# Controlling Graphics Devices for Statistical Graphics

*Allan R. Wilks*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*created: 18 August 1995*

*ABSTRACT*

Pictor* is an environment for statistical graphics that promotes simple commands for common uses, while also offering the ability to experiment with whole new paradigms. Its design comprises several levels, from its high-level graphics model to its low-level device model.

In Pictor, high-level S graphics functions produce graphics objects, or *grobs*, which, when printed, are turned into a sequence of simple instructions for a particular graphics device. The code that implements the last step is the *device driver*. Each device driver in Pictor responds to the same set of commands, though of course it produces output individualized to a particular device, such as a PostScript printer or a Motif window.

The set of commands to which the device drivers respond comprise Pictor's *low-level graphics model*; it is the purpose of this paper to describe that model.

## 1. Introduction

Pictor is an environment for statistical graphics that promotes simple commands for common uses, while also offering the ability to experiment with whole new paradigms. Its design comprises several levels, from its high-level graphics model to its low-level device model.

In Pictor, high-level S graphics functions produce graphics objects, or *grobs*, which, when printed, are turned into a sequence of simple instructions for a particular graphics device. The code that implements the last step is the *device driver*. Each device driver in Pictor responds to the same set of commands, though of course it produces output individualized to a particular device, such as a PostScript printer or a Motif window.

The set of commands to which the device drivers respond comprise Pictor's *low-level graphics model*; it is the purpose of this paper to describe that model. In the next

*Lacaille, one of the first astronomers systematically to chart the southern skies, gave the name *Equuleus Pictoris*, meaning *Painter's Easel*, to the grouping of stars between declinations -43° and -64° and right ascensions $4^h 30^m$ and $6^h 50^m$. Today the name of this constellation has been shortened to simply *Pictor*, meaning *Painter* and can be found just north of the Large Magellanic Cloud.

section, we describe each of the commands of Pictor's low-level model. Then in section 3 we describe a working implementation of the model in the S language. In section 4 is a comparison of Pictor's low-level graphics model with the corresponding part of the model S has used for graphics in the past.

A final section is devoted to a brief roadmap of the rest of Pictor. For more detailed information there are two other documents: Wilks (1994a) describes the constraint-based mechanism that Pictor uses to describe graphs in a natural way and Wilks (1994b) gives an overview of the S Version 4 (Chambers 1994) implementation of Pictor from the programmer's viewpoint.

## 2. Elements of the low-level graphics model

For each piece of a statistical graph, one must specify *what* is to be drawn, and *where* it should go on the display (page or screen). In this section we will see how Pictor achieves these two objectives in its low-level graphics model. We first describe the coordinate systems used to identify positions on the display. We then say what kinds of *drawing primitives* Pictor can represent, and then how the visual appearance of these objects is parameterized for actual rendering of them on the display.

It is also possible to issue several sorts of queries in Pictor. We will present the details of each of the possible queries and then end this section with a description of the commands for general control of the device driver.

### 2.1. Coordinate systems

Drawing primitives always have positional information associated with them, in the form of coordinate pairs. For example, a line segment requires two coordinate pairs to define its endpoints. (The complete set of drawing primitives is given in the next subsection.) One of the main design ideas in Pictor is that each drawing primitive's positional information is given in a coordinate system intrinsic to the primitive. This means that in order to render a primitive onto the display, a *coordinate transformation* must be given that specifies the relationship between the primitive's intrinsic coordinate system and that of the device.

In addition to these two coordinate systems, Pictor adds a third, in order to insulate the low-level graphics model from the device-specific device coordinate system. This intermediate coordinate system is *absolute* in the following sense. Each display (page, screen window) is assumed to be rectangular. The origin of the absolute coordinate system is the lower-left corner of the display with *x* increasing to the left and *y* increasing vertically. The units in both directions are inches.

In the low-level graphics model, it is necessary to specify the coordinate transformation between the intrinsic coordinate system of a primitive and the absolute coordinate system. The device driver knows the transformation from the absolute system to its own private system. Concatenation of these two transformations happens automatically just prior to rendering the primitive.

The coordinate system of a primitive can bear an arbitrary affine relationship to the absolute coordinate system. If $P_i = (x_i, y_i)$ are the (intrinsic) coordinates of a position in a primitive and if the absolute coordinates of this position on the display are $P_a = (x_a, y_a)$, we may write the relationship between $P_i$ and $P_a$ in this way:

$$x_a = ax_i + by_i + t_x$$

$$y_a = cx_i + dy_i + t_y$$

This may be written in homogeneous coordinates as:

$$\begin{bmatrix} x_a \\ y_a \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}.$$

The six-element vector $(a, b, c, d, t_x, t_y)$ completely specifies the coordinate transformation. This scheme is quite similar to the way PostScript (Adobe 1985) handles coordinate transformations. Note, though, that the roles of $b$ and $c$ are reversed in the above equations, which has the effect of treating (homogeneous) coordinates as column vectors that postmultiply the transformation matrix rather than row vectors that premultiply it, as in PostScript.

Traditional systems for doing statistical graphics (S version 3 being a case in point) make some assumptions about a global coordinate system related to the data at hand. It may seem somewhat burdensome, therefore, to expect a coordinate transformation to be specified before each drawing primitive, and especially in this format of a homogeneous transformation matrix, which requires careful bookkeeping to get correct. Fortunately, the other levels of Pictor that feed the low level take care of this automatically. Though we will have no more to say about this, it should be noted that the consequences are far-ranging of having the geometry for each primitive being defined in its own coordinate system. In fact, much of the character of the way Pictor works derives from this decision.

In Pictor's low-level model, there is always a *current coordinate transformation matrix* or *CTM*, in the context of which all geometric data are interpreted. The current CTM is set via the ctm primitive:

ctm: $a,\ b,\ c,\ d,\ t_x,\ t_y$

This should be thought of as a command to a device driver to set the CTM to the matrix implied by the given values. The notation intentionally does not correspond to a language such as C or S, but rather is intended to reveal the salient features of Pictor's low-level graphics model. In fact, there are implementations of each Pictor command in both S and C; we discuss the S implementation in section 3.

### 2.2. Drawing primitives

Pictor's drawing primitives are similar to those of other low-level graphics systems, like GKS, though each primitive describes not just a single object, but a whole collection of them, as in Version 3 of S. Note that if the data for any piece of a primitive contains missing values, that piece will not get drawn.

The following subsections describe Pictor's eight primitives. The actual appearance of these primitives when they get drawn is controlled by the rendering parameters and the rendering maps, which haven't been described yet. So for now we confine ourselves merely to the geometric information expected by each primitive.

### 2.2.1. Polylines

A polyline is a linear spline: a sequence of points connected by line segments. The input data, therefore, is just the sequence of points:

$$\text{polylines:} \quad (x_i, y_i), \quad i = 1, \ldots, n.$$

This sequence of points can represent one or several polylines. If there is more than one, the last coordinate pair of each polyline is separated from the first of the next polyline by a pair of missing values. Hence, $n$ is equal to the total number of points in all the polylines, plus one less than the number of polylines.

### 2.2.2. Polygons

A polygon is a polyline with the last point connected to the first. The data for the polygons primitive is the same as that for polylines:

$$\text{polygons:} \quad (x_i, y_i), \quad i = 1, \ldots, n.$$

The difference is that the last point of each polygon is implicitly connected to the first point, and the resulting figure is both filled and outlined (though either or both of these may be suppressed). As with polylines, one polygons primitive may describe several polygons; these are again separated by NA.

### 2.2.3. Segments

Segments are simply line segments with both endpoints specified:

$$\text{segments:} \quad (x_{1i}, y_{1i}), \quad (x_{2i}, y_{2i}), \quad i = 1, \ldots, n.$$

Notice that segments are a special case of polylines. Indeed, the data for segments could be supplied as data for polylines like this:

$$(x_{11}, y_{11}),\ (x_{21}, y_{21}),\ (\text{NA}, \text{NA}),\ (x_{12}, y_{12}),\ (x_{22}, y_{22}),\ (\text{NA}, \text{NA}),\ \cdots\ .$$

The segments primitive is offered because this representation is wasteful and awkward to use.

### 2.2.4. Rectangles

Rectangles are a special case of polygons:

$$\text{rectangles:} \quad (x_{1i}, y_{1i}), \quad (x_{2i}, y_{2i}), \quad i = 1, \ldots, n,$$

where the two points for each rectangle are the coordinates of two opposite corners (it doesn't matter which pair of opposite corners is given). As with polygons, rectangles can be both filled and outlined.

### 2.2.5. Arcs

Pictor can draw circular arcs:

$$\text{arcs:} \quad (c_{xi}, c_{yi}), \quad r_{i,} \quad (a_{1i}, a_{2i}), \quad i = 1, \ldots, n.$$

The $i$th arc is the part of the circle that is centered at $(c_{xi}, c_{yi})$, with radius $r_i$, that starts at angle $a_{1i}$ degrees, proceeds counter-clockwise, and ends at angle $a_{2i}$ degrees. Arcs can be filled; the interior of the arc is defined to be the union of all radii of the circle with an endpoint on the arc; thus a filled arc would be a natural way to draw the parts of a pie chart (if one were so inclined).

### 2.2.6. Text

Statistical graphs almost always have text of some sort on them; indeed, it is arguable that a graph without text is useless. However, the merging of text with the other, more geometric primitives is a difficult problem. The text primitive in Pictor takes this data:

$$\text{text}:\quad (x_i, y_i),\ \ s_{i,}\quad i = 1,...,n.$$

This means that text string $s_i$ is to be plotted at position $(x_i, y_i)$. This is necessarily somewhat imprecise, for it is not clear (yet) what is the geometric relationship between the plotted text and the given position. Note that strictly speaking the $s_i$ are not geometric data; the are more akin to sequences of marks (see the next subsection) and so should properly be part of the rendering information. However, it is much more convenient to deal with them as part of the drawing primitive, and that is what Pictor does.

### 2.2.7. Marks

One of the most common elements of statistical graphs is a set of small symbols plotted at various positions. In Pictor, these symbols are called *marks*.

$$\text{marks}:\quad (x_i, y_i),\ \ i = 1,...,n.$$

The marks are plotted at the specified positions. Later we will see how marks are defined, and in particular, as with text, what is the relationship between the mark description and position at which it is plotted. We note for now that mark definitions can be very general; virtually all the power of the low-level Pictor model can be used recursively for them.

### 2.2.8. Image

An image is a rectangular array of congruent rectangles, each filled with some color but not outlined. Each of the congruent rectangles of the image is commonly called a *pixel*. Such an array could be specified using the polygons primitive, but this would be grossly inefficient, as it would not take advantage of the rectangular arrangement of the pixels. Hence, Pictor has an image primitive. Unlike the other primitives, this one only specifies one image. Here is the format:

$$\text{image}:\quad (x_1, y_1), (x_2, y_2),\ nrow, ncol, p_1, p_2, \cdots, p_n$$

The image has $n = nrowxncol$ pixels; each $p_i$ is the value of a pixel, proceeding left-to-right top-to-bottom. The pixel is simply an integer giving an index into the colormap (see below). The two ordered pairs give the corners of the image in intrinsic coordinates, and as for rectangles they may specify either pair of corners. This is a very simple type of image and it is expected that Pictor will soon be able to deal with more complicated image data.

### 2.3. Rendering parameters

The actual rendering of Pictor's drawing primitives is controlled by *rendering parameters*. Some of these parameters are simply indexes into a table, or *rendering map*. In this subsection we describe each of the rendering parameters and, where appropriate, the corresponding rendering maps. The parameters and maps are grouped into four categories: color (affecting all drawing primitives), those affecting lines, those affecting text, and those to do with marks.

Each rendering parameter is a vector of values. The values are applied to successive parts of a drawing primitive as it is being drawn. How this happens depends on which primitive is being rendered, but is always obvious. For example, the values in the color vector are applied to successive segments in a segments primitive. For a polylines primitive the colors refer to successive polylines and not to individual segments of the polylines. If one value of a parameter is NA, the corresponding piece of the primitive will not be drawn.

#### 2.3.1. Color

Both color and fill are indexes into the color rendering map:

color: *colormap-index*

fill: *colormap-index*

Normally, all drawing primitives are rendered with the color specified by the color parameter. However, in the case of objects that can be filled, such as polygons and rectangles, the filling is done according to the fill parameter. In fact, these objects are both outlined and filled; in order just to outline, the fill parameter must be NA and just to fill, the color parameter must be NA. Index zero is the background color, which is the color of the device surface when the device driver is first invoked (and when the device is ‘‘cleared’’—see section 2.5).

Pictor uses a painting model, like PostScript. This means that if two primitives are drawn, then in places where they interfere, the one drawn last will appear. Generally this means that to erase a primitive, it can be redrawn in the background color, though this is not perfect. The index for the background color is always 0, and in general, non-negative integers are legal in all device drivers as color indexes. Of course, a particular device may not be able to render a large number of colors, but it is up to the device driver to handle this; users can define as many colors as they like.

It is not illegal for the color or fill parameters to have negative or non-integer values, but the meaning is device-dependent, and documentation for the particular device function should be consulted.

The colormap is an association between non-negative integers and color definitions. All low-level color definitions in Pictor are stored in a standard form as a triple of numbers in [0,1], representing the red, green and blue components of the color:

colordef: *red*, *green*, *blue*.

So, for example, black and white are represented by (0,0,0) and (1,1,1), and cyan by (0,1,1).

There is a mechanism in Pictor for dealing with color models other than red-green-blue, but it is outside the scope of this paper, as it is not part of the low-level model. See Wilks (1994b) for more information.

### 2.3.2. Lines

Many of the primitives involve drawing lines: polylines, polygons, arcs, rectangles, segments, marks. All line drawing is controlled by two parameters:

linestyle: *linemap-index*

linewidth: *width*

Linewidth is the thickness of lines and is given in absolute units of inches*. The linemap rendering map is an association between non-negative integers and linestyle definitions. The form of such a definition closely follows the PostScript model:

linedef: $length_1$, ..., $length_n$, *offset*

The line is drawn as a repeating sequence of dashes and gaps: first a dash of (absolute) length $length_1$, then a gap of length $length_2$, then a dash of length $length_3$, and so on. When the last piece (dash or gap) has been drawn, the cycle repeats, beginning again with a $length_1$ dash. The *offset* is the absolute distance into the total length of the sequence that drawing begins. It is possible for $n$ to be zero (no $length_i$s), in which case a solid line is drawn. So the simplest linedef is just (0), meaning a solid line. A dashed line with dashes and gaps each a tenth of an inch long is represented by (1/10, 1/10, 0).

### 2.3.3. Text

The appearance of text is controlled by four rendering parameters: font, pointsize, textrotation and textadjust. The font parameter is an index into the fontmap rendering map, which gives names of fonts:

font: *fontmap-index*

fontdef: *font-name*

The names of the available fonts will vary from device to device, but font number one will always be available as a general-purpose font.

The pointsize parameter controls the size of the text:

pointsize: *font-pointsize*

and measures the traditional notion of pointsize (we use the rounded version of 72 points per inch; the width of the letter ‘‘m’’ in points is approximately the pointsize).

---

There are three parameters that refer to sizes (linewidth, pointsize and marksize) and in each case the units are absolute. This seems to be the best compromise between ease of use and simplicity of the model. The chief inconvenience of this scheme is that to scale a picture description, the linewidth (and pointsize and marksize) specifications must also be explicitly scaled. In the PostScript model, the linewidth is given in intrinsic coordinates; this scales nicely, but is unnatural to specify and leads to visually odd results in that, for example, lines of different slopes are generically of different thicknesses.

A text string is plotted at a certain position $(x,y)$ (see the description of the text primitive above). In order to be precise about this, we first recall that each character of a font has an *origin*. Imagine moving to a position on the device at which there is a tiny dot, and then plotting a character of a given font and pointsize. The dot is then at the origin of the character. If a further character is plotted, with no explicit move in between, the second character's origin is at a distance equal to the first character's *width* from the first origin.

Now impose the following coordinate system on a given text string *s*: the origin of the coordinate system is at the origin of the first character of *s*, the *x*-axis extends in the direction the string is plotted and the *y*-axis bears the usual relationship to the *x*-axis. Further, assume that one unit in each direction is equal to *P* points, where *P* is the size of the font (the value of the pointsize parameter). The *bounding box* of the text is the smallest rectangle oriented with the axes of the string's coordinate system that exactly encloses the text.

The position of the string with respect to the point $(x,y)$ is controlled by the textadjust parameter:

textadjust: *adjustment-description*

The textadjust parameter is a pair of comma-separated words that describes where the point $(x,y)$ (the position of the string) is with respect the coordinate system described in the last paragraph. The first word names a position in the horizontal direction, according to the following table:

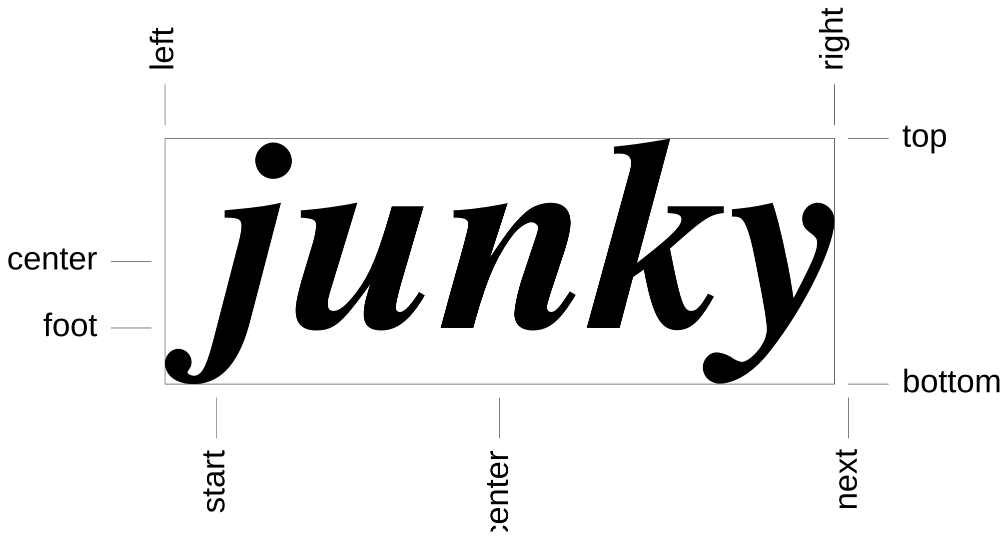
| **Word** | **Horizontal Position** |
|---|---|
| left | left edge of bounding box |
| right | right edge of bounding box |
| start | horizontal origin of first character |
| next | horizontal origin of last character plus its width |
| center | center of bounding box |

Similarly, the second word describes a position in the vertical direction:

| **Word** | **Vertical Position** |
|---|---|
| bottom | bottom edge of bounding box |
| top | top edge of bounding box |
| foot | vertical y-origin of characters |
| center | center of bounding box |

Hence, ‘‘start,foot’’ means that $(x,y)$ is at the character origin, ‘‘left,top’’ means that $(x,y)$ is at the top left corner of the bounding box, and so on. It is permissible to give just one word, in which case the other word is assumed to be ‘‘center’’. It is not necessary to supply a comma to indicate whether the missing word is for the horizontal or vertical position as all the words are different, so the given word determines which is missing. The one exception to this is ‘‘center’’ but since by the rules this represents ‘‘center,center’’ there is no ambiguity after all.

There is one additional word, ‘‘origin,’’ which is an abbreviation for ‘‘start,foot.’’ Also, any word may be replaced by an initial substring of that word, including just the first letter. Here is a diagram that illustrates the words:




A common need for text is to label positions on a graph. For example, to label the position at $(x,y)$, one could use a value of ‘‘left’’ for the textadjust parameter in order to produce a label just to the right of $(x,y)$. (This is only counterintuitive if you forget that the textadjust parameter says where $(x,y)$ is with respect to the text, not the other way around.) Unfortunately, this is usually unsatisfactory, for if there is some sort of mark at $(x,y)$, the abutting label will partially obscure it. To ameliorate this common problem, one may add any number of ‘‘+’’ signs after one of the textadjust words to indicate incremental adjustment to the right or up; similarly, ‘‘–’’ indicates movement to the left or down. The amount of adjustment for each ‘‘+’’ or ‘‘–’’ is equal to 0.5 units, or $P/2$ points (a so-called *printer’s en*). Using this notation, a better adjustment for the label in our example would be ‘‘left–’’. It is possible to get finer control over the positioning of text, though that should not usually be necessary; see the description of `dev.textsize` in section 3.

The final step in positioning a label is to rotate it with respect to the horizontal direction of the ambient coordinate system:

textrotation: *ccw-rotation-horizontal*

The rotation angle is expressed in degrees, and the origin of the rotation is the string position, $(x,y)$.

### 2.3.4. Marks

Marks are selected from the markmap rendering map with the marktype parameter:

marktype: *markmap – index*

An entry in the markmap describes one mark, and is like a little Pictor subroutine:

markdef: $spec_1$, $spec_2$, · · ·

Each $spec_i$ is a specification either of a drawing primitive or a parameter setting. The geometric data for the drawing primitives is given in a fixed coordinate system that extends from $-1$ to 1 in each of the $x$ and $y$ directions. The origin of this coordinate system will lie precisely at the point $(x,y)$ at which the mark is being plotted. Parameter settings temporarily (for the duration of the rendering of the mark) override whatever settings were in effect before the mark's drawing was begun.

The marksize parameter,

marksize: *absolute-size*,

says in absolute units how large a mark will be. It is scaled such that a distance of two units in the mark's coordinate system (from –1 to 1) is equal to marksize absolute units.

Marks can also be rotated:

markrotation: *ccw-rotation-horizontal*.

As with text, the angle is expressed in degrees. The origin of the rotation is the origin of the coordinate system, which corresponds to the point $(x,y)$—this is also consistent with the rotation of text.

As a simple example of a mark, consider the markdef consisting of the single specification:

polygons: $(-1,-1)$, $(1,-1)$, $(1,1)$, $(-1,1)$

This produces a square, centered at $(x,y)$. Whether this square is outlined or filled, and with what colors, is controlled by whatever values of color and fill are in effect at the time the mark is rendered (not at the time it is defined). To ensure that the square is always only outlined, we could include a fill command:

fill: NA

polygons: $(-1,-1)$, $(1,-1)$, $(1,1)$, $(-1,1)$

A commonly used mark is a circle. Here is the definition:

arcs: 0 0 1 0 360

### 2.4. Device queries

Everything described so far can be thought of as instructions for the graphics device. Pictor also allows the device to return information through a number of queries. These are grouped into two categories; queries of device-specific information and user interaction.

#### 2.4.1. Device-specific queries

There are two queries in this category. The first returns a pair of numbers, giving the size of the device in inches:

devicesize: $\rightarrow$ *width*, *height*

The second query is intended for precise control over text string placement. It takes as input a text string *s* and gives back five numbers; the coordinates of the lower left and upper right corners of the text bounding box and the width of the string:

textsize: $s \rightarrow (llx, lly),\ (urx, ury),\ width$

The coordinate system used for these numbers is the one described in section 2.3.3.

#### 2.4.2. Interactive queries

There are three interactive queries: locating the mouse, selecting a menu item and getting a rectangle. The locator query has two forms:

locator: $n \rightarrow (x_1, y_1),\ \ldots,\ (x_m, y_m),\ \ m \le n$

locator: $\rightarrow (x, y),\ status$

In the first form, *n* positions are requested and up to *n* will be returned (less than *n* if input intentionally ends prematurely) in the current coordinate system. Positions are indicated, and input is terminated in a device-dependent way. For example, with a mouse as the input device, a position may be recorded for each click of a mouse button and input terminated with the click of another mouse button. In the second form, the current position of the pointing device is returned immediately, together with a device-dependent status indicator (such as the up-down status of the mouse buttons).

The selector primitive is intended for menu selection, since the window manager of many interactive devices supports things like pop-up or pull-down menus. It simply supplies the user with a set of text strings and returns the number of the item the user selects, indexed from 1.

selector: $s_1,\ \ldots,\ s_n \rightarrow n.$

A return value of 0 indicates that no selection was made. In a way, this query doesn't belong in the device driver, as it doesn't bear directly on graphics, at least in the narrower sense of making (statistical) graphs. We choose to leave it here (as in S currently) since one can view graphics in the broader sense as it is used, for example, in the phrase ‘‘graphical user interface,’’ and also for the pragmatic reason that it is often a reasonable or possibly the only place to provide this functionality to the S user.

The getrect query is a request for the user to indicate a rectangular region on the display:

getrect: $\rightarrow (llx, lly),\ (urx, ury)$

The return value consists of the coordinates (in the current coordinate system) of the lower left and upper right corners of the selected region. The method used to indicate the region is device dependent, but for a screen device will typically be the action of dragging out a rectangle interactively with a mouse.

### 2.5. Device control

The last group of device primitives are those that control its general behavior:

initialize:

wrapup:

clear:

flush:

The first two of these are for any actions necessary when a device is started or ended. For example, wrapup might close a file, or a pipe or a window.

The clear primitive should erase all graphics from the current device. What this actually means depends on the device. The two common cases are a window and a printer; in the first case the graphics in the window are just erased by painting the entire surface with color 0, while in the second case a page is printed (or at least saved) with the current picture, and then any necessary actions for starting a new page are performed. The clear primitive will get called for the beginning of each ‘‘page’’, including the first one. The flush primitive is intended for device drivers that write to files, and simply causes any buffered output to be flushed.

## 3. S interface

We have now seen a complete description of Pictor’s low-level graphics model. In this section, we describe an implementation of the model in S in terms of a set of simple S functions that map directly to the primitives previously described. For the most part, these functions are in one-to-one correspondence with the primitives.

A device driver must be explicitly started with a function call. For example, the driver for PostScript printers uses `ps()` and the driver for a window on an SGI machines uses `gl()`. The return value of one of these functions is a *device descriptor*—a small integer that can be used to direct future primitives to a specific device. In each of the functions below, this integer is the last argument, and is called `device`.

In order to manipulate the coordinate system in Pictor, the following S function is supplied:

```
dev.ctm(ctm, device)
```

This takes a six-element numeric vector, representing a coordinate transformation matrix as described in section 2.1 and tells Pictor to establish it as the current CTM. Until another call to `dev.ctm`, all coordinates given in calls to drawing primitives will be transformed by the device driver into absolute coordinates using the `ctm` argument above.

The functions corresponding to the drawing primitives are:

```
dev.polylines(x, y, device)
dev.polygons(x, y, device)
dev.segments(x1, y1, x2, y2, device)
dev.rectangles(x1, y1, x2, y2, device)
dev.arcs(cx, cy, r, a1, a2, device)
dev.text(x, y, s, device)
dev.marks(x, y, device)
dev.image(x1, y1, x2, y2, nrow, ncol, data)
```

The arguments in these are mainly self-explanatory. For example, for `dev.polylines`, the `x` and `y` arguments give the *x* and *y* coordinates of the points for the polylines. They must be of the same length. In `dev.text`, the third argument is a vector of character strings, again of the same length as `x` and `y`. The `nrow` and `ncol` arguments to the `dev.image` function are single integers giving the dimension of the image and the `data` argument is an integer vector indexing the color table.

The effect of calling one of the drawing primitive functions is to cause the specified device immediately to execute the corresponding primitive. Whatever rendering parameters are needed for the primitive must have been previously set; it is an error to fail to do so.

The next group of functions set the values of the rendering parameters:

```
dev.color(z, device)
dev.fill(z, device)
dev.linestyle(z, device)
dev.linewidth(z, device)
dev.font(z, device)
dev.pointsize(z, device)
dev.textadjust(z, device)
dev.textrotation(z, device)
dev.marktype(z, device)
dev.marksize(z, device)
dev.markrotation(z, device)
```

Each of these takes a numeric vector, with the exception of `dev.textadjust`, whose argument is a vector of character strings. The action of the function is simply to set the rendering parameter to the given value; the value will be used later when drawing occurs. The functions that expect table indexes (color, fill, linestyle, font and marktype) ought to be given integers that correspond to previously defined table values. If not, a device-dependent action may occur, or an error may be flagged.

Table entries are defined with the following four functions:

```
dev.colordef(n, rgb, device)
dev.linedef(n, pattern, offset, device)
dev.fontdef(n, name, device)
dev.markdef(n, device)
```

Each of these expects as its first argument a single integer telling which slot of the table is being defined. A color definition is a triple `rgb` of numbers in [0,1] giving the relative strengths of red, green and blue for the color. The `pattern` and `offset` arguments for `dev.linedef` are a numeric vector and a single number, as discussed earlier. The `name` argument for defining a font is a character string giving the name of the font (known to the device driver).

The `dev.markdef` function requires more explanation. In order to define the pieces of a mark, `dev.markdef` is first called with the number of the mark about to be defined. This alerts the device driver that a mark is about to be defined, and subsequent calls to any of the drawing primitive functions or rendering parameter functions described above will refer to the mark being defined. When the definition is finished, `dev.markdef` is called with an argument of `-1`, to alert the device driver to stop diverting commands into the definition. As an example, we could define the circle mentioned at the end of section 2.3.4 as mark number 3, and make sure that it comes out unfilled, in the following way:

```
dev.markdef(3)
dev.fill(NA)
dev.arcs(0,0,1,0,360)
dev.markdef(-1)
```

Device queries are accomplished through these two functions:

```
dev.devicesize(device)
dev.textsize(s, device)
```

The first returns a list with two components named `width` and `height` giving the device width and height in inches. The second expects a single string `s` and returns a list with five components named `lx`, `ly`, `ux`, `uy` and `width`, as described earlier.

User queries are accomplished through these functions:

```
dev.locator(n, device)
dev.selector(choices, device)
dev.getrect(device)
```

The `dev.locator` function activates the first of the locator forms given earlier if `n` is 0 or the second if `n` is positive. It returns a list with components `x` and `y` holding the coordinates of the pointed-to positions and `status` giving the device-dependent status at the time the last position was found.

The `dev.selector` function expects a vector of character strings and after prompting with a menu labeled with these strings, returns the index of the pointed-to string, or 0 if no selection was made.

Finally, `dev.getrect` prompts for an input rectangle and returns it as a four-element numeric vector giving the coordinates of the lower-left corner and the upper-right corner (in that order) in the current coordinate system.

Device control is achieved through these functions:

```
dev.initialize(device)
dev.wrapup(device)
dev.clear(device)
dev.flush(device)
```

The first is typically called in the device driver function and so need not be called explicitly. Normally `dev.flush` is unnecessary, though it may be useful if graphics output is going to a file (as can be done with the PostScript driver) to flush the output buffer.

## 4. Comparison with S graphics

In this section we compare Pictor's low-level graphics model with that used by S up to Version 3; we shall call the latter ''S graphics.'' The chief difference between the two is that in Pictor there is a clean break between the low-level part of the model and the rest of the graphics system. In S graphics, control over the model was achieved through the ''graphical parameters,'' a large collection of numbers that described everything from how many high-level plots per page there were, to whether the current device was able to rotate characters. The main function for manipulating this array was `par`. Though certainly powerful, `par` was unwieldy to use, not least because all controls were lumped together.

Pictor, on the other hand, separates out the functionality of the graphical parameters into several groups. The group we have been discussing in this paper consists of the rendering parameters; those that control the actual appearance of the drawing primitives. This leaves other parameters, such as those dealing with axes or plot layout, to be implemented in the high-level model. A clean break like this avoids the sometimes inexplicable parameter dependencies that occur in S graphics.

Another difference in the two systems is the way in which the persistence of parameters is dealt with. In S graphics, the ''master copy'' of the graphical parameters was stored below the S level (in a C array) and so it was difficult to manipulate this invisible master copy in a reliable way. In Pictor, nothing below the S level ever changes a rendering parameter or table entry or the CTM; this is entirely the domain of the `dev.xxx` functions described in the last section. In this way, all manipulation of parameters is visible and therefore understandable and modifiable.

Another way in which parameters differ is in the tables for color, font and so on. These were all hard-wired into the device drivers in S graphics (though in later drivers this is less true) in different ways. Pictor gives a uniform, clean, explicit way of defining colors, line styles, fonts and marks.

Pictor rendering parameters can have vector values; this is more consistent with the general S philosophy of dealing with vectors whenever it makes sense, and works better in conjunction with vector-valued drawing primitives, than the single-number graphical parameters of S graphics.

Pictor's drawing primitives have been updated to match more closely the capabilities of modern graphics devices. For example, the arcs primitive is new (there was no way to draw a curve in S graphics), as is the image primitive. The concept of marks is particularly important in Pictor, as it allows customization of the one of the most fundamental aspects of statistical graphics, the plotting of symbols. One thing that has become clear in recent work on graphical perception (see, for example, Cleveland, 1994) is that the shape of the marks can have a large influence on the success of the graph and so the ability to customize marks is a welcome feature.

## References

Adobe Systems Incorporated (1985), *PostScript Language Reference Manual*, Addison-Wesley, Reading, Massachusetts.

Becker, Richard A., John M. Chambers, Allan R. Wilks (1988), *The New S Language*, Wadsworth, Pacific Grove, California.

Chambers, John M. (1994), *Overview of Version 4 of S,* AT&T Bell Labs Research Report, Murray Hill, New Jersey.

Cleveland, William S. (1994), *The Elements of Graphing Data*, Second edition, Hobart Press, Summit, New Jersey.

Wilks, Allan R. (1994a), *Constraint-Based Representations of Statistical Graphs*, AT&T Bell Labs Research Report, Murray Hill, New Jersey.

Wilks, Allan R. (1994b), Pictor *Programmer's Guide*, AT&T Bell Labs Research Report, Murray Hill, New Jersey.

Wolfram, Stephen (1988), *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Reading, Massachusetts.