Solaris VISUAL Stencil/Paint Interfaces 5

Stencil/Paint Interfaces

The Solaris VISUAL environment addresses the needs of graphic arts and related document imaging applications and APIs through the PostScript foundation library, the Display PostScript extension protocol and DPSlib X Extension API (when available from the X Consortium).

PostScript employs a stencil/paint approach to graphics using the conceptual model of a two-dimensional rendering space. Images are built on the display (or page in PostScript) by placing paint in selected areas. The paint may be in the form of lines, shapes, images or text, and may vary in color. The paint on the page is always opaque, so that the last mark on the page overlaps any previous ones drawn.

PostScript has no notion of pixels, but rather utilizes world coordinates for drawing. The effect of the higher-level abstraction from device to world coordinates has the benefit of enabling the same image displayed on both high and low resolution devices, to vary in quality and not physical size. As screen resolution increases, image fidelity will also improve, and it will do so without the necessity to modify the application code.

The stencil/paint model appeals to a number of applications areas, most appropriately to users where the visual paradigm of move-draw is attractive.

Stencil/Paint Applications

The Stencil/Paint rendering model is employed in a number of graphics interfaces today. However its most common usage is in the PostScript interface. As a well-established standard for printing to a large variety of devices, PostScript is commonly used in applications where hard-copy display is important. With the ability to effectively provide equivalent screen display and hard-copy output, PostScript addresses a range of applications ranging from graphic arts to color prepress to desktop publishing.

Color Prepress

Color prepress applications rely upon the ability to apply a range of rendering manipulation to the data as well as color separation to images. The ability to effectively match display and output representation of images enables PostScript to provide a suitable platform for many applications.

Graphic Arts

Applications addressing the graphic arts market use PostScript for manipulation of text and graphics such as is required for advertising, layout and production.

Business Graphics

PostScript is already well established in many desktop markets as the preferred interface for printing and presentation graphics. Its excellent support for a true WYSIWYG model, together with broad support for resolution-independent output makes PostScript a good choice for business graphics applications.

The PostScript Interpreter and Foundation Library

The PostScript page description language was developed by Adobe Systems, for use in specifying the visual appearance of printed documents. Since PostScript's inception, it has evolved under the name Display PostScript (DPS), to drive widely varied-resolution displays and to coexist with the X Window System. PostScript has also evolved from Level 1 to Level 2 by the addition of features designed to enhance the breadth of functionality and the rendering performance.

In SunSoft's Solaris VISUAL graphics environment[1], PostScript functionality is provided through the DPSlib API. This API can accept ASCII PostScript directly or can be called programmatically to do stencil/paint rendering. Since DPS is an X extension that is available within the OpenWindows server, and on other vendors platforms, rendering over the network is supported.

PostScript is implemented under Solaris VISUAL as an interpretive language, in much the same way as it is used in printing. Applications are typically written in the C language, and call a client library for access to the PostScript intepreter.

The client library contains data structures and procedures used by the application to send requests to the PostScript interpreter. Applications generally perform calculations, manage window system activities and handle I/O independent of PostScript. The application developer uses PostScript by calling predefined PostScript procedures (called wraps) or by producing new ones. PostScript code is passed through a translator which converts it into C callable code which is included in the application. Figure 5.1 illustrates the use of PostScript.

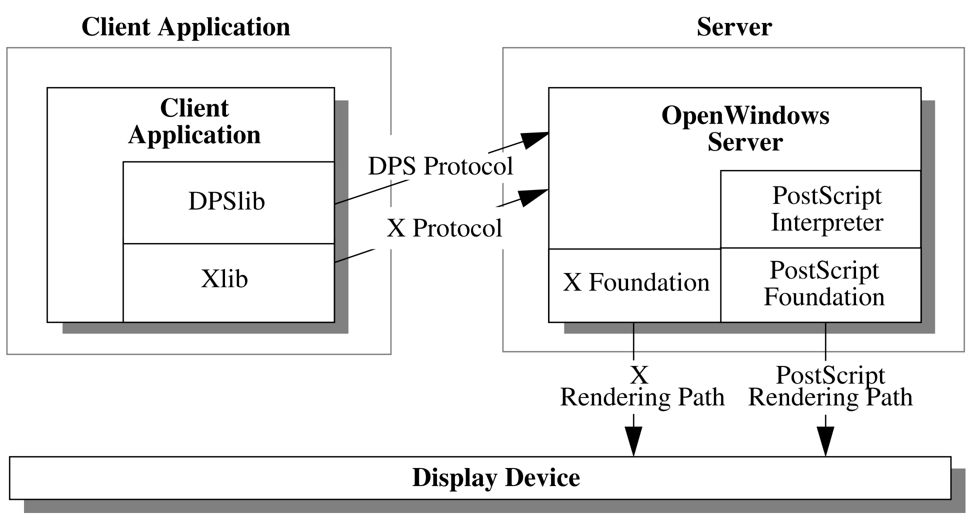


Figure 5.1 Solaris VISUAL PostScript Foundation Library and Interpreter

1. This chapter describes the implementation of Postscript offered in a release of Solaris following Solaris 2.2. Solaris 2.2 utilizes the PostScript implementation offered by the NeWS API, which is PostScript Level 1 and does not offer the DPS API.

From the standpoint of graphics, PostScript provides a minimalist 2-D *move-draw* interface. The number of PostScript primitives directly related to rendering is small. However higher-level primitives can be 'built' from simpler ones. PostScript employs the concept of a *page*, onto which the application places the data to be rendered. A *path* can be used to describe arbitrary regions in 2-D coordinate space, generally representing a sequence of lines and arcs, which ultimately describe fonts and generalized graphics. Figure 5.2 illustrates the Solaris VISUAL Stencil/Paint interfaces.

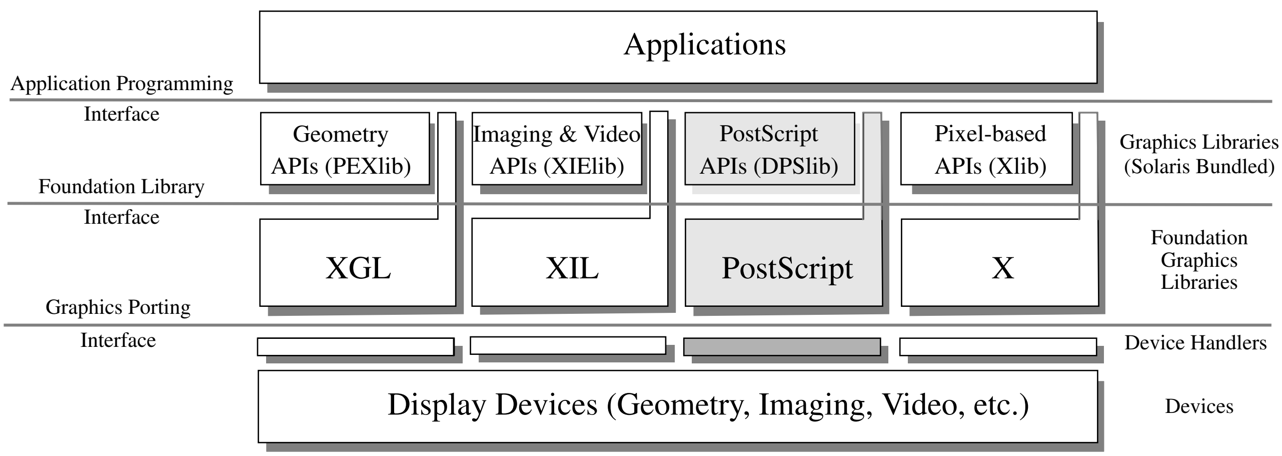


Figure 5.2 Solaris VISUAL PostScript Foundation Library and Layered Interfaces

PostScript Language Architecture

PostScript makes extensive use of *stacks*; locations where objects may be temporarily stored. These stacks are the basis for all operations in PostScript, from variable definitions to control statements.

PostScript supports four separate stacks: the *operand stack*, for the storage of data, the *execution stack* for commands, the *dictionary stack* for dictionary objects and the *graphic state stack* for graphics context settings.

In PostScript, programs are simply sequences of objects, which are parsed into individual tokens. PostScript objects are executed according to their type: upon encountering an object, the object is checked to determine whether it is executable. If the object is executable, it is looked up in the dictionary stack to determine if it is a keyword and has an associated executable operator.

Depending upon the type of object, it is placed onto the operand or execution stack, until it is explicitly executed.

PostScript programs are simply procedures built out of executable arrays. A procedure is executed when a keyword is looked up during the process of executing a sequence of objects, and the value for the keyword found is an executable array. An array is simply a linear group of objects, of one or more data types.

PostScript operators act upon numbers and other data, which has been previously placed on the operand stack. The execution causes the applicable objects to be retrieved from the stack, and the result is placed back on the stack. PostScript employs a language style of declaring the operands first, followed by the operator. This approach is known as *postfix notation*. As objects are interpreted by PostScript, they are 'looked up' in the dictionaries, to determine if they are operator names. When the PostScript interpreter encounters an object in its dictionaries, it executes the instructions associated with it. Figure 5.3 illustrates the rendering operator specific subset of the PostScript language.

Dictionaries are tables which associate pairs of objects. Like its linguistic equivalent, which associates words with their definitions, PostScript dictionaries relate objects called *keys* with different objects, called *values*. Values can be simple or complex - even entire programs can be defined as values. The PostScript dictionaries have a special significance because code segments which are stored in arrays may be referenced via keys stored in the dictionaries. This special ability to dynamically extend the language makes PostScript an extremely powerful environment.

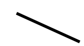
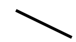
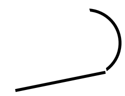
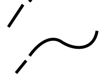
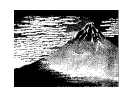
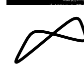
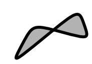
Primitive Call	Description	Example
moveto	Move to current point	
rmoveto	Move relative to current point	
lineto	Add line to the current path	
rlineto	Draw line relative current path	
arcto	Add a circular arc to the current path	
arc	Add a circular arc to current path	
curveto	Adds Bezier curve to the current path	
rcurveto	Adds Bezier curve relative to current path	
image	Renders image onto current page	
stroke	Draws a line through the current path	
fill	Fill current path with a color	

Figure 5.3 PostScript Graphics Operators

Fonts

Text fonts are an extremely important part of PostScript. The OpenWindows server supports the Adobe Character Bitmap Distribution Format (bdf), for access to bitmap fonts. While X Windows supports different facilities for conversion of bdf fonts into displayable characters, the same set of fonts is accessible to both Xlib-based applications and PostScript.

Fonts in X are sets of bitmaps, representing elements in the character set. Every X function which draws text has two versions: one for single-byte fonts, and another for two-byte fonts. The server may be queried for information on available fonts. Applications use the response from the server to determine which fonts may be loaded and displayed.

OpenWindows greatly extends the font facilities of X Windows with the addition of infinitely scalable fonts known as OpenFonts. OpenWindows provides a core set of 57 *F3*™ format typefaces. F3 fonts represent outline fonts, which in addition to carrying geometric information, maintain a set of 'hints', used to insure maximum quality of the text when scaled at any size. Because F3 fonts can be arbitrarily scaled to produce bitmaps of any size, a single font can replace a whole family of bitmap files.

OpenWindows supports Adobe Type 1 format fonts as another scalable font format. The combination of Type 1 format fonts and OpenFonts provide an extremely wide choice of fonts.

The DPS Extension to the X Protocol

Display PostScript was developed by Adobe Systems to provide a means of rendering images and other information on the screen in a fashion compatible with the PostScript language used with printers. Display PostScript's device-independent imaging model is intended to free the application developer from worrying about how information is displayed on the screen.

In SunSoft's Solaris VISUAL Graphics Environment, PostScript is provided as a foundation graphics interface, fully integrated with OpenWindows. Display PostScript is available as an API which utilizes the PostScript foundation library. The facilities of Display PostScript include true interoperability with non-Solaris platforms, as well as distributed display across the network with the DPS protocol extension to the OpenWindows X server (and other X servers which have the DPS extension).

Third-Party Stencil/Paint Interfaces

Third-party APIs written to use the PostScript and Display PostScript interfaces under Solaris will have the same access to PostScript rendering features and remote rendering capabilities as Display PostScript. The use of individual PostScript operators and functionality is dependent upon how much of Display PostScript that the third-party API makes available to the application developer.