PGS DOCUMENT

Stewart A. Brown
Dennis Braddy

Last Updated: 06/18/2003

Other PACT Documentation

Disclaimer

Part of this work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Introduction

One of the biggest headaches for portability is graphics. The objective evidence is that the field is immature. One cannot exactly say that there are no graphics standards. The real problem is that there are too many standards. Until such time that the world settles down, there will be the need for a tool like PGS.

PGS is an application program interface (API) that is independent of the underlying host graphics system. All of the graphics portability headaches are confined to PGS and applications which use PGS are completely portable. PGS currently sits on top of X Windows on UNIX platforms, Quickdraw on Macintoshes, and Microsoft’s graphics library on DOS platforms.

PGS takes a least common denominator approach regarding what graphics functionality it supports. The goal is to run on the widest variety of machines. This lets out high level graphics capabilities such as real time 3D rotations which depend on specific hardware. On the other hand, any rendering capability that can be implemented with reasonable efficiency in software is fair game for PGS. This model will almost certainly change in time as both graphics hardware and software evolve and become ubiquitous across platforms.

PGS adopts a model in which graphics devices such as display windows and PostScript files are represented by a structure which contains all of their state information. Then PGS can manage an arbitrary number of devices simultaneously and any picture can be drawn to any device.

PGS also structures display surfaces with a viewport defined in normalized coordinates, an enclosing boundary where axes are drawn which is defined as a set of offsets from the viewport window, and a world coordinate system attached to the viewport. The enclosing boundary is useful for obtaining a standoff between rendered data such as line plots and the axes used to measure the rendering.

PGS supports both line and text drawing primitives, line and text attributes, and bit maps for handling images and other cell array data. Most functionality in PGS is either primitive operations such as moves and draws or at a very high level such as axis drawing and the rendering algorithms that it supports. These rendering algorithms have two interfaces: one for "raw" data; and one for PML type mappings. This gives a great deal of flexibility to the application developer.

PGS has the following rendering algorithms currently: 1D line plots; 2D contour plots; 2D vector plots; 2D image plots; 3D wire frame mesh plots (for 2D data sets); and Grotrian diagram plots.

Overview of PGS

PGS has two main goals: to provide a portable interface to various host graphics systems; and to provide high level functionality to applications which most host graphics systems do not provide. To meet these goals it was necessary to develop a model of the kinds of devices which the various host graphics systems support so that the functional interface could be defined and implemented.

It was also necessary to identify which graphics primitives to support. Some host graphics systems have a very rich supply of graphics primitives. In fact, some go way beyond supplying primitive graphics functionality and provide high level rendering capabilities. Other host graphics systems have a relatively small set of primitive graphics operations. The decision was made to design PGS so as to depend on as small a subset of graphics primitives as possible. This makes PGS extremely portable. It also forces PGS to either implement or forego higher level graphics functionality. Hopefully, a useful balance has been struck on this issue.

As the above discussion implies, PGS has two obvious layers. The first layer is a low level one that communicates directly with the host graphics system. The other layer is oriented more toward the application and includes the high level rendering and axis drawing functions. This layer actually breaks down into several layers. The details of this breakdown will be discussed as appropriate.

The remainder of this section discusses the PGS device model and the PGS drawing model.

The PGS Device Model

To provide the maximum degree of portability in what is an inherently platform dependent field, the attempt has been made to isolate all of the platform dependencies behind abstraction barriers. The functional interface provides one set of abstraction barriers. This however does not help with the problem of the notion of graphics state. Graphics state consists of information such as the current default line attributes, default text attributes, coordinate systems, and so on. Furthermore, in a general purpose setting, an application may wish to manage multiple independent devices each with its own separate graphics state.

First, PGS defines a graphics device as an abstract entity containing a logical two dimensional drawing surface and a set of parameters which describe how and where all drawing functions are to be performed on the drawing surface.

Nearly all host graphics systems have their own abstraction barrier wrapped around their drawing surface(s). However, they almost all provide a pointer or index to applications to specify which drawing surface is intended for a particular operation. This latter point is most relevant to windows on display screens, although a file indicator is the correct analog for PostScript or CGM type devices. In any case, PGS handles the interface to the host graphics system and hides it from applications. In the place of the host graphics device indicator, PGS supplies a structure called a PG_device. Applications open and manipulate PG_device's only. This way all PGS based applications have a single portable interface to all devices supported by PGS.

The PG_device not only contains the host graphics device indicators, it also contains the graphics state for each device. In this way each PG_device is independent of every other PG_device. This gives applications the ability to draw the same picture on every device by simply changing the PG_device passed to the drawing functions. No device conditional logic is required of applications.

The host graphics systems supported by PGS currently are:

X Windows

PostScript

CGM (Computer Graphics Metafile)

Quickdraw (Apple Macintosh)

Microsoft Graphics Library

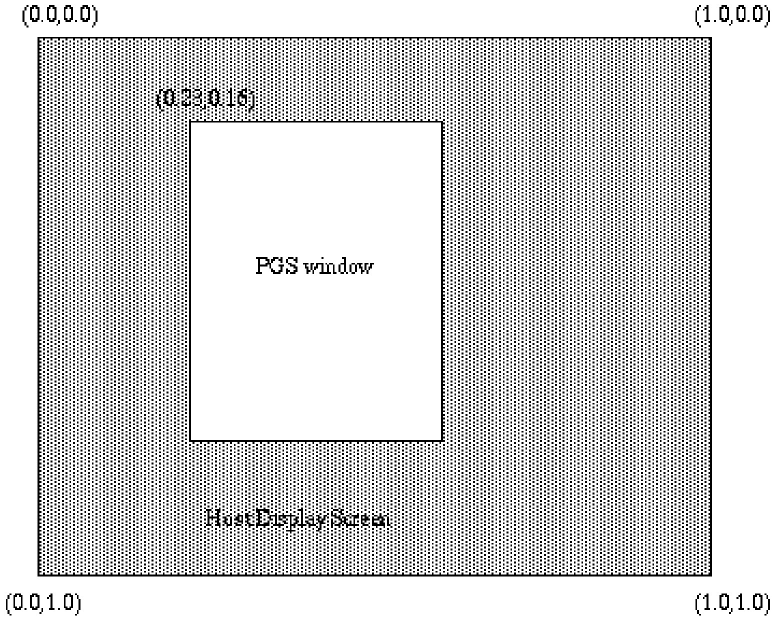
The PGS Drawing Model

Almost all host graphics systems employ a drawing model. This specifies information such as coordinate systems and their origins, how clipping is done, and so on. PGS has a somewhat more difficult time coming up with a drawing model since it must present a drawing model that is compatible with all host graphics systems even when the various host graphics systems are in conflict with one another. PGS accomplishes this task by using the least common denominator of the host systems, defining as much of the drawing model as possible, and mapping host graphics systems models into the PGS model.

A PG_device can be thought of as a window on the display area of the host graphics system. In that view, a PGS window can be defined as the region of the host graphics system display surface controlled by PGS during drawing operations. PGS windows then map naturally onto the kinds of windows associated with such host graphics systems as X Windows and Quickdraw. This idea also has application to a PostScript or CGM device.

PGS Window Placement

In placing a PGS window which contains the drawing surface on a display screen or a PostScript page, the position of the upper left corner of the PGS window is given in normalized coordinates relative to a coordinate origin in the upper left corner of the host graphics system device.

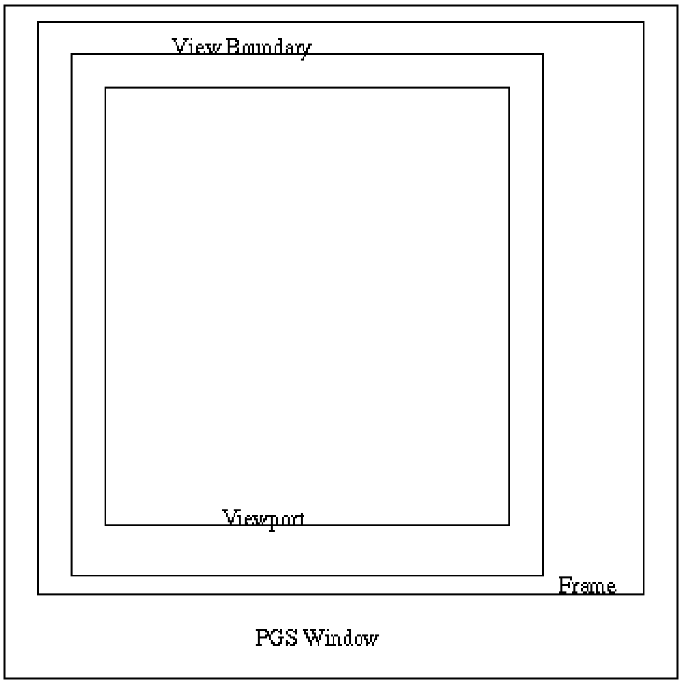


Frames, Viewports, and View Boundaries

Any part of the interior of a PGS window may be drawn on by PGS routines. Windows do have some additional structure to help applications conveniently handle high level plotting constructs.

A window may be partitioned into frames with a view to drawing more than one plot at a time. Within each frame there is a preferred drawing area called the viewport. The viewport is defined relative to its enclosing frame and by default each window has a single frame which is the same size as the window. PGS supports clipping which can render it impossible to draw to any part of the window but the current viewport.

In addition to the viewport there is a bounding region of the viewport which is used to offset axes from the viewport so that there is a nice, application controllable separation between the axes and whatever is drawn in the viewport. This bounding region is called the view boundary.



The viewport and view boundary are tied together. When an application defines the viewport, the view boundary is implicitly defined in terms of offsets from the viewport. These are referred to as topspace, leftspace, rightspace, and botspace.

The application can move the frame or viewport around in the PGS window at any time as well as altering its size.

Coordinate Systems

Inside a PGS window there are three coordinate systems: world coordinates, normalized coordinates, and pixel coordinates. World coordinates are application defined and have whatever meaning the application requires. The lower left corner of the viewport corresponds to the minium x and y values of the world coordinate domain. Normalized coordinates represent the fraction of the PGS window width and height that a point is from the origin which is in the lower left corner of the PGS window. Pixel coordinates represent the integer number of pixels that a point is from the origin which is in the lower left corner of the PGS window.

PGS supplies a set of macros to convert between all of the coordinate systems which a PGS window may have.

The PGS User Interface Model

The subject of user interfaces can be a rather complicated one. For many applications the most natural interface is a graphical one. However, some systems make the mistake of insisting that a graphical interface is the only interface which an application may have. PGS supports a model of user interfaces that permits the application developer to seek the natural interface for his or her application. This means that PGS supports development of textual interfaces, graphical interfaces, and hybrids. In the following sections, we will discuss the concepts underlying user interfaces from PGS's point of view (PGS emphasizes portability and flexibility especially).

A Textual Interface

Consider the following program fragment which is typical of an application with a textual interface:

```
char s[MAXLINE], *t;

char *dispatch(char *s);


printf("-> ");

while (fgets(s, MAXLINE, stdin) != NULL)

   {t = dispatch(s);

    printf("%s\n-> ", t);};
```

This code prints a prompt, gets some input (fgets), processes it (dispatch), and prints the result. It does this in a loop until something ends the program.

For a program so simple and ordinary, it is astonishing how difficult some systems make it to run this code. One of the peg points of PGS is that it must be "easy" to run such an application whether the system wants to make it easy or hard. Some of the graphical user interface models can be ported with some careful abstraction barriers (and PGS does this too), but this example is something of a lowest common denominator. So we will start with this and build up a model that supports this simple text driven style and the most elaborate graphical application.

Adding Abstraction Barriers

By adding two macros and two function pointers we can make an enormous shift in the portability of this program:

#define GETLN (*getln)

#define PRINT (*putln)

char *(*getln)(char *s, int n, FILE *fp)

int (*putln)(FILE *fp, char *fmt, ...)

These items are defined in score.h which is #included by pgs.h

With these elements we can modify the original example as follows:

```
char s[MAXLINE], *t;
char *dispatch(char *s);


getln = fgets;
putln = fprintf;


PRINT(stdout, "-> ");
while (GETLN(s, MAXLINE, stdin) != NULL)
    {t = dispatch(s);
     PRINT(stdout, "%s\n-> ", t);};
```

This doesn't look like much at all, but the impact of this change is enormous! Now we can insert other functions which are call compatible with the standard C library functions, fgets and fprintf. PGS supplies two such functions: PG_wind_fgets and PG_fprintf. In fact, when a call to PG_open_device or PG_open_console is made these functions are connected to getln and putln for you!

PG_wind_fgets, in addition to looking for input from the terminal as fgets does, also looks for events from the windowing system under which the application is running. PG_fprintf prints your formatted text to a terminal or to a screen window depending on what is appropriate to the system on which the application is running.

This example can be filled out to a complete program (modulo the definition of the dispatch function) which is completely portable:

```
#include
main(int c, char **v)
   {char s[MAXLINE], *t;
    char *dispatch(char *s);
```

```
PG_open_console("test", "COLOR", TRUE, 0.1, 0.7, 0.5, 0.3);
PRINT(stdout, "-> ");
while (GETLN(s, MAXLINE, stdin) != NULL)
   {t = dispatch(s);
    PRINT(stdout, "%s\n-> ", t);};
return(0);}
```

There is an important issue remaining here and that is the subject of the next section.

Multiplexed I/O and Interrupt Driven I/O

In the program we have been discussing, input is gathered from either the terminal or the windowing system. This is an example of multiplexed I/O. Many applications use multiplexed I/O. It is common in networking programs for example. With multiplexed I/O a variety of input sources are polled to see whether there is any input ready. Depending on the device that has input, the application takes the appropriate action as it becomes available. In the more efficient applications the operating system is usually involved since it is better able to control machine resources than any application.

Our sample program doesn’t necessarily need to do multiplexed I/O (on the other hand we haven’t said what the dispatch function does!). If it were a graphical application however the chances are that it would have to handle input from both the terminal and from the windowing system. GETLN, more specifically PG_wind_fgets, does just that. If the specified FILE pointer is stdin, it obtains input from either source, and copies terminal input into the buffer passed in as an argument or dispatches input from the windowing system (also referred to as events) to functions which are registered with PGS to handle specific kinds of events. If the specified FILE pointer is in fact something beside stdin it simple performs an "fgets" on that file. PG_wind_fgets only returns when a newline or an end of file condition is encountered. For terminal input this means typing a carriage return.

It appears that input can only be handled when GETLN is called. However, it is often desirable to have input handled whenever it comes in. Some operating systems support this through the use of assignable interrupts. Input handled this way is said to be interrupt driven. In PGS, when a screen window is opened all of the machinery is put into place to allow interrupt driven I/O. The application switches interrupt handling on and off through the macro PG_IO_INTERRUPTS which take a value of TRUE to turn it on and FALSE to turn it off. When I/O interrupts are on input from the terminal is saved in a buffer to be copied into the buffer of the next GETLN call, and input from the windowing system is dispatched to the appropriate event handler. After the input is processed the interrupt handler returns and execution resumes from the point where the interrupt occurred.

With interrupt driven I/O activated, our simple program has all the capabilities of a vastly more complicated application written for certain specific operating environments which enforce a graphical interface only mode of programming. The fact that PGS runs on such systems as wells as those that support text only or hybrid interfaces should give some idea of the idea of portability and flexibility which PGS aims to provide.

Event Handling

Now that we have seen how input is handled in the broadest terms and how PGS presents a portable application interface for input handling, let's turn to an closer examination of the way in which input from a windowing system is dealt with. Generically, window input is said to consist of sequences of events. Events can be such things as key presses when the mouse or locator is in a window, mouse button presses and releases, or the locator entering or leaving a window.

Different windowing system define varying sets of events. PGS supports the following set of events everywhere:

KEY_DOWN_EVENT	a key on the keyboard is pressed
KEY_UP_EVENT	a key on the keyboard is released
MOUSE_DOWN_EVENT	a mouse button is pressed
MOUSE_UP_EVENT	a mouse button is released
UPDATE_EVENT	the window system says that the window has changed in some way
EXPOSE_EVENT	the window has become fully visible (is no longer obscured by another window)
MOTION_EVENT	the mouse has moved in the window

It should be understood that all events have a context. They all happen in or relate to a particular screen window. So when PGS get notified by the windowing system that there is an event present, it determines which window is effected. It then passes both the pointer to the effected window and the event on to the function which is going to handle the event.

Event Handling Functions

Given the above list of recognized events, PGS defines a function pointer (hook) associated with each type of event so that applications may control what is done with specific events. The following functions let applications assign their function to these hooks.

C Binding: PFByte PG_set_key_down_event_handler(PG_device *d, void (*fnc)())
F77 Binding: integer pgsekd(integer d, fnc)
SX Binding:

C Binding: PFByte PG_set_key_up_event_handler(PG_device *d, void (*fnc)())
F77 Binding: integer pgseku(integer d, fnc)
SX Binding:

C Binding: PFByte PG_set_mouse_down_event_handler(PG_device *d, void (*fnc)())
F77 Binding: integer pgsemd(integer d, fnc)
SX Binding:

C Binding: PFByte PG_set_mouse_up_event_handler(PG_device *d, void (*fnc)())
F77 Binding: integer pgsemu(integer d, fnc)
SX Binding:

C Binding: PFByte PG_set_update_event_handler(PG_device *d, void (*fnc)())
F77 Binding: integer pgseup(integer d, fnc)
SX Binding:

C Binding: PFByte PG_set_expose_event_handler(PG_device *d, void (*fnc)())
F77 Binding: integer pgseex(integer d, fnc)
SX Binding:

C Binding: PFByte PG_set_default_event_handler(PG_device *d, void (*fnc)())
F77 Binding: integer pgsedf(integer d, fnc)
SX Binding:

These assign the specified function fnc to be the event handler for the device, d. The function fnc is a pointer to a function returning nothing which takes a PG_device pointer and a PG_event pointer as arguments.The C routines all return the old value of the hook and the following typedef applies:

```
typedef void (*PFByte)();
```

To explicitly call these functions in a generic way (i.e. regardless of the specific function attached to the hook) use the following macros:

void PG_handle_key_down_event(PG_device *d, PG_event *ev)

void PG_handle_key_up_event(PG_device *d, PG_event *ev)

void PG_handle_mouse_down_event(PG_device *d, PG_event *ev)

void PG_handle_mouse_up_event(PG_device *d, PG_event *ev)

void PG_handle_update_event(PG_device *d, PG_event *ev)

void PG_handle_expose_event(PG_device *d, PG_event *ev)

void PG_handle_default_event(PG_device *d, PG_event *ev)

These simply call the specified event handler with the PG_device and PG_event. The default event handler is an additional way to handle events. The application can have a single handler for all events. For example, when PGS gets a mouse down event it first checks to see whether there is a mouse down handler. If so it is called. If not it then checks to see whether there is a default handler and if so calls it.

Event Related Macros

In addition to the above which have to do with routing events off to handlers, there are some other macros which the event handlers or user call-back functions can use to access event information.

C Binding: void PG_GET_NEXT_EVENT(PG_event ev)
F77 Binding:
SX Binding:

Does a blocking read of the next event from the windowing system and fills in the specified PG_event structure, ev.

C Binding: void PG_KEY_EVENT_INFO(PG_device *d, PG_event ev, int *x, int *y, char *bf, int *n, int *mod)
F77 Binding: integer pgqkbd(integer d, integer x, integer y, integer c, integer mod)
SX Binding:

Return the state of the keyboard for the specified window, d. The x and y coordinates of the mouse or locator, which key is pressed, and which modifiers are present are returned in x, y, c, and mod respectively. In the F77 binding this is associated only with one specific keyboard event. It should be called only a single time from a key-down-event handler. In the C binding the actual keyboard event is supplied in ev.

C Binding: void PG_query_pointer(PG_device *d, int *px, int *py, int *pb, int *pq)
F77 Binding: integer pgqptr(integer d, integer x, integer y, integer b, integer q)
SX Binding:

Return the state of the mouse locator or pointer for the specified window, d. The x and y coordinates, which button is pressed, and with modifiers are present are returned in px, py, pb, and pq respectively.

The buttons are:

MOUSE_LEFT

MOUSE_MIDDLE

MOUSE_RIGHT

The modifiers are:

KEY_SHIFT

KEY_CNTL

KEY_ALT

KEY_LOCK

C Binding: PG_device *PG_get_event_device(PG_event ev)
F77 Binding:
SX Binding:
Return a pointer to the PG_device (screen window) in which the event, ev, occurred or to which it relates.

C Binding: int PG_get_char(PG_device *d)
F77 Binding:
SX Binding:
Return the ASCII code of the next character entered from the keyboard into the specified PG_device, d.

Interface Objects

As it stands we have explained the basic machinery by which PGS provides applications with the capability to get input from the terminal or from the windowing system and how various events may be handled. This is a foundation upon which graphical user interfaces may be built. PGS also provides more machinery (which is layered on top of what we have already discussed) to support the construction and editing of graphical user interfaces. The goal goes beyond portability and flexibility to that of enabling the design, construction, testing, and modification of graphical interfaces. This is done in a way that is extremely compact to implement and to use. It lets application developers pass on to their users the ability to modify user interfaces to suit their own individual needs. This can even be done at run time! It is also portable!!!

The principal notion underlying this facility is the abstraction called an interface object. It identifies and encapsulates the fundamental interactions of a user with the windowing system which underlies everything and the "atoms" of such a system. An interface object is a structure which contains the following information and methods:

Type	a string which identifies and differentiates interface objects
Region	a polygonal region in the window defining the context of the object
Visible flag	specifies whether the object is to be drawn or not
Selectable flag	specifies whether the object can be selected or not

Active flag	specifies whether the object is currently active or not
Draw method	how to draw the object if it is visible
Select method	how to select the object if it is selectable
Action method	what action to take when the object is active or activated
Parent	interface object of which this is a child
Children	array of child interface objects

Each PG_device has a tree of interface objects associated with it. Interface objects are created by calls to PG_make_interface_object. These objects individually and by virtue of their relationship in a hierarchy allow one to define the conventional graphical interface tools such as buttons, slider bars, and text boxes. The mechanism is very open ended and extensible. By defining the draw, select, and action methods and building trees of interface objects, application developers can generate virtually any kind of graphical interface functionality they wish.

The window region defines a place on the screen to which to assign certain interpretations of mouse events, most notably button press and motion events. Keeping in mind such elements of a graphical interface as "dialog boxes", it is clear that an object may or may not be drawn at all times. That is left to the interface designer to decide. So a flag and a method control the visibility of an interface object. Similarly depending on the context, an interface object may or may not be selectable. Selection is the process in which mouse events are associated with the designated window region. If an interface object is not selectable, no association is made between a mouse event and the region of the object. The action associated with an interface object can be very nebulous indeed. It can range from a function call when the object is selected to a value to be assigned to an associated variable when the object is selected. Here again, it is up to the application designer to decide.

When defining one interface object to be the child of another, the region of the child is defined in coordinates normalized with respect to the enclosing rectangle of the parent. This makes for a much more flexible and intuitive design in that deeply nested objects only refer to their parents not to the entire outer context. Changing subtrees is much easier since the children are specified relative to the parents.

In addition, interface objects have a border width, a foreground and background color, and a pointer to which anything can be assigned (e.g. a variable or some other structure). These items may be used in fairly arbitrary ways by applications. PGS uses them for the pre-defined objects as discussed in the next section.

Event Handling and Interface Objects

The event handler embedded in PG_wind_fgets treats events from the windowing system in following order fashion:

1. Checks to see whether a mouse down or key down event occurred inside any selectable interface object.

2. If inside an interface object that has an action, then that action is called.

3. Otherwise if there is a handler for the event type, it is called.

Portable User Interface Description

PGS has an ASCII representation of interface objects so that interface designers can simply write text files, called portable user interface files, that describe the interface and have PGS read and interpret them at run time, modify them interactively, and write them back out again. Thus the interfaces can be edited graphically at run time or textually with your favorite text editor. At this writing the only option which PGS supplies for graphical editing is moving objects around. Creation, deletion, duplication, and resizing are planned for future releases. However, these options can all be accomplished by editing the interface file or by application supplied editing routines.

Syntax for Interface Description

The syntax for describing an interface object is:

type [operator(parameters)]* BND(parameter) points [{ children }]

Valid operators are:

ACT	names function that does object action when active
BND	boundary specifier
CLR	specifies fore and background colors (by index)
DRW	names function that draws object when visible
FLG	state flags
NAME	object name
SEL	names function that identifies object as selected when selectable

Valid BND parameters are:

n	an integer number of NDC points (x, y) follow
RECT	2 NDC points specifying the lower left and upper right limits of a boundary rectangle follow

Valid FLG parameters are:

IsVis	object is visible
IsSel	object is selectable
IsAct	object is active

Valid CLR parameters are (using the standard color table):

```
 0      logical BLACK
 1      logical WHITE
 2      LGHT_WHITE
 3      GRAY
 4      BLUE
 5      GREEN
 6      CYAN
 7      RED
 8      MAGENTA
 9      BROWN
10      LGHT_BLUE
11      LGHT_GREEN
12      LGHT_CYAN
13      LGHT_RED
14      YELLOW
15      LGHT_MAGENTA
```

The default for FLG parameters is IsVis and IsSel. You only need use the FLG operator if you wish to change this setting. Each parameter specified turns on its corresponding flag. To turn all flags off use FLG( ) (one space between the parentheses).

A line break may occur between operator specifications. The descriptions of children of an object are delimited by curly braces.

To make this text driven representation work, it is necessary to have a mechanism to register functions and variables with PGS so that a function or variable can be looked up by name. The functions PG_register_callback and PG_register_variable do just this.

This is a very general mechanism for assigning events to actions and generically handling common activities on behalf of applications. PGS goes one step further and supplies a number of specific interface objects.

PGS Interface Objects

The specific objects which PGS supplies are: CONTAINER, BUTTON, TEXT, and VARIABLE. With these objects and some defined relationships between them it is possible to build most of the common "widgets" found in the sets supplied with various systems.

With all of the objects which PGS defines the following hold true: if an object is not visible, then its children are not visible either; if an object is not selectable then its children are not selectable either; if the border width is not zero, a black border is drawn around the object; and if the background color of an object is -1 the closest ancestor with a background color other than -1 supplies the background color.

PGS interface objects have two kinds of containment associations: visual containment in which child objects appear visually (on the screen) within the region defined by their parents; and logical containment in which only the position in the hierarchy defines the containment association relationship, that is children are logically contained within their parents.

The CONTAINER object is defined simply to specify logical containment of objects. When looking up a hierarchy it is a natural breakpoint to identify groups of objects which may have to collectively respond to events (we'll see examples of this later). It also can be used just to ensure that a border is drawn around a group of other objects (these would have to be children of the container).

The BUTTON object is defined as a trigger. When a mouse down event happens in a button object a chain of events is set off. This covers initiation of call back functions, setting the value of a VARIABLE object, or moving a slider around. It depends on what the button means (as defined by context in a hierarchy or by its action method).

A TEXT object is an encapsulation of a PG_text_box structure and the methods which accompany it. The text box is a collection of text and editing functions. The editing functions supply an EMACS-like text manipulation facility for any quantity of text from one line to as much text as can be loaded into memory. Text boxes with a single line of text have the text centered in the box and multi-line text boxes have their text left justified.

A VARIABLE object binds a variable to an interface object. The parent, siblings, and children of the VARIABLE object define its actual behavior.

PGS predefines the following methods using PG_register_callback:

DRW Methods

```
draw-text        draw a text object (default for TEXT objects)

draw-variable    draw a variable object (default for VARIABLE objects)

draw-container   draw a container object (default for CONTAINER objects)

draw-button      draw a button object (default for BUTTON objects)

draw-slider      draw a slider button
```

SEL Methods

```
select-visual    select based on visual containment

select-logical   select based on logical containment
```

ACT Methods

```
slider           action function for slider buttons

toggle           action function to toggle visibility and selectability of c
```

With these building blocks in hand we will look at some common widgets found in many graphical user interface kits. It is very important to note that much of the behavior of these objects derives from their relative positions in the hierarchy of interface objects. This position relationship contributes as much as the atomic properties of the individual objects to the functioning of these combinations. This is a key feature in the flexibility of the PGS design.

Buttons

The intended functionality is a "button" on the screen which when selected by a mouse click invokes a function. For example, consider an "End" button to terminate an applications:

```
Button CLR(3,2) DRW(draw-button) ACT(End) BND(RECT) (0.1,0.9) (0.2,0.95)
   {Text NAME(End) CLR(10,-1) FLG(IsVis) DRW(draw-text) BND(RECT) (0.05,0.3
```

The action method of the button is the function associated with the string "End" in the callback table. The text "End" appears in the button and is visible but not selectable.

Enumerable Variables

Certain variables have a small number of possible values and it is convenient and pleasing to present buttons for each possible value and have the user select the desired one. This situation might look like this:

```
Container CLR(0,0) BND(RECT) (0.1,0.82) (0.25,0.87)
  {Variable NAME(Output) CLR(10,0) BND(RECT) (0.0,0.49) (1.0,1.0)
     {Button CLR(3,2) ACT(1) BND(RECT) (0.0,-1.0) (0.5,0.0)
        {Text NAME(On) CLR(10,-1) FLG(IsVis) BND(RECT) (0.05,0.05) (0.95,0.
      Button CLR(3,2) ACT(0) BND(RECT) (0.5,-1.0) (1.0,0.0)
        {Text NAME(Off) CLR(10,-1) FLG(IsVis) BND(RECT) (0.05,0.05) (0.95,0
```

The children of the variable registered using PG_register_variable under the name "Output" are buttons which when selected set the value of the variable to the value taken from the action of the button (1 for the "On" button and 0 for the "Off" button). A second more elaborate example shows this same principle:

```
Container CLR(0,0) BND(RECT) (0.85,0.88) (1.0,1.0)
  {Variable NAME(Direction) CLR(10,0) BND(RECT) (0.2,0.0) (0.8,0.2)
     {Button CLR(3,2) ACT("n")
       BND(5) (0.51,3.12) (0.64,3.53) (0.51,3.88) (0.36,3.53) (0.51,3.12)
         {Text NAME(N) CLR(10,0) FLG(IsVis) BND(RECT) (0.0,1.0) (1.0,2.0)}
      Button CLR(3,2) ACT("e")
      BND(5) (0.55,3.0) (0.7,2.59) (0.9,3.0) (0.7,3.41) (0.55,3.0)
        {Text NAME(E) CLR(10,0) FLG(IsVis) BND(RECT) (1.0,0.0) (2.0,1.0)}
      Button CLR(3,2) ACT("s")
      BND(5) (0.51,2.88) (0.36,2.53) (0.51,2.12) (0.64,2.53) (0.51,2.88)
        {Text NAME(S) CLR(10,0) FLG(IsVis) BND(RECT) (0.0,-1.0) (1.0,0.0)}
      Button CLR(3,2) ACT("w")
      BND(5) (0.45,3.0) (0.3,3.41) (0.1,3.0) (0.3,2.59) (0.45,3.0)
        {Text NAME(W) CLR(10,0) FLG(IsVis) BND(RECT) (-1.0,0.0) (0.0,1.0)}}
```

Here a variable registered under the name "Direction" has four buttons (which are not rectangles), "N", "E", "S", and "W" which take values that are strings and the text of the buttons is outside of the buttons at the points.

Nonenumerable Variables

Sometimes variables do not take a small set of discrete values. In such a case a reasonably natural way for a graphical interface to set the variable is to enter the text representation of the value and have it converted to the appropriate binary form. PGS currently only supports ints, longs, floats, doubles, and strings in this way. Here is an example of this:

```
Container CLR(0,0) BND(RECT) (0.26,0.82) (0.39,0.87)
  {Variable NAME(Theta) CLR(10,0) BND(RECT) (0.0,0.49) (1.0,1.0)
     {Text NAME(30) CLR(10,-1) BND(RECT) (0.05,-0.95) (0.95,-0.05)}}
```

The text "30" might be the initial value for the variable registered using PG_register_variable under the name "Theta". PGS uses the actual current value of the variable in the code for all display purposes. To change the value, you place the cursor in the region of the text, edit it to show the correct value, and hit a carriage return. The value of the variable is then set to that show in the text box. Note that the VARIABLE object must be selectable in order for the text box which actually controls the value to be selectable.

Sliders

Another way to set numeric variables is to use a button which can move within some limits and whose position within those bounds determines the value of the variable. This is what we call a slider. PGS supports both one dimensional sliders in which there is only one degree of freedom and controls one variable and two dimensional sliders in which there are two degrees of freedom and two variables are controlled simultaneously.

An example of the one dimensional slider is:

```
Container CLR(0,0) BND(RECT) (0.41,0.82) (0.54,0.88)
  {Variable NAME(Phi) CLR(10,0) BND(RECT) (0.0,0.67) (1.0,1.0)
     {Text NAME(-60) CLR(10,-1) BND(RECT) (0.05,-0.93) (0.95,-0.067)}
   Container CLR(2,2) BND(RECT) (0.0,0.0) (1.0,0.3)
     {Button CLR(0,0) DRW(draw-slider) ACT(slider) BND(RECT) (0.28,0.0) (0.
```

What makes this a one dimensional slider is that the BUTTON with the slider action takes up the entire span in the y direction of the parent CONTAINER. The text is not strictly necessary here. It is nice to see the value, however, and you can also change the value by entering it in the text box. Either mode of setting the values causes both indicators to show the same value.

An example of the two dimensional slider is:

```
Container CLR(0,0) BND(RECT) (0.65,0.82) (0.8,0.97)
  {Variable NAME(Theta) CLR(10,0) BND(RECT) (0.05,0.88) (0.45,0.99)
     {Text NAME(45) CLR(10,-1) BND(RECT) (0.05,-0.95) (0.95,-0.05)}
```

```
   Variable NAME(Phi) CLR(10,0) BND(RECT) (0.55,0.88) (0.95,0.99)
     {Text NAME(0) CLR(10,-1) BND(RECT) (0.05,-0.95) (0.95,-0.05)}
   Container CLR(2,2) BND(RECT) (0.0,0.0) (1.0,0.72)
     {Button CLR(0,0) DRW(draw-slider) ACT(slider) BND(RECT) (0.0,0.0) (0.1
```

This is a two dimensional slider because the BUTTON with the slider action does not span either the entire x or y direction of the CONTAINER parent. The first VARIABLE, "Theta", associates with the x direction and the second VARIABLE, "Phi", with the y direction. The text boxes work the same way as for the one dimensional slider.

Transients

Certain interface objects may be visible only transiently in routine use. Pull down menus and dialog boxes are examples of such constructs. Here is an example of an object whose visibility and selectability are toggled when a button is selected.

```
Container NAME(Menu) CLR(0,0) FLG( ) BND(RECT) (0.349,0.021) (0.551,0.121)
  {Text NAME(A) CLR(10,-1) FLG(IsVis) BND(RECT) (0.052,0.014) (0.948,0.333)
   Text NAME(B) CLR(10,-1) FLG(IsVis) BND(RECT) (0.052,0.333) (0.948,0.667)
   Text NAME(C) CLR(10,-1) FLG(IsVis) BND(RECT) (0.052,0.667) (0.948,0.986)

Button CLR(3,2) DRW(draw-button) ACT(toggle,Menu) BND(RECT) (0.101,0.05) (0
  {Text NAME(Menu) CLR(10,-1) FLG(IsVis) DRW(draw-text)
   BND(RECT) (0.053,0.286) (0.947,0.714)}
```

The container "Menu" forms the subtree which is to be summoned and dismissed by actuating the button. Notice how the connection works. The action specifies both the toggle method and an interface object whose visibility and selectability are to be toggled. Compare this with the next example below.

```
Container NAME(Rendering) CLR(0,0) FLG( ) BND(RECT) (0.399,0.051) (0.601,0.
  {Text NAME(Rend) CLR(10,-1) FLG(IsVis) BND(RECT) (0.052,0.014) (0.948,0.3

Container NAME(Axis) CLR(0,0) FLG( ) BND(RECT) (0.449,0.101) (0.651,0.201)
  {Text NAME(Ax) CLR(10,-1) FLG(IsVis) BND(RECT) (0.052,0.014) (0.948,0.333

Container CLR(0,0) BND(RECT) (0.101,0.100) (0.200,0.18)
  {Variable NAME(Which-Panel) CLR(0,0) ACT(toggle) BND(RECT) (0.0,0.0) (1.0
     {Button CLR(3,2) DRW(draw-button) ACT(Rendering) BND(RECT) (0.0,0.5) (
        {Text NAME(Render) CLR(10,-1) FLG(IsVis) DRW(draw-text)
         BND(RECT) (0.05,0.3) (0.95,0.95)}
      Button CLR(3,2) DRW(draw-button) ACT(Axis) BND(RECT) (0.0,0.0) (1.0,0
        {Text NAME(Axis) CLR(10,-1) FLG(IsVis) DRW(draw-text)
         BND(RECT) (0.05,0.3) (0.95,0.95)}}}
```

In this case a variable is defined with the toggle method. The value of the variable is a string which is the name of the container to be toggled. That is, if the "Render" button is pressed, the "Rendering" container becomes visible and selectable. If the "Axis" button is now pressed, the "Rendering" container becomes invisible and unselectable while the "Axis" container becomes visible and selectable. A key feature here is that the variable "Which-Panel" does not and should not be registered by the application. PGS implicitly defines and registers undefined variables like this as strings and uses them as described above.

Rendering Model

This section describes the model used in PGS to do various high level renderings of data for the purposes of scientific visualization. High level renderings refer to the notion of carrying out a large number of graphical operations to generate a "plot" or "rendering" of a set of data. To make this as easy as possible, PGS supplies a set of routines which will give "one picture for one call". These routines have the flexibility to produce plots which fit the needs of the user who can set rendering attributes to control the output in detail. The attributes all have a reasonable default value so that it is possible to make reasonable plots with a single call.

Visualization does not exist in a vacuum. It is strongly coupled to analysis and to data storage. PACT, of which PGS is a part, has extensive facilities in all of these areas. To communicate among them a common representation of data sets is used and instances of these data sets are passed around. Data sets are organized into two mathematically motivated structures: PM_set and PM_mapping. These are described in more detail in the PML Users Manual. A third data structure called a PG_graph is used to contain and associate rendering information with the PM_mapping which only contains the data that a simulation or observation would yield or that an analysis program would need. A picture of how these parts fit together is:

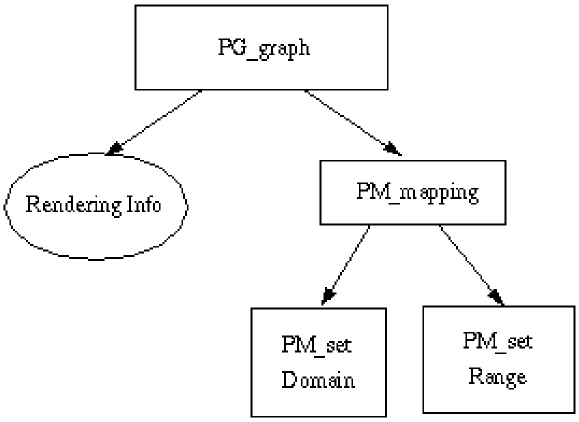


FIGURE 1. Relationship between PG_graphs, PM_mappings, and PM_sets

Data Structures

The data structures employed in PGS for the purposes of visualization come from PGS itself and PML, the math library for PACT. The interested user should consult the PML Users Manual for more complete information as well as descriptions of routines which manipulate these structures.

PM_set

The PM_set or set represents a collection of objects (primarily but not exclusively numbers). It also describes the dimensionality of the set, the dimensionality of the elements, and the connectivity of the elements.

PM_mapping

The PM_mapping or mapping represents the relationship between two sets of elements. In mathematics this is the generalization of a function. A mapping consists of two sets, a domain and range set, and a rule for associating elements of the two sets. Generally in numerical applications the rule is based on the order of the elements of the two sets with a specification of centering and strides through the elements.

PM_mesh_topology

The PM_mesh_topology defines the connectivity of elements in a set if the trivial logical array ordering is not to be used. The logical array ordering is referred to as logical rectangular and is used in many simulations. However, many applications today cannot represent their data in this way and the means for a completely general specfication of neighbor relations is provided by the PM_mesh_topology structure. Such relationships are referred to as arbitrarily connected.

PG_graph

To visualize data requires two ingredients: data; and rendering specifications. The mapping described above takes care of the first part and a list of rendering attributes does the rest. The PG_graph contains these part in one convenient package.

PG_image

A simple but extremely useful data representation is a raster image or cell array. PGS provides a structure to contain such information and visualize it.

PG_palette

The specification of color maps is handled with the PG_palette structure. This information tells PGS how to match 3 dimensional RGB color space with n dimensional data sets. The most common situation is that of a single dimension data in a range set or image, but PGS supports multidimensional palettes for higher dimensional ranges.

Rendering Modes

PGS currently features several rendering modes. They are listed here along with a brief description.

Line Plots

Line plots is a generic term referring to graphs with 1 dimensional domains and ranges. When a graph has 1d domain and a 1d range PG_draw_graph will render the data in any of the following ways depending on the value of the PLOT-TYPE attribute.

CARTESIAN	Canonical x vs y plot.
POLAR	Polar plot with r vs theta.
INSEL	Inselberg plot with parallel axes and (x, y) points represented as lines connecting the values on the axes.
HISTOGRAM	Cartesian histogram plot with the steps starting with the LEFT value, the RIGHT value, or the averaged or CENTER value.
SCATTER	Scatter plot where points are a plotted with a marker character but not connected by line segments (Cartesian)
LOGICAL	Plot y values versus their array index. The x values are ignored.
ERROR_BAR	Like a scatter plot but instead of marker characters being used error bars are drawn. Requires 2 arrays for x error and y error or 4 arrays for positive and negative going x error and positive or negative going y error.

Two Dimensional Plots

Two dimensional plot is a generic term referring to graphs with 2 dimensional domains. The most common situation features a 1 dimensional range but higher dimensions may be used. When a graph has a 2d domain PG_draw_graph will render the data in any of the following ways depending on the value of the PLOT-TYPE attribute.

PLOT_CONTOUR	A traditional iso contour plot. Can be done with either logical or arbitrary connectivity.
PLOT_IMAGE	A rasterized image plot. Can only be done with logical connectivity.
PLOT_WIRE_MESH	The domain values are x and y and the range values are z in a 3 dimensional space. Line segments connect neighboring points. The data can be examined from any specified view angle. The algorithm is a z buffered scan line technique. Can be done with either logical or arbitrary connectivity.
PLOT_SURFACE	The domain values are x and y and the range values are z in a 3 dimensional space. In addition to showing the connecting line segments, the faces bounded by the segments are shaded. Two dimensional ranges are handled by taking the first component to be the z value and the second component as the color value. The data can be examined from any specified view angle. The algorithm is a z buffered scan line technique. Can be done with either logical or arbitrary connectivity.
PLOT_FILL_POLY	The facets bounded by segments connecting neighbors are filled with a single color determined by the range value. Can be done with either logical or arbitrary connectivity.

Vector Plots

Vector plot refers to a plot in which the range has at least two dimensions. Currently PGS can only render 2 dimensional vectors but in the long term will render higher dimensional vectors as well. Also currently vector plots may only be done with 2 dimensional domains. They are requesed by setting the PLOT-TYPE attribute of the graph to PLOT_VECTOR.

Mesh Plots

Mesh plots are graphical renderings of the connectivity of a domain set. In PGS lines are drawn connecting neighboring points. They can be either 2 dimensional or 3 dimensional. If 3 dimensional they may be viewed from any angle similarly to the PLOT_WIRE_MESH plots discussed above. If a NULL range is specified only the mesh is drawn. If a range is supplied the range values are printed at their corresponding nodes. They are requesed by setting the PLOT-TYPE attribute of the graph to PLOT_MESH.

Rendering Attributes

By default PGS assumes a "look and feel" for the various renderings that it can do. This look can be reduced to a list of characteristic or attribute values. Realizing that applications need to control their own look and feel, they are given a mechanism to change these attributes.

Generally speaking, attributes are managed as association lists, that is lists of key-value pairs. Attribute list are associated with graphs (PG_graph) and sets (PM_set). That is to say that some rendering attributes belong naturally with a set (for example, the plotting limits) and others with a graph (for example, the number of contour levels). The PGS rendering routines query the attribute lists for specific attribute values and use their defaults if they are not found.

Modifying attribute lists is done with the following functions (summarized from the SCORE User’s Manual):

`void *SC_assoc(pcons *alist, char *s)`

```
Return the attribute value if present
```

`pcons *SC_add_alist(pcons *alist, char *name, char *type, void *val`

```
Add an attribute value to a list
```

`pcons *SC_rem_alist(pcons *alist, char *name)`

```
Remove an attribute value from a list
```

`pcons *SC_change_alist(pcons *alist, char *name, char *type, void *`

```
Change an attribute value and add it if it is not there
```

void SC_free_alist(pcons *alist, int level)

```
Release an association list. Use level 2.
```

Attributes

This is the list of attributes currently understood by PGS. In use these all appear as quoted strings.

AXIS-TYPE
: Flag specifying an axis type of CARTESIAN, POLAR, or INSEL (int). The default is CARTESIAN.

CHI
: Value specifying the chi component of the three Euler angles (double). The default is 0.0 degrees.

CORNER
: Value indicating the node of a logical rectangle which is associated with the zone center (int). 1 associates lower right, 2 upper right, 3 upper left, and 4 lower left. This applies to logical rectangular mappings only. The default is 2.

DRAW-AXIS
: If TRUE a high level rendering routine will draw a set of axes.

DRAW-LABEL
: If TRUE a high level rendering routine will draw the graph label.

DRAW-LEGEND
: If TRUE the contour plotting routine will draw the legend of contour values.

DX-MINUS
: For PLOT_ERROR_BAR type renderings this is the array of negative going errors on the x values (double *). This must have the same number of entries as the x values.

DX-PLUS
: For PLOT_ERROR_BAR type renderings this is the array of positive going errors on the x values (double *). This must have the same number of entries as the x values.

DY-MINUS
: For PLOT_ERROR_BAR type renderings this is the array of negative going errors on the y values (double *). This must have the same number of entries as the y values.

DY-PLUS

For PLOT_ERROR_BAR type renderings this is the array of positive going errors on the y values (double *). This must have the same number of entries as the y values.

EXISTENCE

An existence map array specifiying missing zones in a logical rectangular mesh (char *). This prevents plotting of information which would be associated with non-existent zones in a domain mesh.

HIST-START

Flag specifying whether a PLOT_HISTOGRAM rendering starts with the value on the LEFT, CENTER, or RIGHT side of the bar (int).

LEVELS

The array of contour levels which must be N-LEVELS long (double *). This is only used when an application wants more control over contour levels than the contour plotting routines in PGS already provide.

LIMITS

Array of minimum and maximum values (one pair per dimension) specifying the plotting limits of a domain or range set (double *). The length must be twice the number of dimensions of the set.

LINE-COLOR

The line color index (int). The default is BLUE.

LINE-STYLE

The line style index SOLID, DOTTED, DASHED, DOTDASHED (int). The default is SOLID.

LINE-WIDTH

The line width (double). The default is 0.0.

MARKER-INDEX

The index into the marker array (int). This depends on how many markers have been defined with PG_def_marker. The default is 0.

MARKER-SCALE

The scale factor applied to a marker when drawn (double). The default is 0.01.

N-LEVELS

The number of isocontour levels used in a contour plot (int). The default is 10.

NORMAL-DIRECTION

A flag specifying the normal direction of surface elements in hidden surface plot (int). The default is 1.

PALETTE
: The name of the palette to use in those plots which need color palettes (char *). The default is the current palette of the device.

PHI
: Value specifying the phi component of the Euler angles defining the viewing angle (double). The default is 0.0 degrees.

PHI-LIGHT
: Value specifiying the phi angle of a light source illuminating a surface plot (double). The default is 45.0 degrees.

PLOT-TYPE
: Flag specifying the plot type CARTESIAN, POLAR, or INSEL (int). The default is CARTESIAN.

RATIO
: Isocontour spacing ratio (double). The default is 1.0.

SCATTER
: Flag specifying that a rendering be done as a scatter plot if TRUE (int). The default is FALSE.

THETA
: Value specifying the theta component of the Euler angles defining the viewing angle (double). The default is 0.0 degrees.

THETA-LIGHT
: Value specifiying the theta angle of a light source illuminating a surface plot (double). The default is 45.0 degrees.

VIEW-PORT
: An array of values in normalized coordinates (xmin, xmax, ymin, and ymax) specifying a viewport to be for a plot (double *). The default is defined by each rendering routine to maximize the area of the plot.

Plots and associated attributes

This section tells which rendering attributes are meaningful to which renderings.

Contour Plot Attributes

DRAW-AXIS

DRAW-LABEL

DRAW-LEGEND

LEVELS

LIMITS

LINE-COLOR

LINE-WIDTH

LINE-STYLE

N-LEVELS

LEVELS

RATIO

VIEW-PORT

Domain Plot Attributes

CHI

CORNER

DRAW-AXIS

DRAW-LABEL

EXISTENCE

LIMITS

LINE-COLOR

LINE-STYLE

LINE-WIDTH

MARKER-INDEX

MARKER-SCALE

PALETTE

PHI

PHI-LIGHT

PLOT-TYPE

THETA

THETA-LIGHT

SCATTER

Line Plot Attributes

AXIS-TYPE

DX-MINUS

DX-PLUS

DY-MINUS

DY-PLUS

HIST-START

LINE-COLOR

LINE-STYLE

LINE-WIDTH

MARKER-INDEX

PLOT-TYPE

SCATTER

Hidden Surface Attributes

NORMAL-DIRECTION

Image Plot Attributes

CORNER

DRAW-AXIS

DRAW-LABEL

EXISTENCE

LIMITS

Fill Poly Plot Attributes

CORNER

DRAW-AXIS

DRAW-LABEL

EXISTENCE

LIMITS

VIEW-PORT

Surface Plot Attributes

CHI

CORNER

DRAW-AXIS

DRAW-LABEL

EXISTENCE

LIMITS

LINE-COLOR

LINE-STYLE

LINE-WIDTH

PHI

THETA

Vector Plot Attributes

CORNER

DRAW-AXIS

DRAW-LABEL

EXISTENCE

LIMITS

LINE-COLOR

LINE-STYLE

LINE-WIDTH

VIEW-PORT

The PGS API

The application program interface (API) for PGS is presented in this section. There are three language bindings for most functions in PGS: C; Fortran; and SX. SX is a part of PACT as is PGS. It is an extended dialect of the Scheme programming language. What you get using SX is like what you get using C and loading with the PGS library. Keep in mind however that SX is an interpreted language and lends itself to certain applications which are not suitable for C or Fortran. For more on SX see the SX User's Manual.

Each language has its own particular features and consequently there are differences in how the PGS functions are used. We have tried to keep consistency between the bindings in order to help users who are familiar with one or more of the languages involved to be able to use any of them. Some discussion of the language differences is given below and the reader is STRONGLY urged to READ this material before proceeding.

The following short hand makes for easier explanations:

ON and TRUE mean 1 in all languages

OFF and FALSE mean 0 in all languages

C

C is the language in which PGS is implemented. This means that the C bindings traffic in the data structures and pointers of the implementation. The other languages require various devices to obtain a functional equivalent to the C functionality. In this sense the C bindings are fundamental and the reader should keep this in mind at times when attempting to understand some of the more abstruse PGS calls.

A note to C users: PGS is portable to a wide variety of platforms, operating systems, and compilers. Older C compilers do not support the void type well. To smooth over that difference PACT supplies a #define'd constant called byte. On an ANSI system byte is defined to be void, but on an older system, byte is defined to be char. This works extremely well and provides a relatively small bit of confusion.

Fortran

For Fortran functions the type designator REAL indicates arguments which must be floating point numbers. Whether the actual type declaration in the calling FORTRAN program should be real or double precision is platform dependent. In all cases PGS expects a 64 bit quantity.

Since there is no accepted standard for how C and Fortran communicate, it is necessary for PGS (and all of PACT) to observe one rule regarding string arguments: two variables are passed. The first is the number of meaningful characters in the string and the second is the string itself.

All functions in the FORTRAN API return TRUE (1) if PGS detects no error (some host systems are better than others about reporting error conditions) and FALSE (0) otherwise unless otherwise noted.

SX

In SX as in all LISP dialects two features must be noted and understood. First, values have types not variables. This means that the description of the SX bindings don't show types associated with the formal parameters. The types of the arguments must match those in the corresponding C calls with the exception of numeric values (they are coerced to the needed type). Second, there are no pointers in the C sense and the language uses pass by value procedure calls. What this means is that nothing is returned to the caller via the argument list. Instead a list of the return values is made and returned. The number and order of the values in the list matches those that are returned via the argument list in the C and Fortran calls.

Compiling and Loading

To compile your C programs you must use the following

#include <pgs.h>

in the source files which deal with PGS graphics.

FORTRAN programs have no special requirements of the sources. It is however important to remember that the PGS FORTRAN routines all begin with "p" and would be implicitly typed as real when in fact they all return integers. You should take care to declare the routines which you use.

To link your application you must use the following libraries in the order specified.

-lpgs [-lX11] -lppc -lpdb -lpml -lscore [-lm ...]

Although this is expressed as if for a UNIX linker, the order would be the same for any system with a single pass linker. The items in [] are optional or system dependent.

Each system has different naming conventions for its libraries and the reader is assumed to understand the appropriate naming conventions as well as knowing how to tell the linker to find the installed PACT libraries on each system that they use.

PGS Functions

PGS has a wide variety of functionality. The functionality is broken down into related groups which are listed together in sections and alphabetically in each section.

Global State Setting Routines

These routines set state that is global in scope as opposed to device or graph level control.

C Binding: int PG_def_marker(int n_seg, REAL *x1, REAL *y1, REAL *x2, REAL *y2)
F77 Binding: integer pgdmrk(integer n_seg, real x1, real y1, real x2, real y2)
SX Binding: (pg-define-marker x1 y1 x2 y2 ...)

This routine defines a new marker in terms of a set of line segments. The arguments are the number of segments, n_seg, and arrays specifying the x and y values of the endpoints of each segment. Each array must be n_seg elements long. The values in the arrays must be between -1.0 and 1.0. The marker can be scaled to any size and rotated by using the macros PG_set_marker_scale and PG_set_marker_orientation. The index of the new marker is returned and should be used as values for the MARKER-INDEX attribute where called for.

C Binding: void PG_set_clear_mode(int mode)
F77 Binding: integer pgsclm(integer mode)
SX Binding:

Set a global mode which the high level rendering routines use to interpret what it means to clear the current picture. There are three interpretations which PGS supports: 1) is to clear the entire PGS window (CLEAR_SCREEN); 2) is to clear only the viewport (CLEAR_VIEWPORT) this leaves axes and labels which have already been drawn intact; and 3) clear only the current frame (CLEAR_FRAME) which leaves still more elements of a picture untouched. Any other value results in no action being taken by the high level renderers to clear anything.

C Binding:
F77 Binding:
SX Binding: (pg-set-view-angle! theta phi chi)

Set a global default viewing angle for 3D plots. From the observer’s point of view: phi is a clockwise rotation about the positive z axis; theta is a clockwise rotation about the positive x axis; and chi is a counter-clockwise rotation about the line of sight which is the same as the z axis after the theta and phi rotations have been applied. The theta rotation is done so that a view looking down the z axis (x, y) is turned into a view looking down the y axis (x, z) in the most economical manner - that is with theta equal to 90 degrees.

Global State Query Routines

C Binding: void PG_get_clear_mode(int mode)
F77 Binding: integer pggclm(integer mode)
SX Binding:

Get the current value of the global mode which the high level rendering routines use to interpret what it means to clear the current picture. See PG_set_clear_mode for a fuller discussion.

Memory Management Routines

These routines allocate and initialize or release instances of PGS data structures.

C Binding: PG_device *PG_make_device(char *name, char *type, char *title)
F77 Binding: integer pgmkdv(integer ncn, char *name, integer nct, char *type, integer ncl, char *title)
SX Binding: (pg-make-device name type title)

Allocate and initialize a new PG_device structure. Name specifies the kind of device wanted (WINDOW, PS, CGM, RASTER). Type specifies whether the device is COLOR or MONOCHROME. Title is either the text of a title bar or the name of an output file as in the case of PS or CGM devices. In the case of PS or CGM devices the title is used as the base of the file name and ".ps" or ".cgm" is added as the suffix appropriately. In addition, with PS devices the EPS conformance level can be specified as follows:

    base_name [PS-level [EPS-level]]

where PS-level and EPS-level specify the level of conformance. PGS writes very highly conforming files but some applications which would import them cannot recognize standards which are higher or lower than the ones for which they are programmed. This method lets PGS based applications target their applications. Values of 2.0 or 3.0 are most common.

C Binding: PG_graph *PG_make_graph_from_mapping(PM_mapping *f, char *info_type, void *info, int id, PG_graph *next)
F77 Binding:
SX Binding:

Setup and return a PG_graph using a PM_mapping, f, and rendering information in the alist info. The id is a character which will be used as a starting data-id on a contour plot or as the data-id of a line plot. To chain graphs together so that they may be plotted together next is used to point to the next graph in a chain.

C Binding: PG_graph *PG_make_graph_from_sets(char *label, PM_set *domain, PM_set *range, int centering, char *info_type, void *info, int id, PG_graph *next)
F77 Binding: integer pgmgfs(integer nl, char *label, integer domid, integer ranid, integer centering, integer id, integer next)
SX Binding: (pg-make-graph domain range [centering color width style emap name])

Setup a new instance of a PG_graph and return it. The arguments are: the domain and range sets of the mapping part of the graph; the relative of the centering of the range and domain data; and rendering information in the form of an alist, info, or line color, width and style and an existence map, emap, for the mesh. The label is a string used to label the entire plot and may be plotted in some circumstances. The id is a character which will be used as a starting data-id on a contour plot or as the data-id of a line plot. To chain graphs together so that they may be plotted together next is used to point to the next graph in a chain.

C Binding: PG_graph *PG_make_graph_r2_r1(int id, char *label, int cp, int kmax, int lmax, int centering, REAL *x, REAL *y, REAL *r, char *dname, char *rname)
F77 Binding: integer pgmg21(integer id, integer nl, char *label, integer cp, integer kmax, integer lmax, integer centering, real x, real y, real r, integer nd, char *dname, integer nr, char *rname)
SX Binding:

Setup and return a specific kind of graph containing a 2d rectangular domain from arrays x and y and a matching 1d range from array r. The size of the arrays is kmax by lmax. If cp is TRUE the x, y, and r arrays will be copied for the domain and range sets. It is sometimes necessary for the sets to have dynamically allocated spaces or to have spaces which they can safely free when they are released. The id is a character which will be used as a starting data-id on a contour plot. The dname and rname are strings used a labels for the domain and range sets respectively. They are never printed on a plot but would be written out to a data file. The label is a string used to label the entire plot and may be plotted in some circumstances.

C Binding: PG_graph *PG_make_graph_1d(int id, char *label, int cp, int n, REAL *x, REAL *y, char *xname, char *yname)
F77 Binding: integer pgmg11(integer id, integer nl, char *label, integer cp, integer n, real x, real y, integer nx, char *xname, integer ny, char *yname)
SX Binding:

Setup and return a specific kind of graph containing a 1d domain from array x and a matching 1d range from array y. The size of the arrays is n. If cp is TRUE the x, y, and r arrays will be copied for the domain and range sets. The reason for this is that it is sometimes necessary for the sets to have dynamically allocated spaces or to have spaces which they can safely free when they are released. The id is a character which will be used as a data-id on a plot. The xname and yname are strings used a labels for the domain and range sets respectively. They are never printed on a plot but would be written out to a data file. The label is a string used to label the entire plot and may be plotted in some circumstances.

C Binding: void PG_rl_graph(PG_graph *g, int rld, int rlr)
F77 Binding: integer pgrlgr(integer g, integer rld, integer rlr)
SX Binding: automatically garbage collected

This function releases an instance of a PG_graph. If rld is TRUE the data arrays in the domain set will be freed and if rlr is TRUE the data arrays in the range set will be freed.

C Binding: PG_image *PG_make_image(char *label, char *type, double xmn, double xmx, double ymn, double ymx, double zmn, double zmx, int k, int l, int bits_pix, PG_palette *palette)
F77 Binding:
SX Binding: (pg-build-image dev data k l [name xmn xmx ymn ymx zmn zmx])

The arguments to this function are:

label	a label string for the image
type	the data type used in the image
(xmn, xmx)	the minimum and maximum extent in the x direction
(ymn, ymx)	the minimum and maximum extent in the y direction
(zmn, zmx)	the minimum and maximum extent in the image data
(k, l)	the number of pixels in the x and y direction respectively
bits_pix	the number of image bits per pixel (1 for MONOCHROME and typically 8 for COLOR)
palette	the palette to be used in rendering the image

C Binding: void PG_rl_image(PG_image *im)
F77 Binding:
SX Binding: automatically garbage collected

These two functions create and release PG_image instances. The images are k by l pixels. They have world coordinate extents from xmin to xmax and ymin to ymax. The data ranges from zmin to zmax and is of type type. A palette may be supplied along with a label for a plot.

Device Control Routines

These routines provide for high level control of PGS devices. They also permit applications to set various aspects of the state of devices.

C Binding: void PG_clear_page(PG_device *dev, int i)
F77 Binding: integer pgclpg(integer devid, int i)
SX Binding:

Clear the page for a text window such as the console. Leave the current line at line i in the PGS window.

C Binding: void PG_clear_region_NDC(PG_device *dev, double xmn, double xmx, double ymn, double ymx, int pad)
F77 Binding: integer pgclrg(integer devid, real xmn, real xmx, real ymn, real ymx, integer pad)
SX Binding: (pg-clear-region dev xmn xmx ymn ymx pad)

Clear the rectangular region, specified in normalized coordinates, of the given device. The limits of the rectangle are xmn, xmx, ymn, and ymx. The pad is a number of pixels to inset the cleared region. This facilitates clearing a region without removing a border line around the region.

C Binding: void PG_clear_window(PG_device *dev)
F77 Binding: integer pgclsc(integer devid)
SX Binding: (pg-clear-window dev)

Clear the entire PGS window on the specified device.

C Binding: void PG_clear_viewport(PG_device *dev)
F77 Binding: integer pgclvp(integer devid)
SX Binding: (pg-clear-viewport dev)

Clear the current viewport region only on the specified device.

C Binding: void PG_close_console(void)
F77 Binding:
SX Binding:

Close the console device.

C Binding: void PG_close_device(PG_device *dev)
F77 Binding: integer pgclos(integer dev)
SX Binding: (pg-close-device dev)

Close the specified device, dev.

C Binding: void PG_finish_plot(PG_device *dev)
F77 Binding: integer pgfnpl(integer devid)
SX Binding: (pg-finish-plot dev)

Finish the picture on the specified device. Once a picture is finished, nothing more can be drawn to the device until a call to PG_clear_window is done without serious consequences. This is especially necessary for devices such as PS and CGM devices.

C Binding: void PG_get_axis_log_scale(PG_device *dev, int *xls, int *yls)
F77 Binding: integer pggaxl(integer devid, integer xls, integer yls)

SX Binding:

Get the x-axis or y-axis log scale flags in the specified device. The argument xls contains the value of the x-axis log flag and yls contains the value of the y-axis log flag on return,

C Binding: void PG_make_device_current(PG_device *dev)
F77 Binding: integer pgmdvc(integer devid)
SX Binding: (pg-make-device-current dev)

Make the specified device the current device for drawing.

C Binding: void PG_open_console(char *title, char *type, int bckgr, double xf, double yf, double dxf, double dyf)
F77 Binding:
SX Binding:

Open a console device at the specified point (xf, yf) with the specified width, dxf, and height, dyf (these are all in normalized coordinates). The console window will have title in the title bar, type, type, and the indicated background color. Type, the window type is one of "COLOR" or "MONOCHROME". Bckgr should be TRUE for white background and FALSE for black background.

C Binding: PG_device *PG_open_device(PG_device *dev, double xf, double yf, double dxf, double dyf)
F77 Binding: integer pgopen(integer devid, REAL xf, REAL yf, REAL dxf, REAL dyf)
SX Binding: (pg-open-device dev xf yf dxf dyf)

Open the specified device at the specified point (xf, yf) with the specified width, dxf, and height, dyf. These values are all normalized to the physical device dimensions. NOTE: to make it easy to create a square window, the actual pixel height of the PGS window is computed as dyf*display_pixel_width!

C Binding: void PG_release_current_device(PG_device *dev)
F77 Binding: integer pgrdvc(integer devid)
SX Binding: (pg-release-current-device dev)

Release the specified device as the current drawing device. (A few host graphics systems need this functionality).

C Binding: void PG_set_attributes(PG_device *dev, PG_dev_attributes *attr)
F77 Binding:
SX Binding:

Set the collection of attributes from the PG_dev_attributes structure attr in the specified device.

C Binding: void PG_set_axis_log_scale(PG_device *dev, int xls, int yls)
F77 Binding: integer pgsaxl(integer devid, integer xls, integer yls)
SX Binding:

Set the x-axis or y-axis log scale flags in the specified device. The argument xls causes the x-axis to be plotted with a log scale if TRUE and yls causes the y-axis to be plotted with a log scale if TRUE.

C Binding: void PG_set_border_width(PG_device *dev, int t)
F77 Binding: integer pgsbwd(integer devid, integer t)
SX Binding: (pg-set-border-width dev t)

Set the width of the window border in pixels.

C Binding: void PG_set_clipping(PG_device *dev, int flag)
F77 Binding: integer pgsclp(integer devid, integer c)
SX Binding: (pg-set-clipping! dev flag)

Turn on clipping to the current viewport if flag is TRUE and turn off clipping to the current viewport if flag is FALSE on the specified device. NOTE: moving the viewport after turning on the clipping does NOT move the clipping rectangle. To do this turn clipping off and back on again.

C Binding: void PG_set_fill_color(PG_device *dev, int color)
F77 Binding: integer pgsfcl(integer devid, integer color)
SX Binding: (pg-set-fill-color dev color)

Set the fill color for the device to color. The color index is mapped through the current palette.

C Binding: void PG_set_finish_state(PG_device *dev, int fin)
F77 Binding: integer pgsfin(integer dev, integer fin)
SX Binding: (pg-set-finish-state! dev fin)

Set the state of the flag that tells the high level rendering routines whether or not to assume a plot is finished and issue a call to PG_finish_plot. This is crucial when doing multiple plots or adding to a plot after the high level renderer returns.

C Binding: void PG_set_marker_orientation(PG_device *dev, double theta)
F77 Binding: integer pgsmko(integer devid, real theta)
SX Binding: (pg-set-marker-orientation! dev theta)

Set the orientation angle to be applied when drawing markers. Markers can be drawn at any angle. The angle, theta, is a uniform rotation from the positive x axis in the counter-clockwise direction of all the segments comprising the marker. It is measured in degrees.

C Binding: void PG_set_marker_scale(PG_device *dev, double v)

F77 Binding: integer pgsms(integer devid, real v)
SX Binding: (pg-set-marker-scale! dev v)

Set the scale factor to be applied when drawing markers. Markers can be drawn to any size since they are defined in normalized units (see PG_def_marker). The scale factor sets the actual size. A reasonable value might be 0.01.

C Binding: void PG_set_max_intensity(PG_device *dev, double osc)
F77 Binding:
SX Binding: (pg-set-maximum-intensity! dev osc rsc gsc bsc)

To better match the characteristics of varying output devices (especially conventional video) this function scales the overall intensity as well as the intensity of the RGB values down from their maximum of unity. The overall, red, green, and blue values are controlled by osc, rsc, gsc, and bsc respectively.

C Binding: PG_palette *PG_set_palette(PG_device *dev, char *name)
F77 Binding: integer pgspal(integer devid, integer nc, char *name)
SX Binding: (pg-set-palette! dev name)

Set the current palette to be the named one. The built-in palettes are named: standard, spectrum, rainbow, bw, wb, rgb, cym, hc, bgy, tri, iron, thresh, rand, reds, yellows, greens, cyans, blues, and magentas. Additional palettes may be read in with PG_rd_palette or created with PG_make_palette. The available palettes may be viewed with PG_show_palettes.

C Binding: void PG_set_res_scale_factor(PG_device *dev, double f)
F77 Binding:
SX Binding: (pg-set-resolution-scale-factor! dev sf)

Set the value of the hardcopy resolution scale factor. Hardcopy devices can be very high resolution devices which can lead to enormous image files. This control lets the application scale down the resolution of the device to keep image files a reasonable size. The default value is 8, that is by default the resolution is a factor of 8 less than could be obtained for the device. This means a factor of 64 in size for raster images.

C Binding: void PG_set_viewport_pos(PG_device *dev, REAL x, REAL y)
F77 Binding: integer pgsvps(integer devid, real x, real y)
SX Binding:

Set the position of the viewport in the window. The specifications are normalized.

C Binding: void PG_set_viewport_shape(PG_device *dev, REAL width, REAL height, REAL aspect)
F77 Binding: integer pgsvsh(integer devid, real width, real height, real aspect)
SX Binding:

Set the shape of the viewport in the window. The specifications are normalized. The aspect ratio is used iff the height is given as 0.0.

C Binding: void PG_turn_autodomain(PG_device *dev, int n)
F77 Binding: integer pgsadm(integer dev, integer dm)
SX Binding: (pg-set-autodomain! dev n)

Determine the domain interval from the data iff n or dm is ON.

C Binding: void PG_turn_autoplot(PG_device *dev, int n)
F77 Binding:
SX Binding: (pg-set-autoplot! dev n)

Set flag to applications to automatically replot iff n is ON. This is simply a global variable provided by PGS which applications may use to control plotting.

C Binding: void PG_turn_autorange(PG_device *dev, int n)
F77 Binding: integer pgsarn(integer dev, integer rn)
SX Binding: (pg-set-autorange! dev n)

Determine the range interval from the data iff n or rn is ON.

C Binding: void PG_turn_data_id(PG_device *dev, int n)
F77 Binding:
SX Binding: (pg-set-data-id-flag! dev n)

Draw data identifiers on plots iff n is ON.

C Binding: void PG_turn_grid(PG_device *dev, int n)
F77 Binding:
SX Binding: (pg-set-grid-flag! dev n)

Turn the full axis grid ON or OFF.

C Binding: void PG_turn_scatter(PG_device *dev, int n)
F77 Binding:
SX Binding: (pg-set-scatter-flag! dev n)

Draw 1D data sets as scatter plots iff n is ON.

C Binding: void PG_update_vs(PG_device *dev)
F77 Binding: integer pgupvs(integer devid)
SX Binding: (pg-update-view-surface dev)

Update the view surface of the specified device. This flushes any buffered graphics to the output medium of the device.

C Binding: void PG_white_background(PG_device *dev, int n)
F77 Binding:

SX Binding: (pg-set-white-background! dev n)

If n is TRUE use a white background otherwise use a black background.

Device Query Routines

C Binding: int COLOR_POSTSCRIPT_DEVICE(PG_device *dev)
F77 Binding:
SX Binding:

TRUE iff dev is a color PostScript device.

C Binding:
F77 Binding:
SX Binding: (pg-device-properties dev)

Return the name, type, and title of the device as given in the PG_make_device call which created the device.

C Binding: PG_dev_attributes *PG_get_attributes(PG_device *dev)
F77 Binding:
SX Binding:

Collect and return the selection of attributes from the specified device in a newly allocated PG_dev_attributes structure.

C Binding: void PG_get_clipping(PG_device *dev, int *flag)
F77 Binding: integer pggclp(integer devid, integer flag)
SX Binding: (pg-clipping? dev)

Get the current clipping state for the specified device in flag.

C Binding:
F77 Binding: integer pggfin(integer dev, integer fin)
SX Binding: (pg-finish-state dev)

Return the state of the flag that tells the high level rendering routines whether or not to assume a plot is finished and issue a call to PG_finish_plot.

C Binding: void PG_get_marker_orientation(PG_device *dev, REAL *v)
F77 Binding: integer pggmko(integer dev, real v)
SX Binding: (pg-marker-orientation dev)

Get the current marker orientation angle of the device. Markers can be drawn at any angle. The angle returned in v is a uniform rotation from the positive x axis in the counter-clockwise direction of all the segments comprising the marker. The angle is measured in degrees.

C Binding: void PG_get_marker_scale(PG_device *dev, REAL *s)
F77 Binding: integer pggmks(integer devid, real s)
SX Binding: (pg-marker-scale dev)

Get the current marker scale factor of the device. Markers can be drawn to any size since they are defined in normalized units (see PG_def_marker). The scale factor returned in s controls the actual size.

C Binding:
F77 Binding:
SX Binding: (pg-maximum-intensity dev)

Return the maximum intensity aggregate value and the individual values for red, green, and blue colors in the specified device. The values are normalized (0.0 to 1.0).

C Binding: PG_palette *PG_get_palette(PG_device *dev, char *name)
F77 Binding: not applicable
SX Binding: (pg-palette->list dev name)

Returns a pointer to the palette specified by name.

C Binding:
F77 Binding:
SX Binding: (pg-palettes dev)

Return a list of palettes available for the specified device.

C Binding: int POSTSCRIPT_DEVICE(PG_device *dev)
F77 Binding:
SX Binding:

TRUE iff dev is a PostScript device.

C Binding: void PG_query_screen(PG_device *dev, int *pdx, int *pdy, int *pnc)
F77 Binding: integer pgqdev(integer devid, integer dx, integer dy, integer nc)
SX Binding: (pg-query-device dev)

Query the device for size in pixels and color planes. The number of colors which a device supports is 2nplanes.

C Binding: void PG_query_window(PG_device *dev, int *pdx, int *pdy)
F77 Binding: integer pgqwin(integer devid, integer dx, integer dy)
SX Binding: (pg-query-window dev)

Query the shape of the window on the device.

C Binding:
F77 Binding:

SX Binding: (pg-show-markers)

Temporarily spawn a window to display the available marker characters on the specified device.

Coordinate Transformation Routines

These routines transform points from one coordinate system to another. The three coordinate systems are: pixel coordinates referring to the integer coordinates of pixels in a device; normalized coordinates whose values range from 0.0 to 1.0 and are device independent; and world coordinates which are user defined coordinates (see PG_set_window) tailored to the particular application at hand.

C Binding: void PtoS(PG_device *dev, int ix, int iy, REAL x, REAL y)
F77 Binding: integer pgptos(integer devid, integer ix, integer iy, REAL x, REAL y)
SX Binding: (pg-pixel->normalized dev ix iy)

Converts (ix, iy) from pixel coordinates to NDC/Screen coordinates (x, y).

C Binding: void StoP(PG_device *dev, REAL x, REAL y, int ix, int iy)
F77 Binding: integer pgstop(integer devid, REAL x, REAL y, integer ix, integer iy)
SX Binding: (pg-normalized->pixel dev x y)

Converts (x, y) from NDC/Screen coordinates to pixel coordinates (ix, iy).

C Binding: void StoW(PG_device *dev, REAL x, REAL y)
F77 Binding: integer pgstow(integer devid, REAL x, REAL y)
SX Binding: (pg-normalized->world dev x y)

Converts (x, y) from screen coordinates to world coordinates.

C Binding: void WtoS(PG_device *dev, REAL x, REAL y)
F77 Binding: integer pgwtos(integer devid, REAL x, REAL y)
SX Binding: (pg-world->normalized dev x y)

Converts (x, y) from world coordinates to screen coordinates.

Coordinate System and Viewport Control Routines

These routines provide access to the coordinate system and viewport of a PGS window.

C Binding: void PG_get_frame(PG_device *dev, REAL *x1, REAL *x2, REAL *y1, REAL *y2)
F77 Binding:
SX Binding: (pg-frame dev)

Get the frame of the specified device. The x and y intervals are specified in NDC by (x1, x2) and (y1, y2) respectively.

C Binding: void PG_get_viewport(PG_device *dev, REAL *x1, REAL *x2, REAL *y1, REAL *y2)
F77 Binding: integer pggvwp(integer devid, REAL x1, REAL x2, REAL y1, REAL y2)
SX Binding: (pg-viewport dev)

Get the viewport of the specified device. The x and y intervals are specified in NDC by (x1, x2) and (y1, y2) respectively.

C Binding: void PG_get_window(PG_device *dev, REAL *xmn, REAL *xmx, REAL *ymn, REAL *ymx)
F77 Binding: integer pggwcs(integer devid, REAL x1, REAL x2, REAL y1, REAL y2)
SX Binding: (pg-world-coordinate-system dev)

Get the world coordinate system defined relative to the viewport for the specified device. The x and y intervals are specified in WC by (x1, x2) and (y1, y2) respectively.

C Binding:
F77 Binding: integer pgrvpa(integer devid, integer n)
SX Binding:

Restore the current viewport, coordinate transformations, and related graphical state. A previously saved state (see pgsvpa) is referenced by the index n.

C Binding:
F77 Binding: integer pgsvpa(integer devid, integer n)
SX Binding:

Save the current viewport, coordinate transformations, and related graphical state. If n < 0, a new space is internally allocated. Otherwise n is interpreted as an existing state whose space will be reused. Returns an index in n which is to be used with a corresponding call to pgrvpa.

C Binding: void PG_set_frame(PG_device, double x1, double x2, double y1, double y2)
F77 Binding:
SX Binding: (pg-set-frame! dev x1 x2 y1 y2)

Set the frame of the specified device. The x and y intervals are specified in NDC by (x1, x2) and (y1, y2) respectively.

C Binding: void PG_set_limits(PG_device *dev, REAL *x, REAL *y, int n, int type)
F77 Binding:
SX Binding:

Set the world coordinate system (defined relative to the viewport) for the specified device finding the limits of the n points in the supplied data arrays. The x and y arrays are specified in WC. Type is the plot type (INSEL, HISTOGRAM, POLAR, CARTESIAN).

C Binding: void PG_set_viewport(PG_device *dev, double x1, double x2, double y1, double y2)
F77 Binding: integer pgsvwp(integer devid, REAL x1, REAL x2, REAL y1, REAL y2)
SX Binding: (pg-set-viewport! dev x1 x2 y1 y2)

Set the viewport of the specified device. The x and y intervals are specified in NDC by (x1, x2) and (y1, y2) respectively.

C Binding: void PG_set_window(PG_device *dev, double x1, double x2, double y1, double y2)
F77 Binding: integer pgswcs(integer devid, REAL x1, REAL x2, REAL y1, REAL y2)

SX Binding: (pg-set-world-coordinate-system! dev x1 x2 y1 y2)

Set the world coordinate system defined relative to the viewport for the specified device. The x and y intervals are specified in WC by (x1, x2) and (y1, y2) respectively.

Line Attribute Control Routines

These routines provide control over how lines appear when drawn.

C Binding: void PG_get_line_color(PG_device *dev, int *plc)
F77 Binding: integer pgglnc(integer devid, integer lc)
SX Binding: (pg-line-color dev)

C Binding: void PG_get_line_style(PG_device *dev, int *pls)
F77 Binding: integer pgglns(integer devid, integer ls)
SX Binding: (pg-line-style dev)

C Binding: void PG_get_line_width(PG_device *dev, REAL *plw)
F77 Binding: integer pgglnw(integer devid, REAL lw)
SX Binding: (pg-line-width dev)

C Binding: void PG_set_line_color(PG_device *dev, int lc)
F77 Binding: integer pgslnc(integer devid, integer lc)
SX Binding: (pg-set-line-color! dev lc)

C Binding: void PG_set_line_style(PG_device *dev, int ls)
F77 Binding: integer pgslns(integer devid, integer ls)
SX Binding: (pg-set-line-style! dev ls)

C Binding: void PG_set_line_width(PG_device *dev, double lw)

F77 Binding: integer pgslnw(integer devid, REAL lw)
SX Binding: (pg-set-line-width! dev lw)

These routines access the state contained in a PG_device pertaining to lines. Specifically color, width, or style. The set routines take the device and the new value, while the get routines take a device and the address where the current line attribute value is to be put. Line widths go from 0.0 (thinnest) on up with 3.0 being a very thick line. Line styles are: SOLID, DOTTED, DASHED, DOTDASHED.

Text Attribute Control Routines

The following provide control over text properties. A good deal of this is either unsupported by many host graphics systems or is superceded by the font based approach common in more modern host graphics systems. Users should preferentially use routines addressing themselves to font and type faces.

C Binding:
F77 Binding: integer pgscpw(integer devid, REAL x, REAL y)
SX Binding: (pg-set-char-path! dev x y)

This routine sets the direction along which text will be written.

C Binding: void PG_set_char_size_NDC(PG_device *dev, double w, double h)
F77 Binding:
SX Binding:

Set the current character size in normalized coordinates.

C Binding: void PG_set_char_space(PG_device *dev, double s)
F77 Binding:
SX Binding:

This routine sets the spacing between characters in world coordinates.

C Binding: void PG_set_char_up(PG_device *dev, double x, double y)
F77 Binding: integer pgscuw(integer devid, REAL x, REAL y)
SX Binding: (pg-set-char-up! dev x y)

This routine sets the direction along which characters will be oriented. This is usually orthogonal to the direction along which characters are written.

C Binding: void PG_set_font(PG_device *dev, char *face, char *style, int size)
F77 Binding: integer pgstxf(integer devid, integer ncf, char *face, integer ncs, char *style, integer size)
SX Binding: (pg-set-text-font! dev face style size)

This routine sets the font in the specified device. Face refers generically to the type face. PGS always supports helvetica, courier, and times. Style refers to the type style and the options are: medium, bold, italic, and bold-italic. Size refers to type size in points.

C Binding: void PG_set_text_color(PG_device *dev, int tc)
F77 Binding: integer pgstxc(integer devid, integer tc)
SX Binding: (pg-set-text-color! dev tc)

Set the text color for the specified device.

C Binding:
F77 Binding: integer pggcpw(integer devid, REAL x, REAL y)
SX Binding: (pg-character-path dev)

This routine returns the direction along which text will be written.

C Binding:
F77 Binding: integer pggcss(integer devid, REAL w, REAL h)
SX Binding: (pg-character-size-ndc dev)

Return the current character size in normalized coordinates.

C Binding:
F77 Binding: integer pggcsw(integer devid, REAL w, REAL h)
SX Binding:

Get the current character size in world coordinates.

C Binding: void PG_get_char_space(PG_device *dev, REAL *pcsp)
F77 Binding:
SX Binding:

This routine returns the spacing between characters in world coordinates.

C Binding: void PG_get_char_up(PG_device *dev, REAL *px, REAL *py)
F77 Binding: integer pggcuw(integer devid, REAL x, REAL y)
SX Binding: (pg-character-up dev)

This routine returns the direction along which characters will be oriented. This is usually orthogonal to the direction along which characters are written.

C Binding: void PG_get_font(PG_device *dev, char **face, char **style, int *size)
F77 Binding: integer pggtxf(integer devid, integer ncf, char *face, integer ncs, char *style, integer size)
SX Binding: (pg-text-font dev)

This routine queries the font in the specified device. Face refers generically to the type face. PGS always supports helvetica, courier, and times. Style refers to the type style and the options are: medium, bold, italic, and bold-italic. Size refers to type size in points. The FORTRAN binding has some extra behavior. The string lengths here are both input and output variables. On input they contain the lengths of the string buffers, face and style. On output they contain the number of actual characters in their respective strings. If the buffers are not long enough pggtxf returns FALSE and does nothing but return the lengths of the strings. The application can then make a second call with larger buffers.

C Binding: void PG_get_text_color(PG_device *dev, int *ptc)
F77 Binding: integer pggtxc(integer devid, integer tc)
SX Binding: (pg-text-color dev)

Query the text color for the specified device.

C Binding: void PG_get_text_ext(PG_device *dev, char *s, REAL *px, REAL *py)
F77 Binding: integer pggtew(integer devid, integer nc, char *s, REAL dx, REAL dy)
SX Binding: (pg-text-extent dev s)

This routine returns the world coordinate extent of the character string s as a width in px and height in py.

C Binding:
F77 Binding: integer pggtes(integer devid, integer nc, char *s, REAL dx, REAL dy)
SX Binding:

This routine returns the normalized coordinate extent of the character string s as a width in px and height in py.

Graphical Text I/O Routines

These routines provide the control over terminal or file I/O in graphical applications. The first two are directed at the special device, PG_console_device, which is opened with PG_open_console.

C Binding: void PG_center_label(PG_device *dev, double sy, char *label)
F77 Binding: integer pgwrcl(integer dev, real sy, integer nc, char *label)
SX Binding: (pg-center-label dev sy label)

This routine prints a text string, label, on the specified device and centered horizontally with a normalized vertical position specified by sy.

C Binding: char *PG_fgets(char *buffer, int maxlen, FILE *stream)
F77 Binding: integer pggtln(integer maxlen, char *buffer, integer stream)
SX Binding:

This function is call compatible with the standard C library fgets call. It gets input from the console device, PG_console_device, window if stream is stdin and from a file otherwise. In the FORTRAN binding using 0 for stream results in the use of stdin.

C Binding: int PG_fprintf(FILE *fp, char *fmt, ...)
F77 Binding: not applicable
SX Binding:

This function is call compatible with the standard C library fprintf call. It prints to the console device, PG_console_device, window if fp is stdout and to a file otherwise.

C Binding: int PG_write_abs(PG_device *dev, double x, double y, char *fmt, ...)
F77 Binding: integer pgwrta(integer devid, REAL x, REAL y, integer nc, char *txt)
SX Binding: (pg-draw-text-abs dev x y txt)

This routine does an sprintf style print to the specified device, dev, and at the world coordinate point specified by (x, y).

Point Move Routines

PGS maintains two "points", a text point and a drawing point, at which the next text and line drawing operations will start.

C Binding: void PG_move_gr_abs(PG_device *dev, double x, double y)
F77 Binding:
SX Binding:

C Binding: void PG_move_tx_abs(PG_device *dev, double x, double y)
F77 Binding:
SX Binding:

C Binding: void PG_move_tx_rel(PG_device *dev, double x, double y)
F77 Binding:
SX Binding:

These move the line drawing point (gr) or the text drawing point (tx) to an absolute world coordinate point or relative to the current world coordinate point.

Primitive Drawing Routines

These routines are fundamental drawing routines. Some of these are primitive in the sense that the host graphics systems all seem to supply them. The others are trivial applications of the primitive ones. However, these are termed fundamental in that most other PGS drawing routines are expressed directly in terms of them.

C Binding: void PG_draw_line(PG_device *dev, double x1, double y1, double x2, double y2)
F77 Binding: integer pgdrln(integer devid, REAL x1, REAL y1, REAL x2, REAL y2)
SX Binding: (pg-draw-line dev x1 y1 x2 y2)

This routine draws a line between two world coordinate points specified by (x1, y1) and (x2, y2).

C Binding: void PG_draw_to_abs(PG_device *dev, double x, double y)
F77 Binding:
SX Binding:

C Binding: void PG_draw_to_rel(PG_device *dev, double x, double y)
F77 Binding:
SX Binding:

These draw a line segment on the specified device to the absolute world coordinate point specified or relative to the current world coordinate point. These both reset the current drawing point to the destination endpoint.

C Binding: void PG_draw_polyline(PG_device *dev, REAL *x, REAL *y, int n, int clp)
F77 Binding: integer pgdpl2(integer devid, REAL x, REAL y, integer n, integer clp)
SX Binding: (pg-draw-polyline-2d dev clp x1 y1 x2 y2 ...)

This routine draws a connected line on the device, dev, starting with the first point and ending with the last point. The n points are specified in world coordinates and are contained in the x and y arrays. Clp causes the polyline to be clipped to the current viewport if TRUE.

C Binding: void PG_draw_disjoint_polyline_2(PG_device *dev, REAL *x, REAL *y, long n, int flag, int coord)
F77 Binding: integer pgsddp2(integer devid, REAL *x, REAL *y, integer n, integer flag, integer coord)
SX Binding: (pg-draw-disjoint-polyline-2d dev flag coord x1 y1 x2 y2 x3 y3 x4 y4 ...)

This routine draws n unconnected line segments whose 2n endpoints are specified in the x and y arrays (NOTE: x and y are 2n long!). If flag is TRUE and the device has either range or domain autoranging on, the limits of the points in x and y will be used to reset the world coordinate system. If coord is TRUE the points are taken to be in world coordinates and otherwise in normalized coordinates.

C Binding: void PG_shade_poly(PG_device *dev, REAL *x, REAL *y, int n)
F77 Binding: integer pgfply(integer devid, REAL *px, REAL *py, integer n, integer c)

SX Binding: (pg-fill-polygon dev c x1 y1 x2 y2 ...)

This routine draws a polygon specified by the n world coordinate points in the x and y arrays and fills it with the current fill color (see PG_set_fill_color).

Basic Line Drawing Routines

These routines add some creature comforts to PGS line drawing. They are still low level routines, but they handle some fairly common situations.

C Binding: void PG_draw_arc(PG_device *dev, double r, double a1, double a2, double

x, double y, int unit)
F77 Binding:
SX Binding: (pg-draw-arc dev r a1 a2 x y unit)

This routine draws an arc with radius, r, in world coordinates from angles a1 to a2 centered about the point (x, y) in world coordinates. Unit controls whether the angle is specified in degrees or radians. See the description for PG_draw_rad below for additional details.

C Binding: void PG_draw_box(PG_device *dev, double xmin, double xmax, double ymin, double ymax)
F77 Binding: integer pgdrbx(integer devid, REAL x1, REAL x2, REAL y1, REAL y2)

SX Binding: (pg-draw-box dev xmin xmax ymin ymax)

This routine draws a rectangle from xmin to xmax and from ymin to ymax which are specified in world coordinates.

C Binding: void PG_draw_disjoint_polyline_3(PG_device *dev, REAL *x, REAL *y, REAL *z, double theta, double phi, double chi, long n, int flag, int norm)
F77 Binding: integer pgddp3(integer devid, REAL *x, REAL *y, REAL *z, REAL theta, REAL phi, REAL chi, integer n, integer flag, integer norm)
SX Binding: (pg-draw-disjoint-polyline-3d dev theta phi chi flag norm x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4 ...)

This routine draws disjoint three dimensional line segments specified in world coordinates. The number of segments, n, is half the number of points. The arrays x, y, and z define the endpoint vectors X. X[2*i] is one endpoint of the ith segment, and X[2*i+1] is the other endpoint. The viewing angle is specified by theta, phi, and chi (in degrees). The flag norm determines whether the segment will be converted to normalized coordinates, and flag specifies whether the line segments are clipped to the viewport limits.

C Binding: void PG_draw_markers(PG_device *dev, int n, REAL *x, REAL *y, int marker)
F77 Binding: integer pgdrmk(integer dev, integer n, real x, real y, integer marker)
SX Binding: (pg-draw-markers dev marker x y)

This routine draws the marker character specified by marker at each of the n world coordinate points defined by the x and y arrays on the specified device.

C Binding: void PG_draw_rad(PG_device *dev, double rmin, double rmax, double a, double x, double y, int unit)
F77 Binding:
SX Binding: (pg-draw-radius dev rmin rmax a x y unit)

This routine draws a radial line from the central point (x, y) from rmin to rmax along the direction specified by the angle a (with respect to positive x axis increasing counter-clockwise). If unit has the value DEGREE then the angle a is in degrees and otherwise it is in radians (there is a RADIAN constant for symmetry).

Axis Drawing Routines

These routines handle various aspects of drawing axes. The most basic routine here draws a single axis with very general and controllable characteristics.

C Binding: void PG_axis(PG_device *dev, int axis_type)
F77 Binding: integer pgaxis(integer devid, integer axt)
SX Binding: (pg-axis device type)

This routine draws a set of axes which are tied to the viewport and world coordinate system. The valid values for axis_type are: CARTESIAN, POLAR, and INSEL. The axes are drawn on the view boundary. See the drawing model section above for the definition of the view boundary.

C Binding: void PG_axis_3d(PG_device *dev, REAL *px, REAL *py, REAL *pz, int n_pts, double theta, double phi, double chi, double xmn, double xmx, double ymn, double ymx, double zmn, double zmx, int norm)
F77 Binding: integer pgdax3(integer devid, REAL *x, REAL *y, REAL *z, integer n, REAL theta, REAL phi, REAL chi, integer norm)
SX Binding:

This routine draws a simple set of 3D axes oriented at the specified angle. The extent of the axes is determined by the n_pts points in (px, py, pz). The view angle for the axes is specified by theta, phi, and chi (in degrees). The unrotated limits of the data are specified by xmn, xmx, ymn, ymx, zmn, and zmx. If the axes should be drawn to normalized coordinates the norm flag should be TRUE.

C Binding: PG_axis_def *PG_draw_axis(PG_device *dev, double xl, double yl, double xr, double yr, double t1, double t2, double v1, double v2, double sc, char *format, int tick_type, int label_type, int flag, ...)
F77 Binding: integer pgdrax(integer devid, REAL xl, REAL yl, REAL xr, REAL yr, REAL t1, REAL t2, REAL v1, REAL v2, REAL sc, integer nc, char *format, integer tickdef, integer tick_type, integer label_type)
SX Binding: (pg-draw-axis dev xl yl xr yr t1 t2 v1 v2 sc format tick_type label_type tickdef)

This routine will draw a single axis and produce labels and/or ticks depending on the arguments. The arguments are:

(xl, yl)	coordinate of beginning end
(xr, yr)	coordinate of terminating end
(t1, t2)	fractional position of v1 and v2
(v1, v2)	first and last tick or label value
	an additional scale factor which is used, for example, when doing an

sc	Inselberg axis in which the range may correspond to one appropriate for the perpendicular dimension (set to 1.0 in most cases)
format	specifies the label format in the standard C way
tick_type	types of ticks

RIGHT_OF_AXIS	ticks on right
LEFT_OF_AXIS	ticks on left
STRADDLE_AXIS	ticks straddle (both)

label_type	types of labels

RIGHT_OF_AXIS	labels on right
LEFT_OF_AXIS	labels on left
NOTHING_ON_AXIS	no labels
ENDS	labels at ends of axis

tickdef	specifies the labels and ticks - one or more may be given and the list is terminated with a value of 0.

MAJOR	major ticks
MINOR	minor ticks
LABEL	labels

flag	if TRUE return a pointer to a PG_axis def structure (this is mainly for internal use) otherwise return NULL.

An axis is a directed line segment (from Xl to Xr) with ticks. The label values as defined by v1, v2, t1, and t2; the ticks associate with the line segment as follows:

(xl, yl) (xr, yr)

---> Axis

| | Ticks

v1 = v(t1) v2 = v(t2) Tick Labels (v)

C Binding: int PG_set_axis_attributes(PG_device *dev, ...)
F77 Binding: integer pgsaxa(integer devid, integer n, REAL attr, char *attrs)
SX Binding:

This routine sets the parameters which control the look of the axes being drawn. An arbitrary number of specifications can be made in key/value pairs. The list is terminated with a zero key. In the F77 binding the key/value pairs are placed as floating point numbers in the array attr. The value n is the number of pairs. If the attribute is a character string, the value is put in the character array attrs, and the number of characters is put in attr as the attribute value following the attribute key. The character strings are packed together with no space between the attribute values. This also means that the order of the attributes in the array attr must match those in attrs since only the number of characters is used to associate the attribute with its position in the string attrs.

The control keys are:

Name	Type	Value	Description
AXIS_LINESTYLE	integer	1	style of the lines
AXIS_LINETHICK	real	2	thickness of the lines
AXIS_LINECOLOR	integer	3	color of the lines
AXIS_LABELCOLOR	integer	4	color of the labels
AXIS_LABELFONT	char *	6	label font type face
AXIS_LABELPREC	integer	7	character precision
AXIS_X_FORMAT	char *	8	format of the x labels
AXIS_Y_FORMAT	char *	9	format of the y labels
AXIS_TICKSIZE	real	10	tick size in fraction of axis length
AXIS_GRID_ON	integer	11	turn on grid if TRUE
AXIS_SIGNIF_DIGIT	integer	12	number of digits in labels
AXIS_CHAR_ANGLE	real	13	orientation angle for label characters
			(not supported on all devices)

C Binding: void PG_get_axis_decades(REAL *d)
F77 Binding: integer pggaxd(real d)
SX Binding:

This routine returns the current maximum number of decades that log axes will span in the argument d. This facility is primarily aimed at making the use of logarithmic axes more flexible by defining a user controlled limit to the number of decades plotted. In this way, potentially ill-defined logarithmic values (such as very small positive numbers) don’t interfere with the display of otherwise fine values.

C Binding: void PG_set_axis_decades(REAL *d)
F77 Binding: integer pgsaxd(real d)
SX Binding:

This routine sets the current maximum number of decades that log axes will span to the value of the argument d. This facility is primarily aimed at making the use of logarithmic axes more flexible by defining a user controlled limit to the number of decades plotted. In this way, potentially ill-defined logarithmic values (such as very small positive numbers) don’t interfere with the display of otherwise fine values.

Colormap Related Routines

C Binding: void PG_show_colormap(PG_device *dev, int all)
F77 Binding:
SX Binding:

Display the colormap for the current palette if all is FALSE and for the entire device colormap if all is TRUE.

C Binding: void PG_show_palettes(PG_device *dev, char *type, int wbck)
F77 Binding:
SX Binding: (pg-show-palettes dev wbck)

This routine pops up a temporary new window to display the palettes available for the specified device. The user may browse through the palettes and select one to be the new current palette. When the selection is made the palette window goes away. The type argument specifies the type of device to be spawned (WINDOW, PS, or CGM), this is useful for making hardcopies of the palette set. The wbck argument is TRUE if a white background is wanted and FALSE for a black one.

C Binding: PG_palette *PG_make_palette(PG_device *dev, char *name, int nc, int wbck)
F77 Binding:
SX Binding: (pg-make-palette dev name nc wbck)

This routine pops up a temporary new window with the available colors of the device and lets the user select from those colors to build up a new palette. When finished the window is dismissed and the new palette becomes the current palette of the device. Colors are selected by clicking on them with the left mouse button. Any number of colors may be selected in this fashion, however, colors selected after the ncth replace previously selected colors. When the new palette is completed, clicking the right button signals acceptance and the window goes away. The new palette is written into a file whose name is the same as the palette name and with a ".pal" extension. The number of colors requested is nc and if wbck is TRUE the temporary window has a white background (otherwise a black one). The palette file can be read in again with the PG_rd_palette function.

C Binding: PG_palette *PG_rd_palette(PG_device *dev, char *fname)
F77 Binding:
SX Binding: (pg-read-palette dev fname)

This function reads a palette file and makes the resulting palette the current palette of the specified device. The format of a palette file is simple. It is an ASCII file whose first line contains the name of the palette and the number of colors, nc. The next nc lines contain normalized red, green, and blue values.

C Binding: int PG_wr_palette(PG_device *dev, PG_palette *pal, char *fname)
F77 Binding:
SX Binding: (pg-write-palette dev pal fname)

This function writes a specified palette, pal, to a palette file. The format of a palette file is simple. It is an ASCII file whose first line contains the name of the palette and the number of colors, nc. The next nc lines contain normalized red, green, and blue values.

Graph Control Routines

These routines control the state of graphs or act on them.

These routines are at the other end of the scale from the previous low level routines. They perform complex drawing operations which are driven by user oriented specifications.

C Binding: void PG_draw_graph(PG_device *dev, PG_graph *data)
F77 Binding: integer pgplot(integer devid, integer dataid)
SX Binding: (pg-draw-graph dev [data]* rendering)

This routine draws the graph specified by data on the device specified by dev. The PG_graph structure contains both data and rendering specifications.

C Binding: void PG_domain_plot(PG_device *dev, PM_set *dom, PM_set *ran)
F77 Binding: integer pgdplt(integer devid, integer domid)
SX Binding: (pg-draw-domain dev [dom]* [type extrema])

This routine draws the domain set specified by dom on the device specified by dev. The PM_set structure contains the data. This is the generalization of a mesh plot. A range set, ran, may optionally be provided if labels or other information associated with the mesh points are desired. The FORTRAN and SX bindings do not allow a range to be given at this time. The SX binding does allow for plotting limits to be set on the domain. They are specified by n (min, max) pairs where n is the dimensionality of the domain. The plot type may also be directly specified as PLOT_SURFACE, PLOT_MESH, or PLOT_WIRE_MESH instead of in the attribute list of the PM_set. The default is PLOT_WIRE_MESH

C Binding: void PG_get_identifier(PG_graph *g, int id)
F77 Binding: integer pgggid(integer gid, integer id)
SX Binding:

Return the identifier character, id, of the specified graph. In the C binding this is a macro.

C Binding: void PG_get_render_info(PG_graph *g, pcons *alst)
F77 Binding: integer pgginf(integer gid, integer alst)
SX Binding:

Return the attribute list, alst, of the specified graph. In the C binding this is a macro.

C Binding:
F77 Binding: integer pgsdlm(integer grid, integer n, REAL v)

SX Binding: (pg-set-domain-limits! gr v1mn v1mx ...)

Restrict the domain of the mapping contained in the graph to the values in the array v. There are 2n values in (min, max) pairs where n is the dimensionality of the domain.

C Binding: void PG_set_identifier(PG_graph *g, int id)
F77 Binding: integer pgsgid(integer gid, integer id)
SX Binding:

Set the identifier character, id, of the specified graph. In the C binding this is a macro.

C Binding: void PG_set_render_info(PG_graph *g, pcons *alst)
F77 Binding: integer pgsinf(integer gid, integer alst)
SX Binding:

Set the attribute list, alst, of the specified graph. In the C binding this is a macro.

C Binding:
F77 Binding: integer pgsrat(integer grid, integer n, char *name, integer t, char *type, char *val)
SX Binding: (pg-set-graph-attribute! gr name type val)

Set a single rendering attribute in the specified graph. The name of the attribute is in name, its type in type and its value in val. See the section on attributes for more information on attributes and their values.

C Binding: void PG_set_plot_type(pcons *inf, int plt, int axs)
F77 Binding:
SX Binding:

Set the plot type, plt, and axis type, axs, for the specified association list, inf. This list is usually the info part of a PG_graph.

C Binding:
F77 Binding: integer pgsrlm(integer grid, integer n, REAL v)
SX Binding: (pg-set-range-limits! gr v1mn v1mx ...)

Restrict the range of the mapping contained in the graph to the values in the array v. There are 2n values in (min, max) pairs where n is the dimensionality of the range.

C Binding:
F77 Binding: integer pgsvlm(integer grid, REAL v)
SX Binding:

A PG_graph may have its own viewport limits which supercede the device viewport limits. This function takes an array, v, containing the limits and attaches them to the info list of the specified graph, grid. The limits are arranged as xmin, xmax, ymin, ymax.

Line Plot Routines

These routines plot curves in various renderings.

C Binding: void PG_plot_curve(PG_device *dev, REAL *x, REAL *y, int n, pcons *info, int l)
F77 Binding: integer pgplln(integer devid, REAL *px, REAL *py, integer n, integer mod, integer axt, integer col, REAL wid, integer sty, integer sca, integer mrk, integer sta, integer l)
SX Binding: use pg-draw-graph

This routine is a moderately high level routine in that it will call the other routines depending on the values in the association list info and plot the n points in the x and y arrays. In the FORTRAN binding the following attributes may be passed directly:

mod	plot type (CARTESIAN, POLAR, INSEL)
axt	axis type (CARTESIAN, POLAR, INSEL)
col	line color
wid	line width
sty	line style
sca	scatter plot flag
mrk	marker index
sta	histogram starting point (LEFT, CENTER, RIGHT)
l	if FALSE the world coordinates are recomputed from the data and the axes are drawn

C Binding: void PG_histogram_plot(PG_device *dev, REAL *x, REAL *y, int n, int lncol, double lnwid, int lnsty, int scatter, int marker, int start, int l)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

C Binding: void PG_insel_plot(PG_device *dev, REAL *x, REAL *y, int n, int lncol, double lnwid, int lnsty, int l)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

C Binding: void PG_polar_plot(PG_device *dev, REAL *x, REAL *y, int n, int lncol, double lnwid, int lnsty, int scatter, int marker, int l)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

C Binding: void PG_rect_plot(PG_device *dev, REAL *x, REAL *y, int n, int lncol, double lnwid, int lnsty, int scatter, int marker, int l)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

These routines plot the n points in the x and y arrays as a: rectangular cartesian plot; histogram plot; Inselberg plot; or polar plot. The qualifying arguments are:

lncol	line color
lnwid	line width
lnsty	line style
scatter	TRUE for scatter plot
marker	index of marker character for scatter plots
start	LEFT, RIGHT, or CENTER start for histogram plots
l	if TRUE the world coordinate system is redefined by x and y data

Contour Plotting Routines

C Binding: void PG_contour_plot(PG_device *dev, PG_graph *data)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

This routine renders the specified graph as a contour plot. PG_draw_graph dispatches to this routine when the rendering specified in data is a contour plot.

C Binding:
F77 Binding: integer pgplcn(integer devid, REAL x, REAL y, REAL z, REAL lev, integer k, integer l, integer nlev, integer labl, integer alst)
SX Binding: use pg-draw-graph

Make a contour plot of the given data set on the specified device. The arguments are:

(x, y)	coordinate point arrays
z	data array
lev	contour level value array
(k, l)	array dimensions
nlev	number of contour levels
labl	starting label character if non-zero
alst	an integer attribute list identifier (use 0 if none)

C Binding: int PG_contour_levels(REAL *lev, int nlev, double fmn, double fmx, double ratio)
F77 Binding: integer pgclev(REAL lev, integer nlev, REAL fmn, REAL fmx, REAL ratio)
SX Binding: set LEVEL attribute

Compute an array of nlev iso contour levels between fmn and fmx using the spacing ratio. Put them in the space provided, lev. The arguments are:

lev	the array of contour levels
nlev	the number of contour levels
(fmn, fmx)	the minimum and maximum values for contour levels
ratio	the spacing ratio between contour levels

Filled Polygon Plot Routines

C Binding: void PG_poly_fill_plot(PG_device *dev, PG_graph *data)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

This routine renders the specified graph as a filled polygon plot. PG_draw_graph dispatches to this routine when the rendering specified in data is a a filled polygon plot.

Image Plot Routines

C Binding: void PG_image_plot(PG_device *dev, PG_graph *data)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

This routine renders the specified graph as a raster image plot. PG_draw_graph dispatches to this routine when the rendering specified in data is an image plot.

C Binding: void PG_draw_image(PG_device *dev, PG_image *im, char *label, pcons *alist)
F77 Binding:
SX Binding: (pg-draw-image dev im)

This routine makes an image plot from a PG_image structure. A pointer to the image is in im and a label for the plot is in label.

C Binding:
F77 Binding: integer pgplim(integer devid, integer nc, char *name, integer nct, char *type, REAL *pz, integer k, integer l, REAL xmn, REAL xmx, REAL ymn, REAL ymx, REAL zmn, REAL zmx, integer alst)
SX Binding:

This routine makes an image plot from "raw" data. The arguments are:

name	the name of the image (used as a label for the plot)
type	the type of the data ("char", "short", "int", "long", "float", "double")
z	the array of pixel values (will be scaled to the current palette of the device)

(k, l)	the dimensions of the image
(xmn, xmx)	the minimum and maximum x values (for axis labels)
(ymn, ymx)	the minimum and maximum y values (for axis labels)
(zmn, zmx)	the nminimum and maximum data values (for palette labels)
alst	an integer attribute list identifier (use 0 if none)

C Binding: void PG_draw_palette(PG_device *dev, double xmn, double ymn, double xmx, double ymx, double zmn, double zmx, double wid)
F77 Binding: integer pgdrpa(integer devid, REAL xmn, REAL ymn, REAL xmx, REAL ymx, REAL zmn, REAL zmx, REAL wid)
SX Binding: (pg-draw-palette dev xmn ymn xmx ymx zmn zmx wid)

This routine draws the specified device palette as a raster image. The palette is drawn next to the viewport in a rectangle along an axis specified by the points (xmn, ymn) and (xmx, ymx). The axis is labeled by values ranging from zmn to zmx. The width of the palette in the rectangle is specified in normalized form by wid.

Surface Plot Routines

C Binding: void PG_surface_plot(PG_device *dev, PG_graph *data)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

This routine renders the specified graph as a surface plot. PG_draw_graph dispatches to this routine when the rendering specified in data is a surface plot.

C Binding: void PG_draw_surface(PG_device *dev, REAL *a1, REAL *a2, REAL *aext, REAL *x, REAL *y, int nn, double xmn, double xmx, double ymn, double ymx, double theta, double phi, double chi, double width, int color, int style, int type, char *label, char *mesh_type, void *cnnct, pcons *alist)
F77 Binding: integer pgplsf(integer devid, REAL *px, REAL *py, REAL *pz, integer n, REAL xn, REAL xx, REAL yn, REAL yx, REAL zn, REAL zx, integer kx, integer lx, REAL th, REAL ph, REAL ch, integer typ, integer col, REAL wid, integer sty, integer nc, char *label)
SX Binding: use pg-draw-graph

This routine is a medium level routine which can be called directly from applications. The data can be rendered as a wire frame mesh or as a true shaded surface. Both forms do hidden line and hidden surface removal. The algorithm uses a raster scan line approach with a single Z buffer line. This choice minimizes the memory requirements of the routine at some expense of speed.

The arguments are:

dev	the device to which the plot is drawn
	the arrays specifying height of the surface and color

a1, a2	shading. If the color array is NULL the a1 array will do both height and color in a shaded plot
aext	the minimum and maximum values for a1 and a2
x, y	the x and y components of the positions of the nodes
nn	the number of points
xmn, xmx	the minimum and maximum values for x
ymn, ymx	the minimum and maximum values for y
theta, phi, chi	the Euler view angles
width	the line width to use
color	the line color to use
style	the line style to use
type	PLOT_SURFACE or PLOT_WIRE_MESH
name	a label for the plot
mesh_type	"Logical-Rectangular" (LR) or "Arbitrarily-Connected" (AC)
cnnct	connectivity specifications: array of dimensions for LR meshes or a pointer to a PM_mesh_topology struct for AC meshes
alist	association list of plotting attributes

Vector Plot Routines

C Binding: void PG_vector_plot(PG_device *dev, PG_graph *data)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

This routine renders the specified graph as a vector plot. PG_draw_graph dispatches to this routine when the rendering specified in data is a vector plot.

C Binding:
F77 Binding: integer pgplvc(integer devid, REAL *px, REAL *py, REAL *pu, REAL *pv, integer n, integer alst)
SX Binding: use pg-draw-graph

Make a vector plot of the given data set on the specified device.

devid	an integer attribute list identifier (use 0 if none)
(px, py)	farrays containing the x and y positions
(pu, pv)	farrays containing the u and v vector components
n	an integer number of vectors
alst	an integer attribute list identifier (use 0 if none)

C Binding: void PG_set_vec_attr(PG_device *dev, ...)
F77 Binding: integer pgsvat(integer devid, ...)
SX Binding: (pg-set-vector-attributes! dev ...)

This routine sets the properties of the vectors for the next vector plot. The parameters are paired, optional, and can be in any order. For each pair, the first value describes the option, the second, the value. The options are ints. The values can be ints, REALs, or chars. What type the values are is determined by the option. Most values are normalized to unity with the angle being the exception. The list must be ended with a zero. The attributes are:

C Id	FORTRAN Id	Description	Default
VEC_SCALE	1	scale factor on lengths	1.0
VEC_ANGLE	2	angle between wings	22.5
VEC_HEADSIZE	3	length of the wings	0.05
VEC_FIXSIZE	4	a fixed vector length	0.0
VEC_MAXSIZE	5	a maximum vector length	0.0
VEC_LINESTYLE	6	line style of vectors	1
VEC_LINETHICK	7	line width of vectors	0.0
VEC_COLOR	8	line color of vectors	WHITE
VEC_FIXHEAD	9	a fixed head size	FALSE

Level Diagram Plot Routine

This routine draws a level diagram or grotrian plot.

C Binding: void PG_grotrian_plot(PG_device *dev, PG_graph *data)
F77 Binding: use pgplot
SX Binding: use pg-draw-graph

This routine renders the specified graph as a level diagram plot. PG_draw_graph dispatches to this routine when the rendering specified in data is a level diagram plot. These graphs have an unusual data layout and would not produce a very meaningful plot rendered any other way.

Graphical Interface Routines

These routines supply a capability for applications to develop portable, user and runtime configurable graphical user interfaces.

C Binding: void PG_draw_interface_objects(PG_device *dev)
F77 Binding: integer pgdrif(integer devid)
SX Binding:

This routine draws the entire graphical interface of the specified device.

C Binding: void PG_query_pointer(PG_device *dev, int *px, int *py, int *pbtn, int *mod)
F77 Binding: integer pgqptr(integer devid, integer x, integer y, integer btn, integer mod)
SX Binding:

This routine allows the application to query the state of the locator or mouse. The mouse state is: the (x, y) position on the screen in pixel coordinates; btn, the indicator of mouse buttons which are pressed (MOUSE_LEFT, MOUSE_CENTER, MOUSE_RIGHT); and mod the indicator of which keyboard modifiers are pressed (KEY_SHIFT, KEY_CNTL, KEY_ALT). These flags can be and'd to ascertain which combinations are pressed.

C Binding: int PG_read_interface(PG_device *dev, char *fname)
F77 Binding: integer pgrdif(integer devid, integer n, char *fname)
SX Binding:

This routine reads the entire graphical interface from the named file into the specified device.

C Binding: void PG_register_callback(char *name, PFVoid fnc)
F77 Binding: integer pgrgfn(integer nc, char *name, function fnc)
SX Binding:

This routine registers a function with PGS so that it may be called by an interface event such as the click of a button or selection of some item of an interface. The name is used to refer to the function (via a lookup) in such applications

C Binding: void PG_register_variable(char *name, char *type, void *var, void *vmin, void *vmax)
F77 Binding: integer pgrgvr(integer nc, char *name, integer nt, char *type, var, vmin, vmax)
SX Binding:

This routine registers a variable with PGS so that its value may be changed by an interface event such as the selection of some item of from a menu. The required information is the variables name, type, and address (var). Optionally pointers to the minimum and maximum values, vmin and vmax, may be supplied (NULL if not wanted).

C Binding: int PG_write_interface(PG_device *dev, char *fname)
F77 Binding: integer pgwrif(integer devid, integer n, char *fname)
SX Binding:

This routine writes the entire graphical interface of the specified device to the named file.

Structures

PGS employs several structures to encapsulate information pertaining to specific groups of functionality. C based applications can have access to these structures (not a good idea in general because the structures may change) and some readers may find the structures revealing of details in the design and implementation of PGS. Some of these are also discussed in the context of rendering. See those sections for additional details.

PG_graph

The PG_graph structure contains information specifying how data is to be rendered. That includes specification of the rendering technique, line attributes, and other information. A PG_graph always contains a PM_mapping (a structure defined by the PACT library PML) pointer for the data set to be rendered.

PG_graph's can be linked together in a list and the PGS rendering functions will render all graphs in the list.

An associated type of the PG_graph is:

```
typedef PG_graph *(*PFPPG_graph)();
```

The last type is a Pointer to a Function returning a Pointer to a PG_GRAPH. This convention is used throughout PACT. See the related documentation for further information.

PG_palette

The PG_palette structure contains the specification of a palette of RGB colors. The number of colors in the palette is determined by the application. If the number of colors specified exceeds the number of colors which the host platform can display, PGS attempts to simulate the colors with dithering of colors which the host graphics system can display.

PG_device

The PG_device structure contains the information which the host graphics system requires applications to provide and maintain and it keeps a set of state variables describing such quantities as coordinate systems, palettes, and drawing attributes.

PG_image

The PG_image structure contains the specification for cell array or image plots. It contains data, the data type, the array dimensions, bounding values for use in defining scales, palette information, and other appropriate data.

PG_dev_attributes
The PG_dev_attributes structure contains a large number of the commonly queried and set attributes found in the PG_device. The intent of this structure is to allow applications to access and change many device attributes quickly and efficiently. The attributes include line, text, and fill colors, line style and width, palette, and clipping state. It is convenient to save these all at once in a PG_dev_attributes, change the device state, perform drawing operation, and restore the original device state from the PG_dev_attributes.

PGS Constants

PGS defines and uses several #define'd constants. These can be used by applications and are listed here by category.

General Purpose Constants

PG_IMAGE_VERSION	0
N_COLORS	16
QUAD_ONE	1
QUAD_FOUR	4
N_ANGLES	180
DEGREE	1
RADIAN	2

Device Characterization Constants

TEXT_WINDOW_DEVICE	128
GRAPHIC_WINDOW_DEVICE	129
PS_DEVICE	130
CGMF_DEVICE	131
HARD_COPY_DEVICE	133
SCREEN_DEVICE	134

Rendering Clear Mode Constants

CLEAR_SCREEN	-5
CLEAR_VIEWPORT	-6
CLEAR_FRAME	-7

Axis Description Constants

AXIS_LINESTYLE	1
AXIS_LINETHICK	2
AXIS_LINECOLOR	3
AXIS_LABELCOLOR	4
AXIS_LABELSIZE	5
AXIS_LABELFONT	6
AXIS_LABELPREC	7
AXIS_X_FORMAT	8
AXIS_Y_FORMAT	9
AXIS_TICKSIZE	10
AXIS_GRID_ON	11
AXIS_SIGNIF_DIGIT	12
AXIS_CHAR_ANGLE	13
MAJOR	1
MINOR	2
LABEL	3
MAJOR_MINOR	4
MAJOR_LABEL	5
MINOR_LABEL	6
MAJOR_MINOR_LABEL	7
NO_TICKS	8
RIGHT_OF_AXIS	9
LEFT_OF_AXIS	10
STRADDLE_AXIS	11
ENDS	12
NOTHING_ON_AXIS	13
NOTICKS	0
INSIDE	1
OUTSIDE	2
INOUT	3
TICKTICK	1
LINELINE	2
TICKLINE	3
LINETICK	4

Grid Description Constants

GRID_LINESTYLE	1
GRID_LINETHICK	2
GRID_LINECOLOR	3
GRID_LABELCOLOR	4
GRID_LABELSIZE	5
GRID_LABELFONT	6
GRID_LABELPREC	7
GRID_XFORMAT	8
GRID_YFORMAT	9
GRID_TICKPOSITION	10
GRID_TICKSIZE	11
GRID_TICKTYPE	12
GRID_SIGNIF_DIGIT	13

Vector Description Constants

VEC_SCALE	1
VEC_ANGLE	2
VEC_HEADSIZE	3
VEC_FIXSIZE	4
VEC_MAXSIZE	5
VEC_LINESTYLE	6
VEC_LINETHICK	7
VEC_COLOR	8
VEC_FIXHEAD	9

Line Plot Types

CARTESIAN	-1
POLAR	-2
INSEL	-3
HISTOGRAM	-4
SCATTER	-5
LOGICAL	-6

ERROR_BAR	-7

Rendering Mode Constants

PLOT_CURVE	10
PLOT_CONTOUR	11
PLOT_IMAGE	12
PLOT_WIRE_MESH	13
PLOT_SURFACE	14
PLOT_VECTOR	15
PLOT_FILL_POLY	16
PLOT_MESH	17
PLOT_ERROR_BAR	18
PLOT_DEFAULT	19

Glossary

Here is a list of terms which are used in this manual.

device	A generic name for a graphical output device such as a display window, a PostScript file. PGS supports WINDOW (display screen), PS (PostScript), and CGM devices.
console	A special device which mimics a UNIX shell window on platforms which lack that functionality (e.g. Macintosh)
NDC	Normalized device coordinates or screen coordinates. Values range between 0.0 and 1.0.
image	An array of pixel values also called a raster image or cell array
graph	Generically the collection of information needed to visualize a collection of data. More specifically a PG_graph structure.
set	Generically a collection of related data. More specifically a PM_set structure.
mapping	Generically a rule of association between elements of two sets. More specifically a PM_mapping structure.
palette	An application oriented representation of a color scheme to be associated with data in a plot.
colormap	A host oriented representation of a color scheme to be associated with colors in hardware.
marker	A user definable "character". A collection of line segments

marker which can be rotated and scaled collectively.

connectivity The collection of neighbor relationships between points of a computational mesh.

PGS By Example

Perhaps the best way to learn to use PGS is by example. Certainly it is the easiest way to explain certain aspects of it. In this section there are examples of some of the low level graphics primitives and the high level rendering functions. The actual tests are:

Text Placement and Drawing

Line Drawing

Making Line Plots

Making Contour Plots

Making Image Plots

Making Wire Frame Mesh Plots

Making Vector Plots

Before showing the examples we discuss some sequences of PGS calls that are common to most PGS applications. The hope is that this will make clear why some of PGS functions exist and how they relate to one another.

Common Call Sequences

The PGS functions are generally not very meaningful taken one at a time. What is important is the way they are used together to accomplish various graphical objectives. We are not focusing on the details of the calls here so many details are omitted. Concentrate on which calls are being made and when they are performed.

Initializing a Device

The first basic job in a graphics application is to setup the devices to be used. Most common is setting up screen windows. Also important is initializing hardcopy devices such as a PostScript device. The sequence is all the same, some of the parameters differ.

```
dev = PG_make_device(...);
```

```
PG_set_viewport_pos(dev, ...);

PG_set_viewport_shape(dev, ...);

PG_white_background(dev, ...);


PG_open_device(dev, ...)
```

The PG_make_device call only allocates a PG_device structure and sets default values for the state it contains. The PG_open_device call actually consults the state of the PG_device and opens the device. The calls in between change the state of the PG_device from the defaults.

Making a Picture

The next basic idea is to put together a picture. This may be done with low level drawing operations or high level rendering calls or both. What is common to all is that the picture must be set up, drawn, and finished.

```
PG_clear_window(dev);

       ...

      draw

       ...

PG_update_vs(dev);

       ...

PG_finish_plot(dev);
```

The call to PG_clear_window must be done for hardcopy devices! It is a pretty good idea for screen window devices too. Sometimes in the process of drawing a picture you want to see the results so far with the idea that more will be drawn later. PG_update_vs makes sure that everything the has been requested is visible. When the picture is complete and there is nothing more to be drawn to it, PG_finish_plot is called. This is crucial for hardcopy devices! It is also crucial that PG_clear_window and PG_finish_plot be called once per picture.

Setting Attributes

To gain control of the appearance of high level plots, applications must set rendering attributes. See the discussion of rendering attributes before going any further here. The sequence is to use PG_get_render_info to obtain the attribute list from the graph, use SC_change_alist to change or add values to the list, and use PG_set_render_info to update the graph's attribute list.

Here are C and FORTRAN examples of setting some attributes in a graph. In this example some attributes are set for contour plotting.

C

```
PG_graph *g;
pcons *alst;
double *clev;
int *nlev;

nlev = MAKE(int);
*nlev = 6;
clev = MAKE_N(double, *nlev);

PG_get_render_info(g, alst);
alst = SC_change_alist(alst, "LEVELS", "double *", clev);
alst = SC_change_alist(alst, "N-LEVELS", "int *", nlev);
PG_set_render_info(g, alst);
```

FORTRAN

```
integer gid, ial, nlev
real clev(10)

call pgginf (gid, ial)
call scchal (ial, 6, 'LEVELS', 6, 'double', nlev, clev)
call scchal (ial, 8, 'N-LEVELS', 7, 'integer', 1, nlev)
call pgsinf (gid, ial)
```

Text Placement and Drawing

The following program demonstrates some of the PGS functionality for placement and drawing of text.

```
#include "pgs.h"

/*--------------------------------------------------------------------*/

void main(argc, argv)
   int argc;
   char *argv[];
   {char s[MAXLINE], *token;
    PG_device *SCR_dev, *PS_dev, *CGM_dev;
    REAL x1, y1, x2, y2, dx, dy;
    char *face, *style;
    int size;

/* connect the I/O functions */
    PG_open_console("PGS Test", "COLOR", TRUE, 0.1, 0.7, 0.5, 0.3);

    SCR_dev = PG_make_device("WINDOW", "COLOR", "PGS Test");
    PG_open_device(SCR_dev, 0.1, 0.1, 0.5, 0.6);
    PG_set_viewport(SCR_dev, 0.1, 0.9, 0.2, 0.8);
    PG_set_window(SCR_dev, 0.0, 1.0, 0.0, 1.0);
    PG_draw_box(SCR_dev, -0.02, 1.02, -0.02, 1.02);

    CGM_dev = PG_make_device("CGM", "MONOCHROME", "gstxts");
    PG_open_device(CGM_dev, 0.1, 0.1, 0.8, 0.8);

    PS_dev = PG_make_device("PS", "MONOCHROME", "gstxts");
    PG_open_device(PS_dev, 0.1, 0.1, 0.8, 1.1);

    PG_expose_device(PG_console_device);
```

```
    PG_clear_window(SCR_dev);
    PG_clear_window(CGM_dev);
    PG_clear_window(PS_dev);

    test_dev(SCR_dev);
    test_dev(CGM_dev);
    test_dev(PS_dev);

    PG_finish_plot(SCR_dev);
    PG_finish_plot(CGM_dev);
    PG_finish_plot(PS_dev);

    SC_pause();

    PG_close_device(SCR_dev);
    PG_close_device(CGM_dev);
    PG_close_device(PS_dev);

    exit(0);}

/*------------------------------------------------------------------*/

/* SF_DT - set the font and draw the text */

static void sf_dt(dev, x1, y1, face, style, size)
   PG_device *dev;
   double x1, y1;
   char *face, *style;
   int size;
   {double x2, y2, dx, dy;
```

```
    PG_set_font(dev, face, style, size);

    PG_get_text_ext(dev, "foo", &dx, &dy);

/* write some text and draw a box around it */
    PG_write_abs(dev, x1, y1, "%s", "foo");

    x2 = x1 + dx;
    y2 = y1 + dy;
    PG_draw_box(dev, x1, x2, y1, y2);

    return;}


/*-----------------------------------------------------------------*/

/* TEST_DEV - test the entire device */

static void test_dev(dev)
   PG_device *dev;
   {PG_set_line_color(dev, dev->BLACK);
    PG_set_text_color(dev, dev->BLACK);


    sf_dt(dev, .1, .9, "helvetica", "medium", 12);
    sf_dt(dev, .1, .8, "helvetica", "italic", 12);
    sf_dt(dev, .1, .7, "helvetica", "bold", 12);
    sf_dt(dev, .1, .6, "helvetica", "bold-italic", 12);


    sf_dt(dev, .1, .50, "helvetica", "medium", 10);
    sf_dt(dev, .1, .45, "helvetica", "italic", 10);
    sf_dt(dev, .1, .40, "helvetica", "bold", 10);
    sf_dt(dev, .1, .35, "helvetica", "bold-italic", 10);


    sf_dt(dev, .1, .30, "helvetica", "medium", 8);
    sf_dt(dev, .1, .25, "helvetica", "italic", 8);
```

```
sf_dt(dev, .1, .20, "helvetica", "bold", 8);
sf_dt(dev, .1, .15, "helvetica", "bold-italic", 8);

sf_dt(dev, .3, .9, "times", "medium", 12);
sf_dt(dev, .3, .8, "times", "italic", 12);
sf_dt(dev, .3, .7, "times", "bold", 12);
sf_dt(dev, .3, .6, "times", "bold-italic", 12);

sf_dt(dev, .3, .50, "times", "medium", 10);
sf_dt(dev, .3, .45, "times", "italic", 10);
sf_dt(dev, .3, .40, "times", "bold", 10);
sf_dt(dev, .3, .35, "times", "bold-italic", 10);

sf_dt(dev, .3, .30, "times", "medium", 8);
sf_dt(dev, .3, .25, "times", "italic", 8);
sf_dt(dev, .3, .20, "times", "bold", 8);
sf_dt(dev, .3, .15, "times", "bold-italic", 8);

sf_dt(dev, .5, .9, "courier", "medium", 12);
sf_dt(dev, .5, .8, "courier", "italic", 12);
sf_dt(dev, .5, .7, "courier", "bold", 12);
sf_dt(dev, .5, .6, "courier", "bold-italic", 12);

sf_dt(dev, .5, .50, "courier", "medium", 10);
sf_dt(dev, .5, .45, "courier", "italic", 10);
sf_dt(dev, .5, .40, "courier", "bold", 10);
sf_dt(dev, .5, .35, "courier", "bold-italic", 10);

sf_dt(dev, .5, .30, "courier", "medium", 8);
sf_dt(dev, .5, .25, "courier", "italic", 8);
sf_dt(dev, .5, .20, "courier", "bold", 8);
```

```
    sf_dt(dev, .5, .15, "courier", "bold-italic", 8);

    PG_update_vs(dev);

    return;}

/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
```

Line Drawing

This program illustrates some of the PGS calls for handling drawing attributes and for drawing lines.

```
 #include "pgs.h"

char
 *color[] = {"BLACK",
             "WHITE",
             "LGHT_WHITE"
             "GRAY",
             "BLUE",
             "GREEN",
             "CYAN",
             "RED",
             "MAGENTA",
             "BROWN",
             "LGHT_BLUE",
             "LGHT_GREEN",
             "LGHT_CYAN",
             "LGHT_RED",
             "YELLOW",
```

```
             "LGHT_MAGENTA"};

/*-----------------------------------------------------------------*/

main(argc, argv)
   int argc;
   char *argv[];
   {PG_device *SCR_dev;
    PG_device *SCR_dew;
    REAL y, dy;
    int i, n;

    SCR_dev = PG_make_device("WINDOW", "COLOR", "PGS Test A");
    SCR_dew = PG_make_device("WINDOW", "COLOR", "PGS Test B");

    PG_set_viewport_pos(SCR_dev, 0.0001, 0.0001);
    PG_set_viewport_shape(SCR_dev, 0.9999, 0.0, 1.0);

    PG_set_viewport_pos(SCR_dew, 0.0001, 0.0001);
    PG_set_viewport_shape(SCR_dew, 0.9999, 0.0, 1.0);

    PG_white_background(SCR_dev, TRUE);
    PG_white_background(SCR_dew, FALSE);

    PG_open_device(SCR_dev, 0.1, 0.1, 0.4, 0.4);
    PG_open_device(SCR_dew, 0.5, 0.1, 0.4, 0.4);


/* connect the I/O functions */
    PG_open_console("PGS Test", "MONOCHROME", 0.1, 0.7, 0.5, 0.3);

    PG_set_viewport(SCR_dev, 0.0, 1.0, 0.0, 1.0);
```

```
    PG_set_window(SCR_dev, 0.0, 1.0, 0.0, 1.0);
    PG_set_viewport(SCR_dew, 0.0, 1.0, 0.0, 1.0);
    PG_set_window(SCR_dew, 0.0, 1.0, 0.0, 1.0);

    n  = 16;
    dy = 1.0/(n + 1.0);
    y  = 0.5*dy;
    for (i = 0; i < n; i++)
      {PG_set_line_color(SCR_dev, i);
       PG_draw_line(SCR_dev, 0.0, y, 0.5, y);
       PG_set_text_color(SCR_dev, i);
       PG_write_abs(SCR_dev, 0.6, y, "%d %s", i, color[i]);

       PG_set_line_color(SCR_dew, i);
       PG_draw_line(SCR_dew, 0.0, y, 0.5, y);
       PG_set_text_color(SCR_dew, i);
       PG_write_abs(SCR_dew, 0.6, y, "%d %s", i, color[i]);

       y += dy;};

    PG_update_vs(SCR_dew);
    PG_update_vs(SCR_dev);

    SC_pause();

    PG_close_device(SCR_dew);
    PG_close_device(SCR_dev);

    exit(0);}

/*--------------------------------------------------------------------*/
```

```
/*--------------------------------------------------------------------*/
```

Line Plots

This program demonstrates some of the line plot capabilities of PGS.

```
#include "pgs.h"

#define N 50

/*--------------------------------------------------------------------*/

main(argc, argv)
   int argc;
   char **argv;
   {int i;
    REAL *x, *y;
    char *s;
    PG_graph *data;
    PG_device *SCR_dev, *SCR_dew, *PS_dev;

    s = SC_strsave("WINDOW");
    for (i = 1; i < argc; i++)
        if (argv[i][0] == '-')
           {switch (argv[i][1])
               {case 'h' : i++;
                           s = argv[i];
                           break;};};

    x = MAKE_N(REAL, N);
```

```
    y = MAKE_N(REAL, N);

    for (i = 0; i < N; i++)
        {x[i] = -(i+1)/8.0;
         y[i] = 6.022e23/x[i];};

    data = PG_make_graph_1d('A', "Test Data #1", FALSE, N,
                            x, y, "X Values", "Y Values", 1, 0.0, 1);

    PG_open_console("GSTEST", "MONOCHROME", 1,
                    0.05, 0.7, 0.9, 0.20);

/* set up the left window */
    SCR_dev = PG_make_device(s, "COLOR", "PGS Test A");
    PG_white_background(SCR_dev, TRUE);
    PG_turn_data_id(SCR_dev, ON);
    PG_open_device(SCR_dev, 0.05, 0.2, 0.45, 0.45);

/* set up the right window */
    SCR_dew = PG_make_device(s, "COLOR", "PGS Test B");
    PG_white_background(SCR_dew, FALSE);
    PG_turn_data_id(SCR_dew, ON);
    PG_open_device(SCR_dew, 0.5, 0.2, 0.45, 0.45);

/* set up the hard copy device */
    PS_dev = PG_make_device("PS", "MONOCHROME", "PGS Test");
    PG_turn_data_id(PS_dev, ON);
    PG_open_device(PS_dev, 0.0, 0.0, 0.0, 0.0);

    PG_set_plot_type(SCR_dev, CARTESIAN);
    PG_draw_graph(SCR_dev, data);
```

```
PG_set_plot_type(PS_dev, CARTESIAN);
PG_draw_graph(PS_dev, data);
SC_pause();

PG_set_plot_type(SCR_dew, POLAR);
PG_draw_graph(SCR_dew, data);
PG_set_plot_type(PS_dev, POLAR);
PG_draw_graph(PS_dev, data);
SC_pause();

PG_set_plot_type(SCR_dev, INSEL);
PG_draw_graph(SCR_dev, data);
PG_set_plot_type(PS_dev, INSEL);
PG_draw_graph(PS_dev, data);
SC_pause();

for (i = 0; i < N; i++)
    {x[i] = i/8.0;
     y[i] = cos(x[i]);};

data = PG_make_graph_1d('B', "Test Data #2", FALSE, N,
                        x, y, "X Values", "Y Values", 2, 0.0, 1);

PG_turn_grid(SCR_dev, ON);
PG_turn_grid(SCR_dew, ON);
PG_turn_grid(PS_dev, ON);

PG_set_plot_type(SCR_dew, POLAR);
PG_draw_graph(SCR_dew, data);
PG_set_plot_type(PS_dev, POLAR);
```

```
    PG_draw_graph(PS_dev, data);
    SC_pause();

    PG_set_plot_type(SCR_dev, INSEL);
    PG_draw_graph(SCR_dev, data);
    PG_set_plot_type(PS_dev, INSEL);
    PG_draw_graph(PS_dev, data);
    SC_pause();

    PG_set_plot_type(SCR_dew, CARTESIAN);
    PG_draw_graph(SCR_dew, data);
    PG_set_plot_type(PS_dev, CARTESIAN);
    PG_draw_graph(PS_dev, data);
    SC_pause();

    PG_close_device(SCR_dev);
    PG_close_device(SCR_dew);
    PG_close_device(PS_dev);

    exit(0);}

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/
```

Contour Plots

This program illustrates the PGS contour plotting functionality.

```
#include "pgs.h"

#define NPLOTS 1
```

```
PG_graph
 *dataset;

/*--------------------------------------------------------------------*/

/* MAIN - start it off here */

main()
   {int i, k, l, kmax, lmax, kxl;
    int id, lncolor, lnstyle, centering;
    REAL *x, *y, *f, r, t, lnwidth;
    REAL xmin, xmax, ymin, ymax, fmin, fmax;
    PG_device *SCR_dev, *PS_dev;

    dataset  = MAKE_N(PG_graph, NPLOTS);
    mark     = MAKE_N(int, NPLOTS);

/* set up the graphics window */
    SCR_dev = PG_make_device("WINDOW", "COLOR", "PGS Contour Test");
    PG_open_device(SCR_dev, 0.05, 0.2, 0.45, 0.45);

    PS_dev  = PG_make_device("PS", "MONOCHROME", "PGS Contour Test");
    PG_open_device(PS_dev, 0.0, 0.0, 0.0, 0.0);

    PG_set_viewport_pos(SCR_dev, 0.25, 0.15);
    PG_set_viewport_shape(SCR_dev, 0.5, 0.0, 0.5/0.3333);

    PG_white_background(SCR_dev, TRUE);
    PG_set_border_width(SCR_dev, 5);
```

```
/* set up data */
    kmax      = 20;
    lmax      = 20;
    xmin      = -5.0;
    xmax      = 5.0;
    ymin      = -5.0;
    ymax      = 5.0;
    kxl       = kmax*lmax;
    x         = MAKE_N(REAL, kxl);
    y         = MAKE_N(REAL, kxl);
    f         = MAKE_N(REAL, kxl);
    id        = 'A';
    lncolor   = SCR_dev->BLUE;
    lnwidth   = 0.0;
    lnstyle   = SOLID;
    centering = FALSE;

    for (k = 0; k < kmax; k++)
        for (l = 0; l < lmax; l++)
            {i = l*kmax + k;
             x[i] = k/10.0 - 5.0;
             y[i] = l/10.0 - 5.0;
             r = x[i]*x[i] + y[i]*y[i];
             t = 5.0*atan(y[i]/(x[i] + SMALL));
             r = pow(r, 0.125);
             f[i] = exp(-r)*(1.0 + 0.1*cos(t));};

    PG_iso_limit(f, kxl, &fmin, &fmax);

    dataset = PG_make_graph_r2_r1(id, "contour", FALSE,
                                  kmax, lmax, centering, x, y, f,
```

```
                                     "xy", "f");

    data->info = PG_set_tds_info(data->info, PLOT_CONTOUR, CARTESIAN,
                                 lnstyle, lncolor, 10, 1.0,
                                 lnwidth, 0.0, 0.0, 0.0, HUGE);

    PG_contour_plot(SCR_dev, dataset);
    PG_contour_plot(PS_dev, dataset);

    SC_pause();

    PG_close_device(SCR_dev);
    PG_close_device(PS_dev);

    exit(0);}

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/
```

Image Plots

PGS has facilities for manipulating cell array data and rendering such data sets. In PGS these are referred to as image plots. The following program generates and plots some images.

```
#include "pgs.h"

/*--------------------------------------------------------------------*/

main()
   {int k, l;
```

```
int X_max, Y_max;
double dx, dy, x, y, sf;
char *s;
PG_palette *pl;
PG_image *calc_im;
PG_device *SCR_dev, *SCR_dew, *PS_dev;
unsigned char *bf, *p;

X_max  = 250;
Y_max  = 250;

calc_im = PG_make_image("Test Image", SC_CHAR_S,
                        0.0, 0.0, 0.0, 0.0, -1.0, 1.0,
                        X_max, Y_max, 4, NULL);
if (calc_im == NULL)
   {PRINT(stdout, "CAN'T ALLOCATE IMAGE");
    exit(1);};

PG_open_console("GSIMTS", "MONOCHROME", 1,
                0.05, 0.7, 0.9, 0.20);

SCR_dev = PG_make_device("WINDOW", "COLOR", "PGS Image Test");
PG_background_white(SCR_dev, FALSE;
PG_open_device(SCR_dev, 0.05, 0.1, 0.4, 0.4);

SCR_dew = PG_make_device(s, "COLOR", "PGS Image Test");
PG_background_white(SCR_dew, TRUE);
PG_open_device(SCR_dew, 0.55, 0.1, 0.4, 0.4);

PS_dev  = PG_make_device("PS", "COLOR", "PGS Test");
```

```
    PG_open_device(PS_dev, 0.0, 0.0, 0.0, 0.0);

    PG_set_viewport(SCR_dev, 0.0, 1.0, 0.0, 1.0);
    PG_set_window(SCR_dev, 0.0, 1.0, 0.0, 1.0);

    PG_set_viewport(SCR_dew, 0.0, 1.0, 0.0, 1.0);
    PG_set_window(SCR_dew, 0.0, 1.0, 0.0, 1.0);

/* draw the first image */
    bf = calc_im->buffer;
    pl = PG_set_palette(SCR_dev, "bw");
    sf = pl->n_pal_colors;
    for (l = 0; l < Y_max; l++)
        {for (k = 0; k < X_max; k++)
             *bf++ = sf*((double) k)*((double) (Y_max - l - 1))/
                            ((double) Y_max*X_max);};

    PG_draw_image(SCR_dev, calc_im, "Test Data A");

    PG_set_palette(PS_dev, "bw");
    PG_draw_image(PS_dev, calc_im, "Test Data HC");

    SC_pause();

/* draw the second image */
    dx = 2.0*PI/((double) X_max);
    dy = 2.0*PI/((double) Y_max);
    bf = calc_im->buffer;
    pl = PG_set_palette(SCR_dew, "rainbow");
    sf = pl->n_pal_colors;
    for (l = 0; l < Y_max; l++)
```

```
        {for (k = 0; k < X_max; k++)
             {x = ((double) k)*dx;
              y = ((double) (Y_max - l - 1))*dy;
             *bf++ = sf*(0.5 + 0.5*sin(x)*cos(y));};};

    PG_clear_window(SCR_dew);
    PG_draw_image(SCR_dew, calc_im, "Test Data B");

    SC_pause();

    PG_close_device(SCR_dev);
    PG_close_device(SCR_dew);
    PG_close_device(PS_dev);

    PG_rl_image(calc_im);

    exit(0);}

/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
```

Surface Plots

PGS also has the ability to render data sets as wire-frame meshes with hidden line removal. The following program shows how this is done.

```
#include "pgs.h"

#define N_POINTS 10

static void
```

```
 DECLARE(draw_set, (PG_device *SCR_dev, PG_device *PS_dev,
                    PG_device *CGM_dev, REAL *rz, REAL *rs, REAL *ext,
                    REAL *rx, REAL *ry, int n_pts,
                    double xmn, double xmx, double ymn, double ymx,
                    double theta, double phi, double width,
                    int color, int style, int type, int *maxes)),
 DECLARE(print_help, (byte));

/*------------------------------------------------------------------*/

main(argc, argv)
   int argc;
   char **argv;
   {int i, k, l;
    int maxes[2], X_max, Y_max, n_pts, use_color, type;
    double x, y, dx, dy, theta, phi, dp;
    char *s, t[MAXLINE], *token;
    PG_device *SCR_dev, *PS_dev, *CGM_dev;
    REAL *rx, *ry, *rz, *rs, *px, *py, *pz, *ps, *ext;

    X_max     = N_POINTS;
    Y_max     = N_POINTS;
    use_color = FALSE;
    type      = PLOT_WIRE_MESH;
    dp        = 0.0;
    for (i = 1; i < argc; i++)
        {if (argv[i][0] == '-')
            {switch (argv[i][1])
                {case 'c' : use_color = TRUE;
                            break;
                 case 'p' : dp = ATOF(argv[++i]);
```

```
                        break;
                 case ‘s’ : type = PLOT_SURFACE;
                            break;
                 case ‘w’ : type = PLOT_WIRE_MESH;
                            break;
                 case ‘z’ : X_max = Y_max = atoi(argv[++i]);
                            break;};}

         else
            break;};

    maxes[0] = X_max;
    maxes[1] = Y_max;
    n_pts    = X_max*Y_max;

    PG_open_console("GSSFTS", "MONOCHROME", 1,
                    0.55, 0.1, 0.4, 0.8);

    if (use_color)
       SCR_dev = PG_make_device("WINDOW", "COLOR", "PGS Surface Test");
    else
       SCR_dev = PG_make_device("WINDOW", "MONOCHROME",
                                "PGS Surface Test");
    PG_open_device(SCR_dev, 0.1, 0.1, 0.4, 0.4);

    if (type == PLOT_SURFACE)
       PG_set_palette(SCR_dev, "spectrum");

    rx = px = MAKE_N(REAL, n_pts);
    ry = py = MAKE_N(REAL, n_pts);
```

```
    rz = pz = MAKE_N(REAL, n_pts);

    if (type == PLOT_SURFACE)
       rs = ps = MAKE_N(REAL, n_pts);
    else
       rs = ps = pz;

    ext = MAKE_N(REAL, 4);
    ext[0] = -1;
    ext[1] =  1;
    ext[2] = -1;
    ext[3] =  1;

/* draw the first image */
    dx = 2.0*PI/((double) (X_max - 1));
    dy = 2.0*PI/((double) (Y_max - 1));
    for (l = 0; l < Y_max; l++)
        {for (k = 0; k < X_max; k++)
             {*px++ = x = ((double) k)*dx;
              *py++ = y = ((double) l)*dy;
              *pz++ = cos(y);
              if (type == PLOT_SURFACE)
                 *ps++ = sin(x);};};

    if (dp == 0.0)
       {while (TRUE)
           {PRINT(stdout, "Viewing Angle: ");
            GETLN(t, MAXLINE, stdin);

            if ((token = strtok(t, " ,")) == NULL)
               break;
```

```
            theta = DEG_RAD*ATOF(token);

            if ((token = strtok(NULL, " ,")) == NULL)
               break;
            phi = DEG_RAD*ATOF(token);

            draw_set(SCR_dev, PS_dev, CGM_dev,
                     rz, rs, ext, rx, ry,
                     n_pts,
                     0.0, 2.0*PI, 0.0, 2.0*PI,
                     theta, phi,
                     0.0, SCR_dev->BLUE, SOLID,
                     type, maxes);};}

    else
       {for (phi = 0.0; phi <= 90.0; phi += dp)
            {draw_set(SCR_dev, PS_dev, CGM_dev,
                      rz, rs, ext, rx, ry,
                      n_pts,
                      0.0, 2.0*PI, 0.0, 2.0*PI,
                      0.0, DEG_RAD*phi,
                      0.0, SCR_dev->BLUE, SOLID,
                      type, maxes);};};

    PG_close_device(SCR_dev);

    exit(0);}

/*--------------------------------------------------------------------*/

/* DRAW_SET - draw a complete set of plots */
```

```
static void draw_set(SCR_dev, PS_dev, CGM_dev, rz, rs, ext, rx, ry,
                     n_pts, xmn, xmx, ymn, ymx, theta, phi,
                     width, color, style, type, maxes)
   PG_device *SCR_dev, *PS_dev, *CGM_dev;
   REAL *rz, *rs, *ext, *rx, *ry;
   int n_pts;
   double xmn, xmx, ymn, ymx, theta, phi, width;
   int color, style, type;
   int *maxes;
   {PG_draw_surface(SCR_dev, rz, rs, ext, rx, ry,
                    n_pts,
                    xmn, xmx, ymn, ymx, theta, phi, width, color,
                    style, type, "Test Data",
                    "Logical-Rectangular", maxes);
    return;}

/*------------------------------------------------------------------*/
/*------------------------------------------------------------------*/
```

Vector Plots

PGS can render 2 dimensional vector fields. The following program demonstrates this capability.

FORTRAN API Example

```
c
c GSTEST.F - test of PGS FORTRAN API
c
c --------------------------------------------------------------------
```

```
      program gsftst

      integer pgmkdv
      integer idev, idvp
      double precision x1, y1, x2, y2
      double precision x(5), y(5), u(4), v(4)
      character*8 names, namep
      character*10 name2, name3
      character*12 name4

c ... set up the window
      names = 'WINDOW'
      name2 = 'COLOR'
      name3 = 'PGS Test A'
      idev = pgmkdv(6, names, 5, name2, 10, name3)
      x1 = 0.05
      y1 = 0.2
      x2 = 0.45
      y2 = 0.45
      call pgopen(idev, x1, y1, x2, y2)

c ... set up the PS device
      namep = 'PS'
      name2 = 'MONOCHROME'
      name3 = 'gsftst'
      idvp = pgmkdv(2, namep, 10, name2, 6, name3)
      x1 = 0.0
      y1 = 0.0
      x2 = 0.0
      y2 = 0.0
      call pgopen(idvp, x1, y1, x2, y2)
```

```
      call pgclsc(idev)
      call pgclsc(idvp)

c ... set up the view port and world coordinate system
      x1 = 0.1
      x2 = 0.9
      y1 = 0.1
      y2 = 0.9
      call pgsvwp(idev, x1, x2, y1, y2)
      call pgsvwp(idvp, x1, x2, y1, y2)
      x1 =   0.0
      x2 =  10.0
      y1 = -15.0
      y2 =  30.0
      call pgswcs(idev, x1, x2, y1, y2)
      call pgswcs(idvp, x1, x2, y1, y2)

c ... draw a bounding box
      x1 =   0.0
      x2 =  10.0
      y1 = -15.0
      y2 =  30.0
      call pgdrbx(idev, x1, x2, y1, y2)
      call pgdrbx(idvp, x1, x2, y1, y2)

c ... write a string
      x1 = 5.0
      y1 = 0.0
      name4 = 'TEXT STRING'
```

```
 100  format(1p, e10.2)
      call pgwrta(idev, x1, y1, 11, name4)
      call pgwrta(idvp, x1, y1, 11, name4)

c ... draw a line
      x1 =  1.0
      x2 =  9.0
      y1 = -4.0
      y2 = -1.0
      call pgdrln(idev, x1, y1, x2, y2)
      call pgdrln(idvp, x1, y1, x2, y2)

c ... do a vector plot
      x(1) =  3.0
      y(1) =  4.0
      u(1) = -0.5
      v(1) = -0.25

      x(2) =  4.0
      y(2) =  4.0
      u(2) =  0.5
      v(2) = -0.25

      x(3) = 4.0
      y(3) = 5.0
      u(3) = 0.5
      v(3) = 0.0

      x(4) =  3.0
      y(4) =  5.0
      u(4) = -0.5
```

```
      v(4) =  0.5

      call pgplvc(idev, x, y, u, v, 4)
      call pgplvc(idvp, x, y, u, v, 4)

c ... draw and fill a polygon (color 4 is blue)
      x(1) = 5.0
      x(2) = 6.0
      x(3) = 6.0
      x(4) = 5.0
      x(5) = 5.0
      y(1) = 8.0
      y(2) = 8.0
      y(3) = 9.0
      y(4) = 9.0
      y(5) = 8.0
      call pgfply(idev, x, y, 5, 4)
      call pgfply(idvp, x, y, 5, 4)

      call pgfnpl(idev)
      call pgfnpl(idvp)

      pause

c ... close the device
      call pgclos(idev)
      call pgclos(idvp)

      call exit
      end
```

c --

Other PACT Documentation

PGS depends on the SCORE and PML PACT libraries for certain key supporting functionalities. In turn PGS structures are used in ULTRA II, SX, and PANACEA. Some readers may find it helpful to refer to these and other PACT documents.

The list of PACT Documents is:

PACT User's Guide	UCRL-MA-112087	
SCORE User's Manual	UCRL-MA-108976 Rev.1	
PPC User's Manual	UCRL-MA-108964 Rev.1	
PML User's Manual	UCRL-MA-108965 Rev.1	
PDBLib User's Manual	M-270 Rev.2	
PGS User's Manual	UCRL-MA-108966 Rev.1	Current Document
PANACEA User's Manual	M-276 Rev.2	
ULTRA II User's Manual	UCRL-MA-108967 Rev.1	
PDBDiff User's Manual	UCRL-MA-108975 Rev.1	
PDBView User's Manual	UCRL-MA-108968 Rev.1	
SX User's Manual	UCRL-MA-112315	

Send questions and comments to rodrigues2@llnl.gov