons, a graphics state is retained for each hand, except for the alarm. It has the same offset as the hour hand and so it uses that gstate. Using a graphics state to manage this information eliminates having to translate each hand before drawing. Although this provides a slight performance gain, its principal advantage is that it provides a cleaner, more manageable switching between drawing states.



(The alarm hand can be selected and moved around the dial, although implementing the actual alarm and the voice-activated shut-off has been left as an exercise for the reader.)

3. USER OBJECTS

In the PostScript language examples shown above, graphics state objects are retained as user objects. User objects provide an efficient way to refer to PostScript language objects in the server. The traditional way to refer to objects is to use a *key*, the key normally being a name object. Using names to refer to objects is preferred when writing PostScript language by hand because the dictionary lookup mechanism has been optimized for names. Referring to objects as names quickly becomes burdensome though when referring to PostScript objects in the client program. Static character arrays or string literals can be used and sent as arguments with wraps but this reference system is clumsy when more than a few objects are used. In addition, this approach does not avail itself to handling dynamically created objects. User objects provide the means to tag objects with an integer instead of a name, saving space and providing a system to dynamically name objects.

The example below is taken from the **Dial** application highlighted in Technical Note #5054. This example uses static character arrays to refer to user paths in the server. The character arrays are sent to the server first to define the user path and then again to refer to the user path when rendering.

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 (char *str)
  str ustroke
endps
```

C-language code:

```
static char *upath1 = {"upath1"};

PSWDefineUserPath(pts, i, ops, j, upath1);

PSWDrawUserPath(upath1);
```

The example below shows the sequence used in the **Clock** application to define the user paths as user objects. The user path points and operators are stored in one of the client files as static float array and character arrays. They just as easily could be stored in a section of a Mach-O segment. For each user path, a point and an operator array are sent to the server within a wrap and placed on the stack as a single array of two elements. The procedure, **DPSDefineUserObject( )**, is then used to define the user object. A NULL identifier is passed as argument prompting the procedure to create a new object identifier. Subsequent referrals to the user path descriptions employ the user object.

PostScript language code:

```
defineps PSWSetUpath (float Pts[Tot_Pts]; int Tot_Pts;
        char Ops[Tot_Ops]; int Tot_Ops)
  [Pts (Ops)]
endps

defineps PSWUpathFill(userobject UPath)
  UPath ufill
endps
```

C-language code:

```
int upathHour;     /* Instance variable, NULL initial value */

PSWSetUpath(ptsHour, sizeof (ptsHour)/sizeof (float),
       opsHour, sizeof (opsHour)/sizeof (char));
upathHour = DPSDefineUserObject(upathHour);

PSWUpathStrokeFill(upathHour);
```

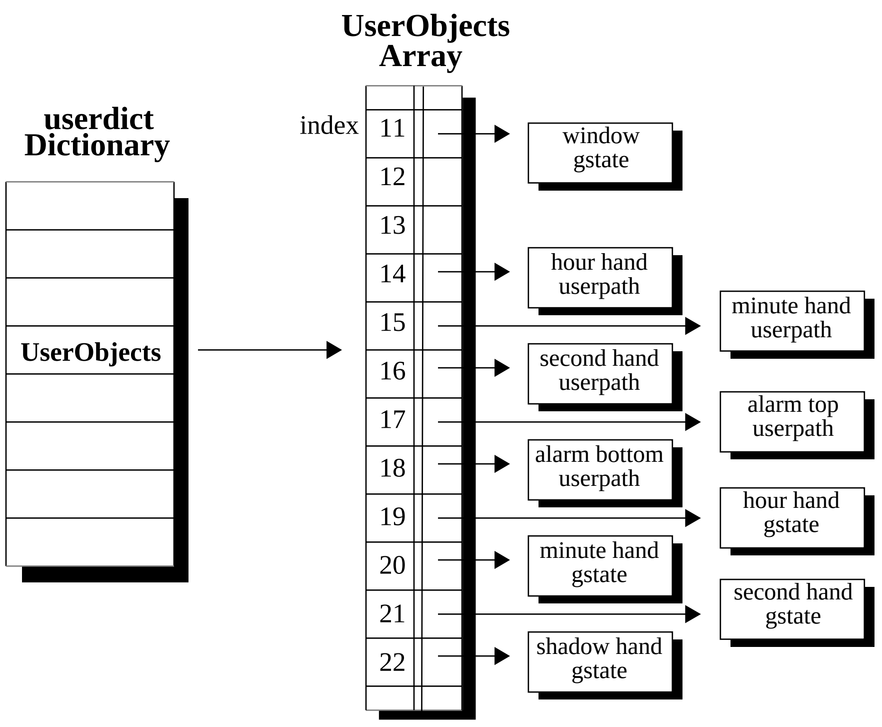
User objects are recommended for gstates, dictionaries, large arrays or strings, user paths and other objects that are stored in the server and that might be directly invoked from the client application. User objects can also identify procedures but procedures are usually called from within wraps. The application programmer might want to keep these as name objects for clarity and easier management of the code. Items that can be retained in the server are discussed in a section that follows.

User objects are stored in the **userdict** dictionary in an array named **UserObjects**. This array is defined as read-only and so specific operators, **defineuserobject**, **execuserobject** and **undefineuserobject**, are available for placing, executing and removing objects in the array. The **defineuserobject** operator takes two arguments, a non-negative integer index and the object to be defined. The object to be defined is placed into the array at the position specified by the index. If the index already exists, the new object replaces the existing object. If the index does not exist, then the number of entries in the array is extended to include the new index. The **UserObjects** array is created in private VM.

In the NeXT implementation, the **defineuserobject** operator should not be called directly. Many Application Kit methods define user objects and integer keys chosen by the application programmer may overwrite those already used by Application Kit methods. The NeXT procedure, **DPSDefineUserObject( )**, should be used instead to create a new identifier. This procedure handles the allocation of new identifiers as well as the recycling of existing identifiers. A zero should be passed to the procedure to create a new identifier. The current identifier should be passed to redefine an existing one. When a user object identifier is assigned to another PostScript language object, the previous object is made available for garbage collection provided no other references to the object exist. As a result, the previous object does *not* need to be explicitly freed with the **undef** operator.

The user objects for graphics state objects for the hands of the **Clock** are not redefined. When the graphics state for a hand changes, the new structure is copied into the old structure. This can be done because a graphics state is a composite object and the user object retains a pointer to the structure.

Below lies the **UserObjects** array for the **Clock** application. Entries 0 -13 have been allocated by Application Kit objects used in the application.



If user object identifiers are passed in single operator and wraps as type **userobject**, the Client Library automatically performs an **execuserobject** on the identifier, eliminating any need to perform an explicit **execuserobject**.

PostScript language code:

```
defineps PSWUpathFill(userobject UPath)
  UPath ufill  % An execuserobject is unneccessary
endps
```

3.1 Printing Issues with User Objects

The NeXT print driver provides user object emulation. The only caveat is that the user objects must be defined in the page description in order for them to be used. The definition can occur in one of three sections in the description: the setup section for the document, the setup section for each page or the script for each page. (Please refer to the *Document Structuring Conventions Specification* for a further description of these sections.)

The document setup section is the optimal location for the definitions. This section is designed to have a global scope. Anything defined in this section can be referenced throughout the document.

Defining user objects within the page setup section brackets the definitions within the page level **save**/**restore** pairing. Because the page **restore** removes the definitions, they have to be defined in the page setup section of every page. (The page level pairing ensures that a given page can be extracted from the document, combined with the document prologue and document setup and printed successfully.)

Defining user objects within the page section not only runs into the page level **save**/**restore** pairings but also any downloadable font **save**/**restore** pairings. (Placing **save**/**restore** constructs around downloadable fonts reclaims PostScript server virtual memory allowing for the use of an unlimited number of downloadable fonts.) User objects defined within a page section would not only have to be redefined for each page but potentially for each set of downloadable fonts. (Three or more fonts can typically be included within a **save**/**restore** pairing.)

Note*: Defining user objects in the prologue creates problems for printing managers that strip prologues out of files. Some print managers store procedure sets in printer memory and therefore strip out prologues in order to print faster and save space.*

4. GRAPHICS STATES

Painting operators such as **stroke** and **fill** and character rendering operators such as **show** and **xyshow** cause an image to be transferred to the page or screen. These operators make use of a number of parameters in deciding what pixels to turn on and which ones to leave off. This set of these parameters make up what is called a graphics state. The current graphics state defines the setting in which printing operators execute. A few of these parameters are the current device, the current transformation matrix (the matrix that maps positions from user space to device space), the current color, the current line width, the current point, the current font and even the current path. The table below contains the complete list of parameters that make up a graphics state and their default values.

Graphics State Parameters

(Initial values in italics)

device
A set of internal primitives for rendering graphical objects in a particular area of raster memory.
Specified at context creation time.

transformation matrix
The matrix that maps positions from user space to device space.
The default CTM.

path
The path that would be rendered by a fill or stroke operation.
No presumed value.

clipping path
The path that defines the current boundary against which output is cropped.
The frame rectangle for a Window or View in the NeXT Application Kit.

position
The current position in user space also know as the current point.
No presumed value.

font
The set of graphic shapes (characters) that define the current typeface.
No presumed value.

color
The color to use during painting operations. Several different color models can be specified.
0 - Black.

halftone screen
A collection of objects that define the halftone screen pattern for gray and color output.
A device-dependent, type 3 halftone dictionary.

flatness
A number that reflects the accuracy with which curves are to be rendered. Smaller numbers give smoother curves at the expense of more computations.
1.0

miter limit
The limit of length of the line joins for line segments connected at a sharp angle.
1.0

stroke adjustment
A boolean value that determines whether automatic stroke adjustment is on or off.
true

line width
The thickness in user coordinates of lines to be drawn by the stroke operator.
1.0 in user space coordinates.

line cap
A number that defines the shape of the endpoints of any open path that is stroked.
0 - square butt end.

line join
A number that defines the shape of joints between connected segments of a stroked line
0 - mitered joins.

dash pattern
A description of the dash pattern to be used when lines are rendered by the stroke operator.

***alpha**
A value that represents the amount of transparency the current color will have.
1.0 - opaque.

***instance drawing mode**
A boolean value that determines whether instance drawing is on or off.
false

Source: The PostScript Language Reference Manual

*NeXT specific

The NeXT toolkit drawing guidelines instruct that the current graphics state be the same leaving a **View** object as when entered except for the color, position, path, font and line width. Since the **display** method of a view encapsulates the **drawSelf::** message between **gsave** and **grestore** operations, the restoration of the graphics state is automatic when using **display**. Any drawing that bypasses the **display** mechanism needs to take their own measures to restore the graphics state to its initial state after any drawing. A view will inherit the graphic state left by any subviews so it is important to make sure the correct parameters are set before drawing. In addition, a view needs to explicitly set the color, position, path, font and line width since these parameters are not guaranteed to have a specific value.

gsave, grestore and the Graphics State Stack

Managing graphics states in devices without graphics state objects is a matter of pushing and popping graphics states onto and off of the graphics state stack with **gsave** and **grestore** operations. The graphics state stack is appropriate in a printing environment since the drawing is highly structured. **gsave**s and **grestore**s isolate major changes in the current graphics state, with minor changes accomplished by setting and resetting particular parameters. Most states in a page description are based on the previous state and are used for only one instance. It makes little sense to initialize them, store them elsewhere and then install them when needed as is done with graphics state objects. In a page description, it is most often easier to change the parameters as the drawing proceeds.

***Note**: In some cases, it is more efficient to use the operand stack to save and reset previous parameter values than it is to use a **gsave/grestore** nesting. For example, a **currentlinewidth/setlinewidth** could encapsulate a change to the line width without the need for a **gsave** and **grestore**.*

```
currentlinewidth 0.5 setlinewidth stroke setlinewidth
```

4.1 Graphics State Objects

Changing the parameters in a graphics state is a larger issue in a display environment. Many graphical objects are drawn repeatedly within the same or different windows with each object sharing little or no resemblance to other objects. The clipping paths, coordinate systems and drawing attributes can be quite different from one window or view to another window or view. It is therefore more efficient to retain graphics states in the PostScript sever virtual memory for significant and frequently invoked drawing objects. When the objects are drawn, the graphics state can be installed with one operator instead of having to store and reset each individual parameter. Graphics state objects offer an attractive alternative to the **gsave**, **grestore** and graphics state stack method of managing graphics states.

In the NeXT Application Kit, graphics state objects are used principally for **Window** objects. These objects often have distinct coordinate systems as well as clipping paths. In addition, windows reserve space in raster memory. The *current device* component in the graphics state serves as the link to this memory. When the gstate is set with a **setgstate** operation, the device in the **Window** object’s gstate is made the current device, installing the raster memory for the window as the drawing buffer. Using a gstate for a window provides a very convenient and efficient way to install a coordinate system, a clipping path and the window’s raster memory.

Graphics state objects are optional for **Views**. Since views do not reference raster memory, they do not benefit from gstates as significantly as windows. The principal reason for using a gstate for a view is to install a particular graphics state at the time the view is focused instead of inheriting the state from the previous view. In other words, a graphics state can be allocated and initialized with a specific set of parameters. When the view is focused, the parameters in its graphics state are installed as the current parameters. Although more rare, graphics states can also be used for images within views. This approach is used for the hands of the clock, making the drawing of each hand easier and slightly faster. Each hand has a separate translation and color. A graphics state is used to retain these values so that they do not have to be set explicitly before each hand is drawn.

A graphics state object takes up a couple hundred bytes of virtual memory (with a null current path) and so there is some appreciable cost to a graphics state object. The **gsave**, **grestore** and the graphics state stack should still be used drawing images that are tightly connected and share many but not all of the same graphics state parameters. An example is the display of a group of graphics within a drawing program. Graphics state objects here are not advised because only a few parameters are likely to change from object to object. In addition the parameters that do change are likely to change frequently requiring the resetting of the graphics state object in the server. (Plus, creating a page description that could be printed to devices without the Display PostScript system extensions requires a relatively inefficient emulation of the gstate operators.)

Because the path and clipping path are saved in the graphics state, it is important to make sure these parameters are set to their desired values when the graphics state is defined. Defining a graphics state with a non-empty path will install the path as the current path when the graphics state is made the current graphics state. Any subsequent rendering operator prior to a **newpath** operator will render that path in the current color, line width, etc. The same treatment applies to a clipping path. Any non-empty clipping path will clip subsequent drawing. Although a graphic state object can be used to retain a path, it is not desirable to use it for this purpose. Using a cached user path or an offscreen buffer is a better approach for painted paths.

A good example of graphics state objects in action is when scrolling. The movement of the scollbar must coincide with the scrolling of the text or graphic. This is done by quickly swapping between the **Scroller** and the **ClipView**. A graphics state object is used for the **ClipView** which allows the quick installation of the clipping path and the coordinate system with a single operation. This installation provides a critical performance advantage over creating the clipping path and making the coordinate system adjustments individually.

4.2 Allocating Graphics States in Views

The default instance of a view does not have a graphics state but one can be allocated with the **allocateGState** method of the **View** class. If a graphics state is specified, it is installed as the current graphics state during the **lockFocus** message. If no graphics state is specified, then the window's graphics state is installed and then adjusted to reflect the view's coordinate system and clipping path.

Below lies the PostScript language operations sent to the server for a custom view contained in a window. (These instructions are sent within methods of the View class. They do not need to be sent by the application developer.) The view object has set the clipping to *YES* but has not allocated a graphics state. The **lockFocus** method performs a **gsave** and then installs the window's graphics state as the current graphics state. (The Application Kit stores the gstate in the server as a user object.) A clipping path is installed around the bounds of the view and then the user space is translated to the lower left of the view's bounding box. The **flushgraphics** and **grestore** operator following the drawing instructions is invoked in the **unlockFocus** method.

PostScript language trace:

```
gsave
11 execuserobject setgstate
52 38 302 152 rectclip
52 38 translate

< drawing instructions placed within drawSelf:: would appear here >

flushgraphics
grestore
```

In the case where a view allocates a graphics state, the PostScript language instructions would look like those below. The view's graphics state is installed, which in this instance is stored as user object 20, instead of the window's graphics state. No clipping path or translation occurs because these values are a part of the graphics state object.

PostScript language trace:

```
gsave
20 execuserobject setgstate

< drawing instructions would appear here >

flushgraphics
grestore
```

4.3 Initializing a View’s Graphics State

If many of the graphics state parameters differ from those inherited upon focusing, the view should consider allocating a graphics state object and initializing the graphics state parameters. (The key word here is *frequently*. A graphics state object will produce little benefit for an infrequently drawn view.) Regardless of whether or not a graphics state is allocated, any changes made to the graphics state in the course of drawing are not saved unless the graphics state is redefined with a **currentgstate** operation. (Of course, only the view’s gstate should be redefined. The window’s gstate should be left alone.)

The parameters of a graphics state object can be initialized with a two step process. The first step is to override the **initGState** method in the View class. The appropriate single operator or wrap calls to configure the graphics state parameters should be placed in this method. The second step is to allocate the graphics state. This can be done by invoking the **notifyToInitGState** method with a *YES* argument and then allocating a gstate by messaging the **allocateGState** method.

The message to **notifyToInitGState** method causes the **initGState** method to be invoked when a gstate is created. A gstate is created when it is first needed which is usually when the view is next displayed. The example below shows the sequence of steps to initialize a graphics state. The PostScript language trace that results is generated from a combination of the **View** methods and the single ops placed in the **initGState** method.

C-language code:

```
/*
 * Called next time the view lockFocuses after
 * receiving the notifyToInitGState message
 */
- initGState
{
  PSsetlinewidth(2.0);
  PSsetgray(0.5);
  PSsetlingjoin(2);

  return self;
}

/*
 * (placed either in the newFrame: method or
 * in some other initial method)
 */
  [self notifyToInitGState:YES];
  [self allocateGState];
```

PostScript language trace:

```
gstate
19 exch defineuserobject
2.0 setlinewidth
0.5 setgray
2 setlinejoin
19 execuserobject currentgstate pop
```

In the trace, the **gstate** operator creates a gstate object. The second line defines the gstate as a user object. The next three lines set the gstate parameters (the results of the single operator or wrap calls in the **initGState** method). The last line copies the new gstate into the old gstate structure. The user object identifier points to the same structure as before so the user object does not need to be redefined. The **pop** removes the gstate left on the stack from the **currentgstate** operation.

Redefining a view's graphics state object is a matter of locking the focus to the view, changing the gstate parameters (by making single operator or wrap calls), copying the current gstate into the old gstate structure with the **currentgstate** operator, removing the gstate from the stack with the **pop** operator and then unlocking the focus. These steps produce the same trace as above except that the old graphics state is installed as the current graphics state and not allocated anew. The source code and the PostScript language trace are provided below.

C-language code:

```
[self lockFocus];
  PSsetlinewidth(0.5);
  PSsetgray(0.25);
  PSsetlingjoin(1);
  PScurrentgstate([self gState]);
  PSpop();
[self unlockFocus];
```

PostScript language trace:

```
gsave
19 execuserobject setgstate
0.5 setlinewidth
0.25 setgray
1 setlinejoin
19 execuserobject currentgstate
pop
grestore
```

4.4 Multiple Graphics States

The **View** class can have only one graphics state. A subclass of **View** can store additional gstates as instance variables. The subclass of **View** in the **Clock** application uses four gstates to maintain the origins, color and line widths for each hand. (The rotations are passed to the wrap and are performed after the gstate is installed but before the path is rendered.) The coordinate systems of the gstates are translated to the center of the dial plus some offset to provide depth to the clock hands. The scale is the same as the view's scale. Scaling the view with the scale method and then redefining the gstates for the hands captures the scale of the view in the scale of the gstates. This technique takes the place of either scaling each gstate separately or scaling the points of the user paths.

The section below, taken from the **Clock** application, defines the graphics states. This step is done at initialization and whenever the window is resized (resizing triggers a scaling of the view and a redefinition of the graphics states). The steps described in the previous section are *not* used to initialize the graphics states. These steps initialize the graphics state for the *view* and not the graphics states for the *hands*. Also, defining the graphics states does not occur within the **newFrame:** method. The view's window does not exist at the time of the **newFrame:** method and so a **lockFocus** message would not install a graphics state to use as a basis. The definition of the graphics states occurs after the view has been installed in a window. (A subclass of **Application** creates a **ClockView** object, installs it in a window and then calls the method to define the graphics states.)

C-language code:

```
/* Initialize the user object identifiers to be zero. When passed
 * to DPSDefineUserObject(), a unique user object identifier will
 * be assigned.
 */
+ new
{ self = [super new];  . . .
  gstateHour = gstateMin = gstateSec = gstateShad = 0;  . . .
  return self;
}

/* Create a gstate and define a user object for it if one does not
 * exist already. If a gstate does, then copy the new gstate into
 * the old structure. No need to redefine the user object because
 * it still refers to the same structure. (This only takes place
 * in the case of a composite object such as an an array.) The
 * PSpop() pops the result of PScurrentgstate() off of the stack.
 */
static int  definegstate(gstate, offsetx, offsety, color, linewidth)
  int  gstate;
  float offsetx, offsety, color, linewidth;
{
  PSgsave(); PSWSetGstate(offsetx, offsety, color, linewidth);
  if (!gstate) {
    PSgstate(); gstate = DPSDefineUserObject(gstate);
  } else {
    PScurrentgstate(gstate); PSpop();
  }
  PSgrestore();
  return gstate;
}
```

```
/* Called at initialization (after the view has been installed
 * in a window) and when the window is resized (invoked in the
 * sizeTo:: method of ClockView).
 */
- defineGStates
{
  NXPoint   center;

  center.x =  floor(bounds.size.width/2);
  center.y = floor(bounds.size.height/2);

  . . .
  [self  lockFocus];
    . . .
    gstateHour = definegstate (gstateHour, center.x, center.y,
      CLRHANDS - 0.2, LNWIDHANDS);

    gstateMin = definegstate(gstateMin, center.x + OFFSETHANDSX,
      center.y + OFFSETHANDSY, CLRHANDS - 0.2, LNWIDHANDS);

    gstateSec = definegstate(gstateSec,
      center.x + (2 * OFFSETHANDSX),
      center.y + (2 * OFFSETHANDSY),
      CLRSECOND, LNWIDSECOND);

    gstateShad = definegstate(gstateShad,
      center.x + (2 * OFFSETHANDSX) + OFFSETSHADX,
      center.y + (2 * OFFSETHANDSY) + OFFSETSHADY,
    CLRSHADOW, LNWIDSECOND);
  [self  unlockFocus];
  return self;
}
```

4.5 Printing Issues with Graphics States Objects

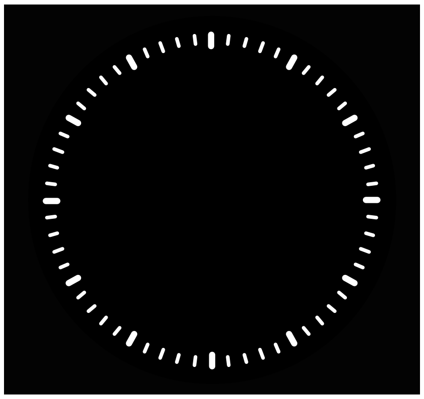
Unlike the **Clock** application, most applications will want to create a page description to be used for printing. As a result, any application that uses graphic states within a page description will need to create an external representation of a graphics state. One example is to store the values of the parameters in an array. When defining a graphic state object, the current graphic state parameters would be placed into the array. When setting a graphic state object, the values in the array would be taken out of the array and made the current graphic state parameters.

Any emulation scheme for graphic state objects has the same problems as the emulation of user objects when used in a page description. The location of the definition within the page description sections can have certain repercussions depending on the presence of any **save**/**restore** pairings. Any definition that occurred within a **save**/**restore** pairing would not exist after the execution of the **restore**.

Because of the memory management issues and performance issues relating to the suggested emulation scheme, graphic state objects are not recommended for use within page descriptions. The **gsave**/**grestore** approach is a much more desirable paradigm for managing graphic states when printing.

5. OFFSCREEN BUFFER

In the **Clock** application, the clock face is only drawn at initialization and whenever the window is resized. Instead of drawing directly into the view, the application draws the clock face into an offscreen buffer. For each tick, the offscreen bitmap is first composited into the window and then the hands are drawn atop the image in the window. The clock uses a window with a buffered backing so the window is not flushed onto the screen until all the drawing has been completed.



C-language code:

```
/* Invoked at initialization and whenever the window is resized.
 * The frame size of the view is used instead of the bounds
 * size because the bounds size does not reflect the correct
 * size after scaling relative to an outside object.
 */
- drawFace
{
. . .
  if (bitmapId)
  {
    [bitmapId  getSize:&bmapsize];
    if (bmapsize.width < frame.size.width ||
          bmapsize.height < frame.size.height)
      [bitmapId  resize:frame.size.width  :frame.size.height];
  }
  else
  {
    bitmapId = [[Bitmap newSize:frame.size.width
        :frame.size.height    type:NX_NOALPHABITMAP]  setFlip:NO];
  }
. . .
  [bitmapId lockFocus];
    /* Draw the Clock Face. */
  [bitmapId  unlockFocus];
. . .
  return self;
}

/* Invoked whenever the clock is drawn. Again the frame
 * size is used because it provides the correct size
 * from the bitmap’s point of view, regardless of the
 * scale of the view.
 */
- drawSelf:(NXRect *)r :(int)count
{
. . .
  NXRect    cRect;

  cRect.origin = bounds.origin;
  cRect.size = frame.size;
. . .
  [bitmapId composite:NX_COPY fromRect:&cRect
      toPoint:&bounds.origin];
. . .
  return self;
}
```

It would be impractical to use an offscreen buffer to hold a simple image like a rectangle because a **rectfill** is a very simple and efficient operation. Offscreen buffers are recommended for complex images that remain unchanged for a considerable amount of time. The clock face takes approximately 6/10 of a second to draw a 400 by 400 point image. Since it remains the same unless the window is resized, it makes sense to draw it into an offscreen buffer and then composite it into the window before drawing the hands.

(The NeXT programming guidelines recommend using NX_NOALPHABITMAP wherever possible instead of NX_UNIQUEBITMAP and NX_ALPHABITMAP in order to reduce space requirements.)

Offscreen buffers for the hands are not appropriate because the hands are rotated. Since bitmaps cannot be rotated easily, an offscreen buffer would be necessary for every position of every hand. This comes out to a total of 219,720 buffers. Storing the user paths descriptions for the hands in the server provides the optimal solution to imaging the hands.

5.1 Printing Issues with Offscreen Buffers

Bitmaps can be printed using the **image** operator but they are limited to the resolution of the screen. The highest quality results can be obtained for each device by directly executing the PostScript language operators that image the page. Offscreen buffers are a technique that should only be used for performance purposes during interactive display, not for the printing process.

6. STORING DATA IN THE SERVER

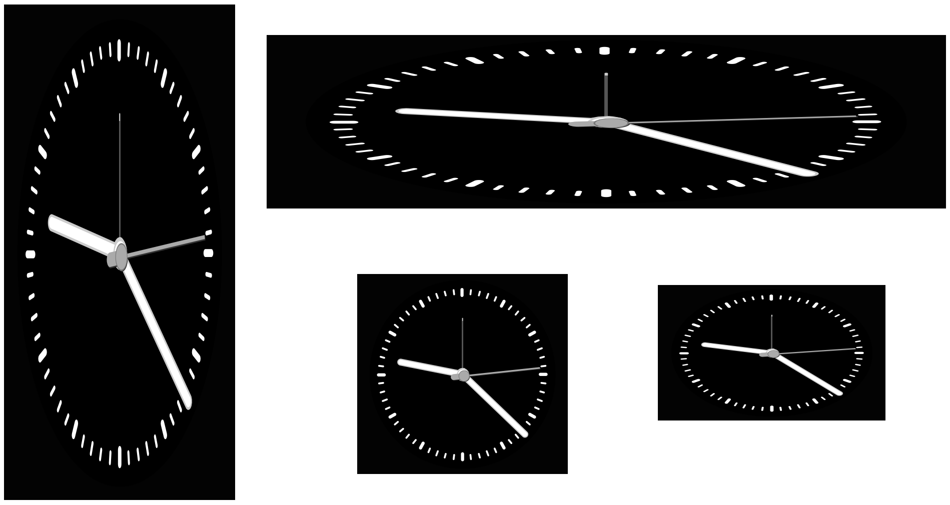
Deciding what items make good candidates for storing in the server is really a matter of judgement. The structure of gstates and dictionaries are not visible outside the server so they must reside in the server. Any application that makes extensive use of gstates and dictionaries might consider user objects as a way to allow for smoother management of these objects.

In addition to PostScript language objects, data structures such as arrays, strings and user paths can be retained in either the client or the server. The determination comes down to several factors – the size of the structure, the frequency of its execution, the number of times the structure changes and the client-server network transmission overhead. In most cases, these structures do not belong in the server. But in some cases, it may make sense to store the data in the server rather than to keep passing it from the client.

The hands of the clock are a good example of objects that can be retained in the server. The hands have moderately sized descriptions, are called repeatedly and do not change. Once the client has sent the descriptions to the server, the client has no other need for the descriptions. Calling a wrap to draw the user path and passing in the appropriate user object identifier is the only subsequent responsibility for the client. Interactive applications such as drawing applications or word processors, though, might find little use for storing data in the server because its data changes too frequently to be worth managing identical copies in both the server as well as in the client.

7. SCALE

The **Clock** application scales the view in which the hands reside in order to scale the hands. The coordinate systems for the hands are based on the coordinate system of the view. As a result, the scale of the view is captured in the redefinition of the graphics state for each hand.



The coordinate system of a **View** object can be altered with the **scale::**, **moveTo::** and **rotate:** methods available in the **View** class. These methods have the same effect on the current transformation matrix in the graphics state object of the view or window as a **scale**, **translate** and **rotate** operation would when performed with a wrap or single op call. Instead of redefining the graphics state to incorporate the new CTM, a matrix is concatenated to the CTM after the graphics state has been installed through a **lockFocus** message.

Note*: These methods change the graphics state relative to its present state. Doubling the scale of a view that has been scaled to one half its scale will return the view to its initial scale. As a result, some instance variables may be needed or some ratios made between the frame size and the bounds size in order to return to an absolute scale.*

The code segment below shows how the **scale::** method is used to distort the clock without having to distort each element in the clock. Resizing a window also resizes the content view, which in this case is the **ClockView**. Because of this action, it is possible to override the **sizeTo::** method in the **View** class and insert a message to the **scale::** method. When the window resizes, not only will the **ClockView** resize but also it will also scale in order to fill the entire content region. This approach replaces either scaling each graphics state individually or scaling the user path descriptions.

C-language code:

```
/* The width and height arguments make up the
 * new frame size. The xframe variable holds the
 * previous size. The ratio of the two provides the
 * new scale.
 */
- sizeTo:(NXCoord)width :(NXCoord)height
{
  NXRect  xframe;
  . . .
  xframe = frame;
  [super sizeTo:width :height];

  if (window)
  {
    [self scale:width/xframe.size.width:height/xframe.size.height];
    [self drawFace];
    [self defineGStates];
  }

  return self;
}
```

The **sizeTo::** method sizes both the **frame** and the **bounds** instance variables while the **scale::** method just scales the bounds. As a result, when the window is made smaller, the **sizeTo::** makes the **frame** and the **bounds** smaller but the **scale::** returns the bounds to its previous size. Whenever the **scale::** method is used, care should be taken to use the correct dimensions relative to a particular object. For resizing the bitmap for the clock face, the **frame** size is used but for moving to the center of the view, the **bounds** size is used. The **frame** size should be used by external objects to obtain the unaffected dimensions while the **bounds** size should be used by the view internally to reflect the scale of its coordinate system.

The **HitDetection** application covered in Technical Note #5057, *HitDetection and Zooming*, uses the **scale::** method of the view to allow zooming.

7.1 Printing Issues with Scaling the Coordinate System

Printing a scaled view will print a scaled image. Applications that make use of a scaled or rotated view for the sake of magnifying and unmagnifying a page or rotating a page will have to set the graphics state of the view to its default scale before rendering a normal sized and scaled image.

8. SUMMARY

User objects and graphics state objects are useful for managing objects within a display environment. A user object is an integer identifier to a PostScript language object. The traditional way to refer to objects is with a name key. Names are easy to manage in PostScript language code but are cumbersome when used in C-language code. User objects provide a system to create and manage integer keys for objects. The procedure, **DPSDefineUserObject( )**, should be used instead of using the PostScript operator, **defineuserobject**, directly. This procedure manages the creation of user objects on an application wide basis. Some Application Kit objects define user objects and so **DPSDefineUserObject( )** prevents the application developer from overwriting existing identifiers.

A graphics state defines the context for graphic operators such as **fill**, **stroke** and **show**. A graphics state object is a graphics state stored in virtual memory. A graphics state object can be installed as the current graphics state by using the **setgstate** operator, circumventing the usual graphics state stack operations. A graphics state is very useful for windows and other items that reserve raster memory because they allow an efficient way to install the raster memory as the current imaging buffer. Graphics states can be beneficial for views and other images that 1) require different graphics state parameter settings than the previously drawn view or image and 2) are frequently focused. Allocating and initializing a graphics state for a view will install the initialized parameters in the current graphics state when the view is focused. A graphics state can perform a similar function for an image within a view.

Offscreen buffers, storing data in the server and scaling a view object are other techniques that are useful in a display environment. Imaging into a bitmap and then compositing the image onto the screen eliminates having to redraw the image with each display. Storing frequently drawn and unchanging user paths in the server means that the data is sent only once and not each time the path is drawn. Scaling the view instead of each image provides an easy way to globally scale a drawing with composite images.