Department of Computing Science Working Paper Series

Faculty of Engineering and Information Sciences

1985

Graphical Kernel System (GKS)

M. A. Sahib
University of Wollongong

Graphical Kernel System (GKS)

M. A. Sahib

Dept. of Computing Science
University of Wollongong
Wollongong
N.S.W.

<u>ABSTRACT</u>

There has been a longstanding need for an international standard in computer graphics. The decision by ISO to define an international graphics standard was supported throughout the world and over the last six years there has been a thorough technical evaluation by the graphics experts of many countries. The result was that in December 1982 the Graphical Kernel System was accepted as a Draft International Standard and is expected to become the International Standard towards the end of 1985. The aim of this report is to introduce the reader to the concepts of GKS. All examples in this report are expressed using the 'C' language because of the GKS implementation available. The author assumes no prior knowledge of computer graphics.

Table of Contents

The GKS implementation installed at the Dept. of Computing Science is the pre-released version of August 1983. Hence, section headings followed by (*) indicate that these features are either not available presently or are available in a restricted form.

1. Graphical Kernel System (GKS) - Introduction

The Graphical Kernel System (GKS) provides a set of functions for the purpose of generating 2D pictures on vector graphics and/or raster devices. It also supports operator input and interaction by supplying basic functions for graphical input, picture segmentation and subsequent storage, retrieval and dynamic modification of graphical information.

In doing so GKS provides a functional interface between an application program and a configuration of graphical input and output devices. The functional interface contains all basic functions for interactive and non-interactive graphics on a wide variety of graphical equipment.

The interface is at such a level of abstraction that hardware peculiarities are shielded from the application program. As a result, a simplified interface presenting uniform output primitives (POLYLINE, POLYMARKER, TEXT, FILL AREA, CELL ARRAY, GENERALIZED DRAWING PRIMITIVES) and uniform input classes (LOCATOR, STROKE, VALUATOR, CHOICE, PICK, STRING) is obtained.

A central concept, both for structuring GKS and for realizing device independence, is the workstation concept. Each workstation available at a particular installation is grouped into one of six categories (OUTPUT, INPUT, OUTIN, WISS, MO, MI) depending on the actions it can perform. The notion of multiple workstations allows simultaneous output to workstations of type OUTPUT and OUTIN and input from workstations of type INPUT and OUTIN. Facilities for internal and external storage of graphical information are provided by a special set of workstations of type WISS, MO and MI that use metafiles.

Not all of the GKS implementations support the full set of functions. Nine levels of implementation are defined to meet the different requirements of graphical systems and each GKS implementation, classified into one of these levels, provides the functions of that level only. The levels themselves are upward compatible.

GKS defines only a language independent nucleus of a graphics system. For integration into a language, GKS is embedded in a language dependent layer containing the language conventions, for example, function calls, parameters passing, etc. The implementation available on the Department of Computing Science's Perkin-Elmer 3230s is coded in 'C' and runs under UNIX.*

*UNIX is a Trademark of Bell Laboratories.

Before describing the features and facilities available in GKS, distinction must be made between the persons involved with any particular installation. Firstly, the installation manager (or implementor) is in charge of a particular installation of GKS. He is the one who installs new workstations / devices as they are incorporated into GKS and thus provide access to and information about the workstations available at a particular site. The application programmer constructs application programs that use GKS in order to complete some task or problem. Finally, an operator uses the application programs to get specific instances of the task or problem done by either executing it or providing data for the interactive ones. Henceforth, the latter two groups will be collectively refered as users of GKS.

1.1. Concepts

The graphical output that is generated by GKS is built up from two groups of basic elements called output primitives and primitive attributes. The output primitives are abstractions of basic actions a device can perform, such as drawing lines, markers or printing character strings. The attributes control the aspects of output primitives on a device, such as line types, marker types, text characteristics (eg. height, spacing, expansion factor) and colour. Non-geometric aspects such as colour can be controlled for each workstation individually to make best use of its capabilities.

The graphics information that is input from a device, as a result of operator actions, is mapped by GKS into one of six classes of input each represented by a data type referred to as a logical input value. An instance of such a device representation is called a logical input device. The effect of the input action on the display surface, such as prompt and echoes, is controlled by GKS for each logical input device individually.

The two abstract concepts (abstract input and abstract output) are the building blocks of the so called abstract workstation. A workstation of GKS represents a unit consisting of zero or one display surface and zero or more input devices, such as keyboard, tablet and lightpen. The workstation presents these devices to the application program as a configuration of abstract devices thereby shielding hardware peculiarities.

The geometric information (coordinates) contained in the output primitives, attributes and logical input values can be subjected to transformations. These transformations perform mapping between three coordinate systems, namely:

(a) World Coordinates (WC) used by the application programmer to describe graphical information to GKS.

(b) Normalized Device Coordinates (NDC) used to define a uniform coordinate system for all workstations.
(c) Device Coordinates (DC), one coordinate system per workstation, representing its display surface coordinates. Input containing coordinates are expressed to GKS by the device using DC values.

Output primitives and attributes are mapped from WC to NDC by normalization transformation, from NDC to NDC by segment transformation (as indicated by a transformation matrix defining rotation, scaling and shift factors) and from NDC to DC by workstation transformation. Input from the display surface (expressed in DC) are mapped by an inverse workstation transformation from DC to NDC and by one of the inverse normalization transformation from NDC to WC.

Output primitives and primitive attributes may be grouped together in a segment. Segments are units for manipulation and change. Manipulation includes creation, deletion, and renaming while change includes transforming a segment, changing its visibility and also highlighting segments, ie. causing segments to "flash". Segments also form the basis for workstation independent storage of pictures at run time. Via this storage, a special workstation called Workstation Independent Segment Storage (WISS), segments can be inserted into other existing ones or transferred to other workstations.

The attributes which control the appearance of parts of the picture (output primitives, segments, prompt and echo types of input devices) on the display surface are organized in a uniform manner. Two groups of attributes apply to the appearance of each output primitive, namely: primitive attributes and workstation attributes. Once primitive attributes are used to create an instance of a primitive they are bound to that primitive. These attributes include all geometrical aspects of a primitive such as character height of text or pattern size of fillarea. The non-geometrical aspects of primitives are controlled by the primitive attributes in one of two ways. Either a single attribute is used to specify all the non-geometric aspects of the primitive by an index which points to a workstation dependent representation (set of values) or one attribute is used to specify each of the non-geometric aspects in a workstation independent way. The former is referred to as bundled specification and the latter is individual specification.

Workstation attributes include the actual representation on a workstation pointed to by indices used in bundled specification of non-geometrical aspects. For example, the representations (or "bundles" as they are collectively known) for polylines each contain values of linetype, linewidth scale factor and colour index. Workstation attributes also specify the colour index and pattern tables and the control over the time interval between modifications and alteration to displays to accommodate these changes. Workstation attributes can be reset dynamically.

The appearance of segments is controlled by segment attributes, which are segment transformation (used for scaling, rotating and shifting), visibility, highlighting, segment priority values and detectability. These may also be set dynamically. Segment attributes can be a basis for feedback during manipulations (eg. highlighting).

The attributes which control the operation of logical input devices can be specified either upon initialization, or as a part of the input device setting, depending upon the attributes. Through initialization, an initial value, a prompt and echo technique and an area of the screen for echoing can be specified. A data record may further provide device specific attributes. Through the input device setting the operating mode may be selected and the echo may be switched on or off. The operating modes of logical input devices specify who (operator or application program) has the initiative:

- SAMPLE: input is acquired directed by the application program;
- REQUEST: input is produced by the operator in direct response to the application program;
- EVENT: input is generated asynchronously by the operator and is collected in a queue for the application program.

At run time, GKS can be in one of five different operating states (GKS Closed; GKS Open; At Least One Wkst. Open; At Least One Wkst. Active; Segment Open). Associated with each of these states is a set of allowable GKS functions and a set of state variables. The operating state values allow the proper specification of initialization and the effect of various functions, especially with respect to the maintenance of device independence and proper error identification. One special set of GKS functions, called the inquiry functions are allowed in all states. They give read-only access into appropriate state lists that are created when GKS is in a particular state. In this way, useful information can be provided when errors occur. Other inquiry functions allow read-only access into the workstation description tables to allow the application program to adopt to particular workstation capabilities. Inquiry functions do not cause errors. Instead, they return to the user information specifying whether a valid inquiry has been made.

GKS provides an interface to a system of filing graphical information for the purpose of external long term storage and exchange. The interface consists of a GKS output workstation (workstation type MO), which writes output information to a so-called graphics Metafile (a sequential file), and a GKS input workstation (Metafile of type MI). In addition to the normal functions for output to workstation, a GKS Metafile output workstation may accept items containing non-graphical information.

2. Graphical Output

2.1. Introduction

The main objective of the Graphical Kernel System (GKS) is the production and manipulation of pictures in a way that does not depend on the host computer or graphical devices being used. Such pictures vary from simple line graphs, to engineering drawings of intergrated circuits (using colour to differentiate between layers), to images representing medical or astronomical data in grey-scale or colour.

In GKS, pictures are considered to be constructed from a number of basic building blocks. These basic building blocks, or primitives as they are called, are of a number of different types each of which can be used to describe a different component of a picture. The four main primitives in GKS are:

(i) polyline: which draws a sequence of connected line segments through specified points;

(ii) polymarker: which marks a sequence of points with the same symbol;

(iii) fillarea: which displays a enclosed area using specified textures or patterns;

(iv) text: which draws a string of characters at the specified location;

Associated with each primitive is a set of parameters which specify particular instances of that primitive. For example, the parameters of the text primitive are the string of characters to be drawn and the starting position of that string. Thus:

```
text("String", pos)
```

will draw the characters String at the position specified by the coordinates contained in pos.

Although the parameters enable the form of the primitives to be specified, additional data are necessary to describe the actual appearance (or aspects) of the primitives. For example, GKS needs to know the height of a character string and the angle at which it is to be drawn. These additional data are known as attributes.

The attributes represent features of the primitives which vary less often than the form of the primitives. Attributes will frequently retain the same values for the description of several primitives. Once the desired height has been obtained, for example, all the characters of a string may be plotted using that height.

2.2. Polyline

The main line drawing primitive of GKS is polyline which is generated by calling the function:

```
polyline(n, pts)
```

where pts is a pointer to a list structure containing of n coordinate pairs (pts->w.x and pts->w_y). The polyline generated consists of (n - 1) line segments joining adjacent points starting with the first and ending with the last.

As the first example suppose one wishes to plot a graph of a set of (19) data, possibly representing some experimental results. The data consists of a set of ordered pairs, thus:

```
x:  0.0   2.0   4.0   6.0  ....  29.0
y:  8.8   7.6   7.1   7.4  ....  13.2
```

The graph may be drawn by joining adjacent points with straight line segments. Thus we can plot our graph by using the following 'C' routine (Note : Wc is the list structure mentioned above pointing to the coordinate pairs expressed in WC):

```
graph()
{
        Wc   *cds;

        cds = pts;
        cds->w_x = 0.0;   cds->w_y = 8.8;   cds++;
        cds->w_x = 2.0;   cds->w_y = 7.6;   cds++;
        cds->w_x = 4.0;   cds->w_y = 7.1;   cds++;
                .                .                .
                .                .                .
        cds->w_x = 29.0;  cds->w_y = 13.2;
        polyline(19, pts);
}
```

This produces the output given in Figure 2-1. Note: In this and all subsequent examples, pts is a pointer to a list used for storing coordinate pairs.

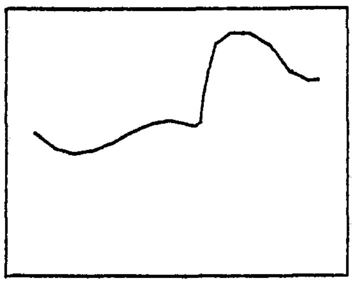

Figure 2-1

However, in order to interpret the graph one needs to relate the graph to the coordinate data by drawing the axes too. Without repeating the above code, the example can be extended to:

```
axes()
{
        Wc   * ax;

        ax = pts;
        ax->w_x = 29.0;   ax->w_y = 0.00;   ax++;
        ax->w_x =  0.0;   ax->w_y = 0.00;   ax++;
        ax->w_x =  0.0;   ax->w_y = 17.0;
        polyline(3, pts);
        graph();
}
```

This will produce the output in Figure 2-2.

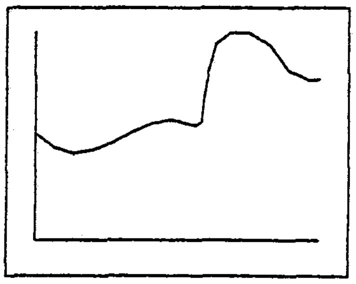

Figure 2-2

2.3. Polyline Representation

Any complex picture will contain a number of polylines and it may be necessary to differentiate between them. This is done by using different representations of the polylines. Different representations can be obtained by specifying different values for a polyline attribute associated with the polylines.

In the above example, one may wish to draw a constant Y line to enable points above the line to be identified. Again, this can be achieved by using the polyline primitive. Note that there is no need to use separate variables to specify each polyline. The same variables may be reused to specify previous polylines. Thus to draw the same picture with the line Y = 9.5 we need only add:

```
midline()
{
        Wc   *ax;

        ax = pts;
        ax->w_x =  0.0;   ax->w_y = 9.5;   ax++;
        ax->w_x = 29.0;   ax->w_y = 9.5;   ax++;
        polyline(2, pts);
};
```

This generates the graph shown in Figure 2-3. (In subsequent examples separate variables shall be specified for each polyline to simplify the discussion of the examples).

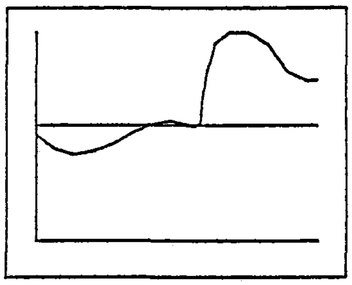

Figure 2-3

This does not give a satisfactory result as it is not easy to distinguish between the new line and the previous one drawn to represent data. One needs to draw the two lines in such a way that they can be distinguished. This is done by specifying each polyline with a different representation.

As there are numerous different facilities available on actual devices for distinguishing lines (eg. colour, pen thickness, line types, etc.), the method of distinguishing lines is described to GKS in a device independent manner. GKS does this by having a single attribute called polyline index. If a user wishes to distinguish between two polylines, he does so by defining each of them with a different polyline index. The exact representation of the polyline on a specific device depends on what facilities are available. For the moment, all that we need to know is that polylines with different polyline index values associated with them can, and normally will, appear differently on the actual display used. The polyline index is set by the GKS function:

```
s_pl_i(n)
```

This sets the polyline index to the value n for all subsequent polylines until its value is reset by another call to this function.

Assume that polyline representation 1 is a solid line and representation 2 a dashed line. For both representations, a standard line width and colour is used. The pictures drawn so far had used the default value of the polyline index which was, in fact, 1.

This is illustrated by the following sequence of calls to the previously defined functions:

```
picture()
{
        Wc    *ax;

        s_pl_i(1);
        axes();
        midline();

        s_pl_i(2);
        graph();
}
```

This produces the results shown in Figure 2-4.

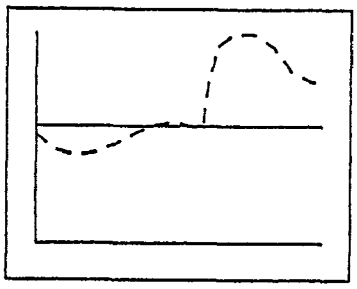

Figure 2-4

It is good practice to set the polyline index to 1 explicitly before the first polyline (drawn by axes), although this will not alter the polyline's appearance as the default value is 1. Note that the polyline, called from midline, is also drawn using a polyline index of 1 as the index had not been reset. In contrast, the line representing the data is drawn using a polyline index value of 2 to differentiate it from the axes. This method of controlling the appearance of primitives is a very powerful feature. For example, a high level graphics routine to perform contouring will need to distinguish between contours, whether to highlight those above or below a certain height or every nth one. By using a different polyline index to draw each set of contours in the higher level routine, the application program which uses it may choose the representation for each index to achieve the desired contour map.

2.4. Polymarker

Instead of drawing lines through a set of points, we may wish just to mark the set of points. GKS provides the primitive polymark to do just this. A polymarker is generated by the function:

```
polymark(n, pts)
```

where the arguments are the same as those for the polyline function. Polymark places a centered marker at each point.

One may now, as shown in Figure 2-5, plot the data points rather than a line through the points. This is done by replacing the call to polyline (in our case, the call inside function "graph") with a call to polymark.

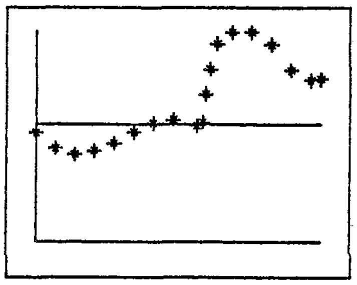

Figure 2-5

Of course, one may wish to both identify the set of points with markers and to plot the line through them as follows:

```
graph()
{
        Wc  *cds;

        cds = pts;
        cds->w_x = 0.0;     cds->w_y = 8.8;  cds++;
        cds->w_x = 2.0;     cds->w_y = 7.6;  cds++;
        cds->w_x = 4.0;     cds->w_y = 7.1;  cds++;
                 .                   .                .
                 .                   .                .
        cds->w_x = 29.0;    cds->w_y = 13.2;

        cds = pts;
        s_pm_i(1);
        polymark(19, pts);

        pts = cds;
        s_pl_i(1);
        polyline(19, pts);
}
```

This composite output is shown in Figure 2-6.

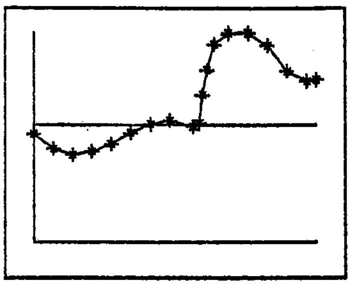

Figure 2-6

In a similar manner to polyline, GKS provides a facility to distinguish different sets of points. This is done by assigning different values to a polymarker attribute called the polymarker index. The polymarker index is set by the function:

```
s_pm_i(n)
```

where n is the desired value of the polymarker index. Assume that polymarker representation 1 is an asterisk, 2 is a circle and 3 is a plus sign, all in standard size and colour. A second set of (10) data is now introduced. points:

```
x:  15.7   17.0   17.7   17.3  ....  4.7
y:   7.0    6.1    5.0    3.8  ....  5.2
```

The following program will draw the two sets of points with different representations so that one can distinguish between them:

```
wing_graph()
{
        Wc  *cds;

        cds = pts;
        cds->w_x = 0.0;     cds->w_y = 8.8;  cds++;
        cds->w_x = 2.0;     cds->w_y = 7.6;  cds++;
        cds->w_x = 4.0;     cds->w_y = 7.1;  cds++;
               .                .                .
               .                .                .
        cds->w_x = 29.0;    cds->w_y = 13.2;

        s_pm_i(1);
        polymark(19, pts);

        cds = pts;
        cds->w_x = 15.7;    cds->w_y = 7.0;  cds++;
        cds->w_x = 17.0;    cds->w_y = 6.1;  cds++;
        cds->w_x = 17.7;    cds->w_y = 5.0;  cds++;
               .                .                .
               .                .                .
        cds->w_x =  4.7;    cds->w_y = 5.2;

        s_pm_i(2);
        polymark(19, pts);
}
```

This is shown in Figure 2-7.

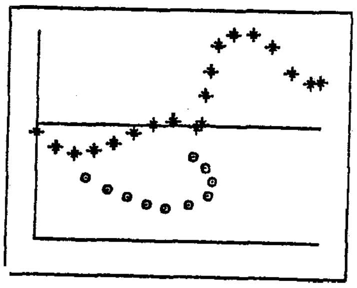

Figure 2-7

2.5. Fillarea

There are many applications for which line drawings are insufficient. The design of integrated circuit layouts requires the use of filled rectangles to display a layer. Animation systems need to be able to shade areas of arbitrary shape. Other applications using colour only realize their potential when they are able to use coloured areas rather than coloured lines.

GKS provides a fillarea primitive to satisfy the application needs which can use the varying device capabilities. Defining an area is a fairly simple extension of defining a polyline. A list of points is specified which defines the boundary of the area. If the area is not _closed_ (ie. the first and last points are not the same), the boundary is defined by the points but extended to join the last point to the first. A fillarea may be generated by invoking the function:

```
fillarea(n, pts)
```

where, as usual, _pts_ points to _n_ number of coordinates. Extending the original data set to describe a closed area, a further 24 coordinates is added.

```
x:  28.0   27.2   25.0   23.0  ....  1.3
y:  12.3   11.5   11.5   11.2  ....  6.0
```

Like other primitives considered so far, fillarea also has a representation accessed by a fillarea attribute called the _fillarea index_. The fillarea index is set by the function:

```
s_fa_i(n)
```

where _n_ is the desired value of the fillarea index. Filled areas may be distinguished by their filling style (called _interior style_ in GKS) and colour. Assume that fillarea representation 1 is interior style HOLLOW and representation 2 is interior style SOLID, each in standard colour.

```
fill_graph()
{
        Wc  *cds;

        cds = pts;
        cds->w_x = 0.0;     cds->w_y = 8.8;  cds++;
        cds->w_x = 2.0;     cds->w_y = 7.6;  cds++;
        cds->w_x = 4.0;     cds->w_y = 7.1;  cds++;
                .                .                .
                .                .                .
        cds->w_x = 29.0;    cds->w_y = 13.2;  cds++;
        cds->w_x = 28.8;    cds->w_y = 12.3;  cds++;
        cds->w_x = 27.2;    cds->w_y = 11.5;  cds++;
        cds->w_x = 25.0;    cds->w_y = 11.5;  cds++;
                .                .                .
                .                .                .
        cds->w_x = 4.1;     cds->w_y = 6.0;

        s_pl_i(1);
        axes();

        s_fa_i(1);
        fillarea(43, pts);
}
```

This will produce the output like a duck given in Figure 2-8.

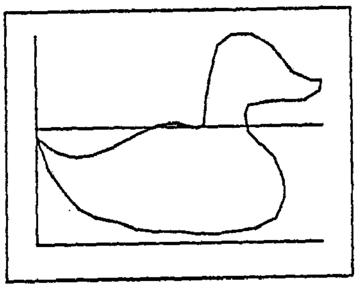

Figure 2-8

It is shown that for interior style HOLLOW only the boundary has been drawn. It is interesting to note, however, the contrast result that would have been obtained if the last two statements of the above code were changed to:

```
        s_pl_i(1);
        polyline(43, pts);
```

In this case the output would be as shown in Figure 2-9.

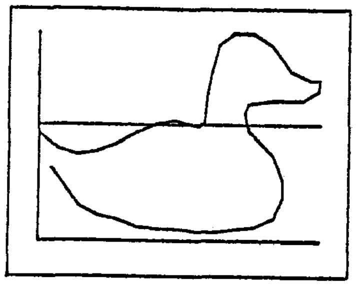

Figure 2-9

This illustrates the fact that the fillarea primitive is a true area primitive while the polyline is not. The last point is joined to the first by the fillarea primitive to complete the outline.

If the fillarea example were drawn using fillarea representation 2 (SOLID), ie. the last two statements were:

```
s_fa_i(2);
fillarea(43, pts);
```

the output given in Figure 2-10 would be obtained.

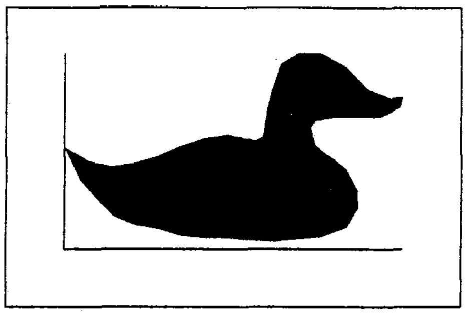

Figure 2-10

Let fillarea representation 3 be interior style HATCH using diagonal hatch and a standard colour. The fillarea primitive is defined to fill the area only and so does not include drawing the boundary. Thus, if the last two statements were changed to:

```
        s_fa_i(3);
        fillarea(43, pts);
```

the picture would be as shown in Figure 2-11.

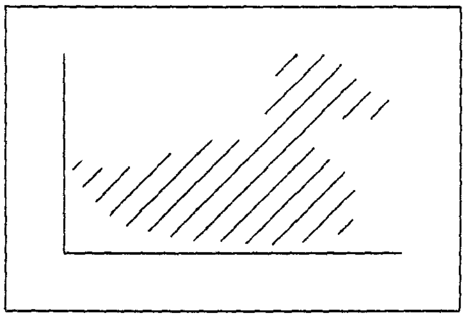

Figure 2-11

If the user wishes to include the boundary, he may additionally draw a polyline around the area. Here care must be taken because, as can be seen, polyline will not draw the complete enclosure. Hence, a 44th coordinate must be defined which, in fact, has the same value as the starting one. Thus, the whole code now look as such:

```
fill_graph()
{
        Wc  *cds;

        cds = pts;
/* */   cds->w_x = 0.0;      cds->w_y = 8.8;  cds++;
        cds->w_x = 2.0;      cds->w_y = 7.6;  cds++;
        cds->w_x = 4.0;      cds->w_y = 7.1;  cds++;
                 .               .              .
                 .               .              .
        cds->w_x = 29.0;     cds->w_y = 13.2;  cds++;
        cds->w_x = 28.8;     cds->w_y = 12.3;  cds++;
        cds->w_x = 27.2;     cds->w_y = 11.5;  cds++;
        cds->w_x = 25.0;     cds->w_y = 11.5;  cds++;
                 .               .              .
                 .               .              .
        cds->w_x = 4.1;    cds->w_y = 6.0;  cds++;
/* */   cds->w_x = 0.0;    cds->w_y = 8.8;

        s_pl_i(1);
        axes();

        cds = pts;
        s_fa_i(3);
        fillarea(43, pts);

        pts = cds;
        s_pl_i(1);
        polyline(44, pts);
}
```

This gives the hatched duck shown in Figure 2-12.

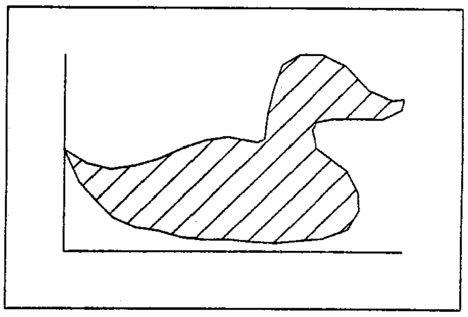

Figure 2-12

So far filling in of simple shapes have only considered. Suppose more data is added to the example data used in "fill_graph", above, by concatenating the coordinates contained in the function "wing_graph" making the 44th coordinate equal to the first one in "wing_graph", and so on. Hence, now there are 53 coordinates contained in the list pts.

If this new set of data is drawn using fillarea representation 1 (HOLLOW), the output would be as shown in Figure 2-13

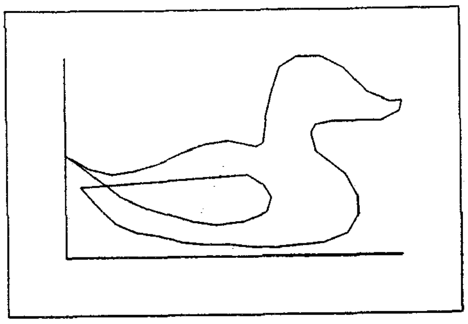

Figure 2-13

The area is now complex. Supposing one was to fill it with fillarea representation 2 (SOLID), which area(s) would be filled? The rule used is that if a line drawn from any point to infinity crosses the boundary an even number of times (including zero), the point is outside but if it crosses the boundary an odd number of times, the point is considered as being inside the fillarea. One can now confidently plot the data with fillarea representation 2 giving the output shown in Figure 2-14.

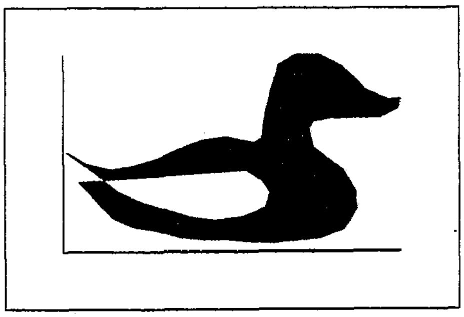

Figure 2-14

2.6. Text

So far no attempt has been made to put a title on the pictures. To do this, GKS has a text primitive which is used to title pictures or place labels on them as appropriate. A text string may be generated by invoking the function:

```
text("String", pts)
```

where pts points to the starting position (called text position) of the text "String".

However, text is more complicated than the other primitives that have been examined. Text is printed at different sizes, in different fonts, at different orientations and using different spacing. Graphics devices, on the other hand, are often not good at text, indeed some are not capable of any at all. Those that do often have only a restricted number of sizes, one or two orientations and a single font.

GKS attempts to match these requirements with its text primitive. Each of the other primitives examined so far has had a single attribute which controls the aspects of its appearance (via a representation). Since text is more complex, its appearance is affected by several aspects. Some of these are too important to be controlled by a representation. At the same time, they do not vary with every text string that is output. These important aspects are the character height, character up vector, text path and text alignment.

In GKS, they are attributes of text which may be individually assigned values. Of course, text also has a representation, accessed by the text index. An example of text primitive is:

```
pts->w_x = 0.2;
pts->w_y = 0.4;
string = "A char string";
text(string, pts)
```

which produces the output shown in Figure 2-15.

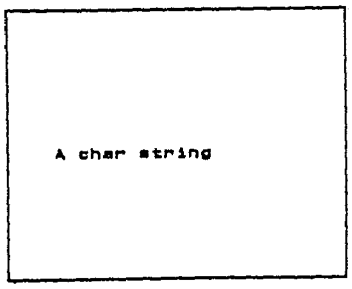

Figure 2-15

2.6.1. Text Attributes

The character height determines the height of the characters in the string. Since a character in a font will have a designed aspect ratio, the character height also determines the character width. Character height is set by the function:

```
s_ch_ht(ht) (*)
```

where ht is the character height specified as a real number.

```
s_ch_ht(1.0);
pos->w_x = 0.2;
pos->w_y = 0.6;
text('Char Height 1.0', pos);
s_ch_ht(2.0);
pos->w_x = 0.2;
pos->w_y = 0.4;
text('Char Height 2.0', pos);
```

produces the output given in Figure 2-16.

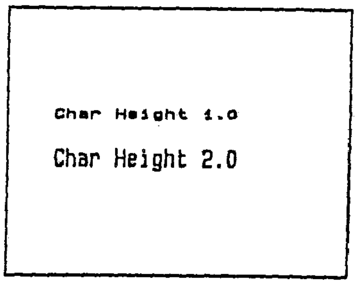

Figure 2-16

(*)Because this implementation is a pre-released version of GKS, to use this attribute, "6," is to be inserted before the first parameter if the Hewlett-Packard plotters is being used or a "9," if the Servogor plotter is being used.

The character up vector is perhaps the most important text attribute. Its main purpose is to determine the orientation of the characters. However, it also sets a reference direction which is used in the determination of text path and text alignment. The character up vector specifies the up direction of the individual characters. Its also specifies the orientation of the character string in that, by default, the characters are placed along the line perpendicular to the character up vector. The function:

```
s_ch_up(v) (*)
```

is used for setting the character up vector. v consists of two values (v->w_x and v->w_y) both of which are offsets from the horizontal text position of the up direction of the characters. For example, if both the values were 1.0 this would result in the characters being written at an angle of 45 degrees to the horizontal (ie. rise and run of 1.0). Thus specifying a vector is just another way of specifying an angle. The magnitude of the vector is not used. Thus:

```
vl->w_x = 1.0;              v2->w_x = 15.0;
vl->w_y = 1.0;              v2->w_y = 15.0;
s_ch_up(vl);                s_ch_up(v2);
```

are both the same effect-wise. Both cause subsequent character strings to be plotted at 45 degrees to the horizontal. Thus:

```
v->w_x = 1.0;
v->w_y = 1.0;
s_ch_up(v);
_text("A char string", pos);
```

generates the text string shown in Figure 2-17.

(*)Because this implementation is a pre-released version of GKS, to use this attribute, "6," is to be inserted before the first parameter if the Hewlett-Packard plotters is being used or a "9," if the Servogor plotter is being used.

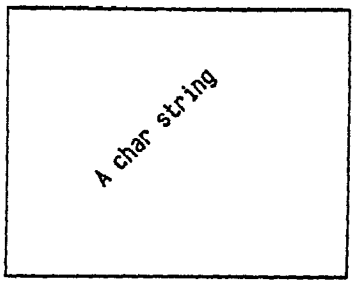

Figure 2-17

2.6.2. Text Representation

As mentioned earlier, text also has a representation provided by the text index. The text index is set by the function:

```
s_tx_i(n)
```

where _n_ is the desired value of the text index. Assume that text representation 1 is a Roman font at STROKE precision (the highest quality) using a unit character expansion factor, standard spacing and a standard colour. Let representation 2 be the same except that it uses CHAR precision (medium quality) and representation 3 be the same using STRING precision (the lowest quality). Then the following example:

```
v->w_x = -1.0;
v->w_y =  1.732;
s_ch_up(v);
s_tx_i(1);
text("Angled Text", pos);
```

would give the output shown in Figure 2-18, which is an accurate representation of the requested text.

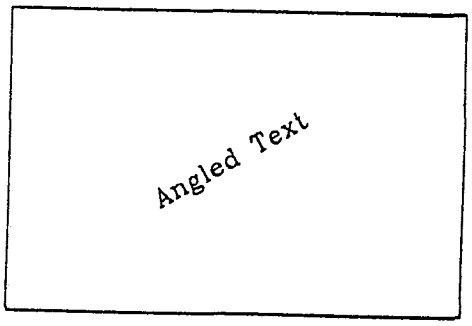

Figure 2-18

However, if one uses text index 3, the output shown in Figure 2-19 would be obtained.

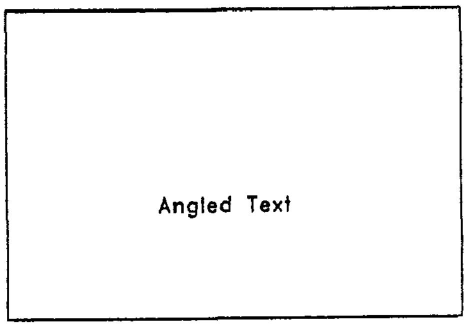

Figure 2-19

This is because the STRING precision text does not need to use all the text attributes. It is the text that will often be used on interactive devices where it needs to be produced quickly.

The highest precision text is that which would be used for the high quality output referred to earlier. To preview this high quality text one could use the medium precision text; using representation 2 gives the output shown in Figure 2-20.

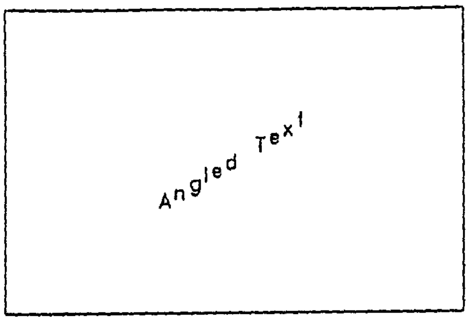

Figure 2-20

Here the position of the characters is precise but the orientation of the individual characters is upright. This gives a reasonable representation of the text suitable for preview but does not take the time required to produce the angled characters.

Now suppose text representation 4 is the same as text representation 1 except uses an italic font. Using text index 4 in the above example will then produce the output shown in Figure 2-21.

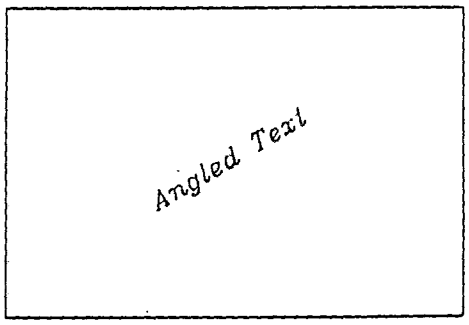

Figure 2-21

These examples show both flexibility of the text primitive of GKS and how text in GKS answers the application requirements referred to earlier.

2.7. Primitives and Attributes

In this chapter the four primitives polyline, polymarker, fillarea and text have been examined. One attribute has been described for each primitive except for text which requires three attributes. By suitable combinations of primitives and values of primitive attributes, quite complex pictures may be described to GKS with suitable annotations and titles, as shown in the following example:

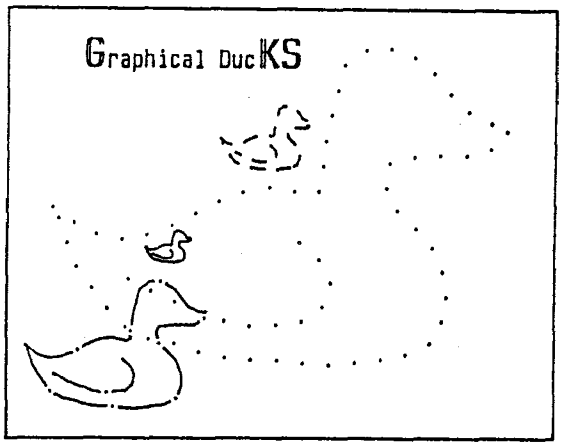

Figure 2-22

3. Coordinate Systems

3.1. Introduction

In the previous chapter, the main output primitives of GKS were described without any indication of the coordinate systems used. It can be assumed, and correctly so, that the coordinate system used to produce the diagrams so far, relied on equal unit length on both axes. Hence, if the unit length along the axes were not the same, quite different pictures would have been produced.

The user, therefore, needs to be able to specify the form that the picture will take on a specified device by being able to define differential scaling, if that is appropriate to the problem or task.

3.2. User and World Coordinates

For any problem, the user will have a preferred coordinate system. This coordinate system is referred to as user coordinates. Most frequently, this will be a Cartesian coordinate system, but it is possible to have logarithmically scaled or polar coordinate systems also. However, only Cartesian coordinate system is used by GKS. All others must be mapped onto this by the user himself. On the other hand, if the user coordinate system is Cartesian, it can also be used for the purpose of output description by GKS. To differentiate between the user preferred coordinates and the ones presented to GKS, the World Coordinate (WC) system is used to present graphical information to GKS.

3.3. Normalized Device Coordinates

Consider a typical example of graphical output and how it would be dealt with in GKS. The table below indicates some characteristics of New York City temperatures that are to be displayed in a graphical form on some output device.

	Over 100 Years		1980	
Month	Max	Min	Max	Min
Jan	72	- 6	55	9
Feb	75	-15	61	13
.	.	.	.	.
.	.	.	.	.
Nov	84	5	71	31
Dec	70	-13	63	18

One may be interested in displaying the range of temperatures experienced by New York over the past 100 years or how 1980 compares with the extremes of temperatures reached during that period. In this example, the user and world coordinate systems can be made identical if temperature is used as the Y coordinate and X goes from 1 to 12 representing the months of the year.

The above data could be stored in 4 lists each consisting of 12 items (month-wise), MaxH, MinH, Max80 and Min80 respectively. To produce a display on some output device, it is necessary to convert the WC by some transformation to the coordinates relevant to the specific device. This requires the following to be done:

(i) The WC origin has to be positioned on the display screen.

(ii) A unit in WC space has to be defined as a number of device coordinate units in each X and Y direction.

As there are so many different types of output devices, each with a different set of coordinate limits, there is a need to map all the different limits onto one virtual or normalized device. Such a device is defined by GKS which has a display surface visible in the range 0 to 1 in both X and Y directions. The coordinates of this device are called Normalized Device Coordinates (NDC) with its origin at the lower left corner. To produce visible output, the world coordinates defining the output must be mapped onto coordinate positions within the NDC unit square from 0 to 1 in both X and Y directions with the origin being at the lower left hand corner.

3.4. Window and Viewport Transformation

As mentioned earlier, the transformation from WC to NDC must specify the position in NDC space equivalent to a fixed position in the WC space. This enables the user to define the position and extent that the graphical output will occupy on the normalized device since the output may be mapped onto the whole or some small part of the NDC space.

A convenient method for doing this, and one that has been used extensively is to define a rectangular area in WC space (called the window) and define where this area will appear on the normalized device (called the viewport). GKS uses this method of specifying how WCs are transformed to NDCs . This is shown in Figure 3-1.

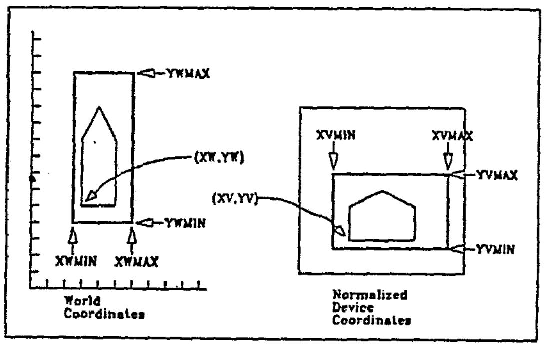

Figure 3-1

The window is defined by two pairs of coordinates expressed in world coordinate space (xwmin, ywmin) and (xwmax, ywmax). This is then transformed into the viewport expressed in NDC space consisting of (xvmin, yvmin) and (xvmax, yvmax). Thus, the mapping is defined in such a way that the points (xwmin, ywmin) and (xwmax, ywmax) are transformed to the positions (xvmin, yvmin) and (xvmax, yvmax) respectively with the requirement that (xwmin, ywmin) and hence (xvmin, yvmin) define the lower left hand corner. Also, any given position, ie. (xw, yw), is mapped to a position (xv, yv) in the NDC space in such a way as to preserve its relative position in the rectangle. Thus, the ratio of the distance of a point from the X boundaries to the length of the X range in WC is equal to the same ratio in NDC. This also applies to the Y axis. Note that both the window and viewport are rectangles but that there is no constraint that the aspect ratio of the viewport is the same as that of the window. Consequently, a tall thin house can be mapped into a short fat house on the device. GKS uses two functions to define a window and viewport as given below:

```
s_window(ntran, Wrect)
s_viewport(ntran, Nrect)
```

Ignoring the first parameter for the moment, the latter specifies the limits of the window and viewport respectively. (Note : Wrect and Nrect are "struct"s containing two coordinates pairs(xmin, ymin, xmax, ymax) in WC and NDC respectively. ntran is used to differentiate between different normalization transformations) Thus, to produce a graph of the New York maximum temperature for 1980 which fills the whole of the NDC visible space would require window and viewport definitions as follows:

```
Wrect   wrect = {0.0, 0.0, 12.0, 120.0};
Nrect   vrect = {0.0, 0.0, 1.0,  1.0};

s_window(ntran, &wrect);
s_viewport(ntran, &vrect);
cds = pts;
cds->w_x =  1.0;  cds->w_y = 55.0;  cds++;
    .      .       .       .       .
    .      .       .       .       .
cds->w_x = 12.0;  cds->w_y = 63.0;
polyline(12, pts);
```

This is shown in the second of the two graphs in Figure 3-2. (Note: axes and labels have been added to simplify the representation.)

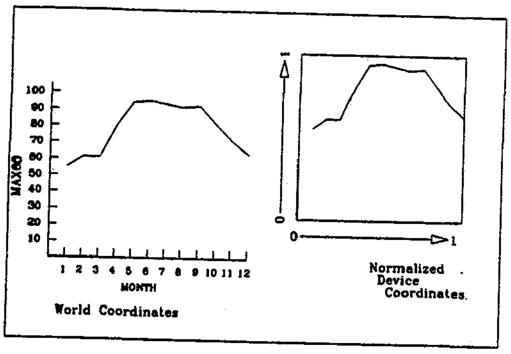

Figure 3-2

If the output is generated before either "s_window" or "s_viewport" is called, default values for these are used, equal to the unit square. Thus calling polyline before defining either of the window or viewport is equivalent to :

```
Wrect   wrect = {0.0, 0.0, 1.0. 1.0};
Nrect   vrect = {0.0, 0.0, 1.0, 1.0};

s_window(ntran, &wrect);
s_viewport(ntran, &vrect);
cds = pts;
cds->w_x =  1.0;  cds->w_y = 55.0;  cds++;
    .      .       .       .       .
    .      .       .       .       .
cds->w_x = 12.0;  cds->w_y = 63.0;
polyline(12, pts);
```

The main purpose of the window and viewport definitions is to define the transformation from WCs to NDCs. It is possible for several window and viewport combinations to give the same mapping coordinates from WC to NDC. For example:

```
        Wrect   wrect = {0.0,  0.0, 12.0, 100.0};
        Nrect   vrect = {0.25, 0.25, 0.75, 0.75};

        s_window(ntran, &wrect);
        s_viewport(ntran, &vrect);
```

gives the same mapping as :

```
        Wrect   wrect = {-6.0, -50.0, 18.0, 150.0};
        Nrect   vrect = {0.0, 0.0, 1.0, 1.0};

        s_window(ntran, &wrect);
        s_viewport(ntran, &vrect);
```

Both define the position (0,0) in WC as being mapped onto the NDC position (0.25, 0.25). Both define the scaling in the X direction such that 1 unit in WC is equivalent to 1/24 of a unit in NDC space and 1 unit in the Y direction in WC as being equivalent to 1/200 in the NDC space.

The transformation of coordinates from WC to NDC defined by the window to viewport mapping is often called the normalization transformation in GKS.

3.5. Multiple Normalization Transformation

One may wish to generate graphical images in the NDC space when the image may have different WCs for different parts of the picture. This is achieved by defining several window to viewport mappings. In the previous section, the parameter ntran in the definitions of the functions "s_window" and "s_viewport" is used to differentiate between the different normalization transformations. Values range from 0 to 16 (the maximum number of mappings allowed). Thus:

```
        s_window(ntran, &wrect);
```

defines the ntranth window and similarly for "s_viewport". It is good programming practice to define the windows and viewports used in a particular application at the head of the program. They can be thought of as a declaration defining the overall structure of the output.

To define which window to viewport mapping is to be used with a particular output primitive, a function is provided which selects a particular transformation as being active until another is selected. The function:

```
        sel-cntran(ntran)
```

specifies that normalization transformation ntran should be selected. The default number is 0. Consequently, the examples in the previous section were not strictly correct and should have been:

```
        s_window(ntran, &wrect);
        s_viewport(ntran, &vrect);

        sel_cntran(ntran);

        polyline(12, pts);
```

An example showing how multiple normalization transformations may be used is given below. The aim is to produce four graphs with the axes giving the maximum and minimum temperatures over the 100 year period and for the year 1980. Producing the four graphs requires the following normalization transformations:

```
        Wrect   wrect1 = {0.0, -15.0, 12.0, 120.0};
        Nrect   vrect1 = {0.1, 0.6, 0.4, 0.9};
        s_window(1, &wrect1);
        s_viewport(1, &vrect1);

        Wrect   wrect2 = {0.0, -15.0, 12.0, 120.0};
        Nrect   vrect2 = {0.6, 0.6, 0.9, 0.9};
        s_window(2, &wrect2);
        s_viewport(2, &vrect2);

        Wrect   wrect3 = {0.0, -15.0, 12.0, 120.0};
        Nrect   vrect3 = {0.1, 0.1, 0.4, 0.4};
        s_window(3, &wrect3);
        s_viewport(3, &vrect3);

        Wrect   wrect4 = {0.0, -15.0, 12.0, 120.0};
        Nrect   vrect4 = {0.6, 0.1, 0.9, 0.4};
        s_window(4, &wrect4);
        s_viewport(4, &vrect4);
```

Hence, the four graphs are as shown below in Figure 3-3, (with labels attached):

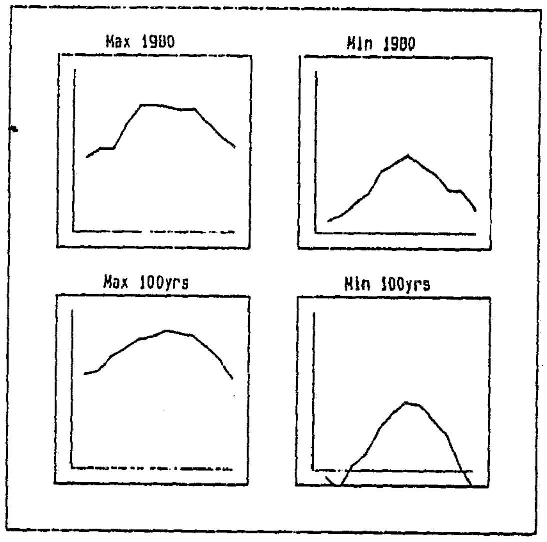

Figure 3-3

3.6. Graphical Annotation

As an example of how to use several window to viewport transformations to compose a picture, consider the problem of annotating the graph of maximum New York temperatures. A coordinate system with temperature in one direction and months of the year in the other is not the most ideal for defining the size and position of textual annotations to the graph. The graphical output falls into three distinct classes:

(i) the graphical information

(ii) annotation on the X axis

(iii)annotation on the Y axis

and these three windows need to be mapped onto viewports as shown in Figure 3-4.

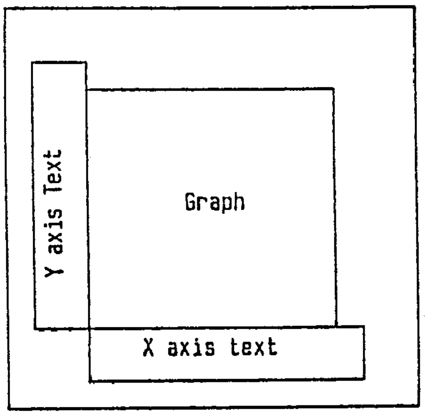

Figure 3-4

A possible set of window to viewport mappings would be:

```
        Wrect   wrect1 = {0.0, 0.0, 12.0, 120.0};
        Nrect   vrect1 = {0.2, 0.15, 0.8, 0.75};
        s_window(1, &wrect1);
        s_viewport(1, &vrect1);

        Wrect   wrect2 = {0.0, 0.0,  4.0, 16.0};
        Nrect   vrect2 = {0.0, 0.15, 0.2, 0.95};
        s_window(2, &wrect2);
        s_viewport(2, &vrect2);

        Wrect   wrect3 = {0.0, 0.0, 15.0, 3.0};
        Nrect   vrect3 = {0.2, 0.0, 0.95, 0.15};
        s_window(3, &wrect3);
        s_viewport(3, &vrect3);
```

Note the second and third normalization transformations preserve the aspect ratio. Space has also been left to allow the Y axis to extend above the height of the graph and similarly the X axis extends beyond the right end of the graph. The graph can be drawn as before:

```
        sel_cntran(1);
        axes(12, 120);
        polyline(12, pts);
```

where pts contains the graph coordinates as before.

To draw the text of Figure 3-5 on the Y axis would require:

```
        sel_cntran(2);
        s_pm_i(3);
        for (i = 1; i <= 12; i++)
        {
                pts->w_x = 4.0;
                pts->w_y = 1.0 * i;
                polymark(1, pts);
        }
        s_ch_ht(0.5);
        ang->w_x = 90.0;
        ang->w_x = 1.0;
        s_ch_up(ang);
        pts->w_x = 4.0;
        pts->w_y = 1.0;
        text("MAX NEW YORK TEMP.", pts);
        ang->w_x = 0.0;
        ang->w_x = 1.0;
        s_ch_up(ang);
        for (i = 1; i <= 12; i++)
        {
                pts->w_x = 2.0;
                pts->w_y = 1.0 * i;
                convtx((i*10), string);
                text(string, pts);
        }
```

The first part of the output generates tick marks on the axes using the plus sign marker. The second part defines the main annotation on the Y axis while the last part defines the temperature values. The function "convtx" converts the integer number given into its character form (returned via string - a char pointer) and is provided by the application itself.

The text on the X axis could be produced in a similar way:

```
        sel_cntran(3);
        for (i = 1; i <= 12; i++)
        {
                pts->w_x = 1.0 * i;
                pts->w_y = 3.0;
                polymark(1, pts);
        }
        s_ch_ht(0.25);
        ang->w_x = 0.0;
        ang->w_y = 1.0;
        s_ch_up(ang);
        s_tx_i(n);
        pts->w_x = 0.625;
        pts->w_y = 2.0;
        text("Jan Feb Mar .. Nov Dec", pts);
```

where text representation n is assumed to be in the same font but with the height and spacing between each character being smaller. The output is shown in Figure 3-5.

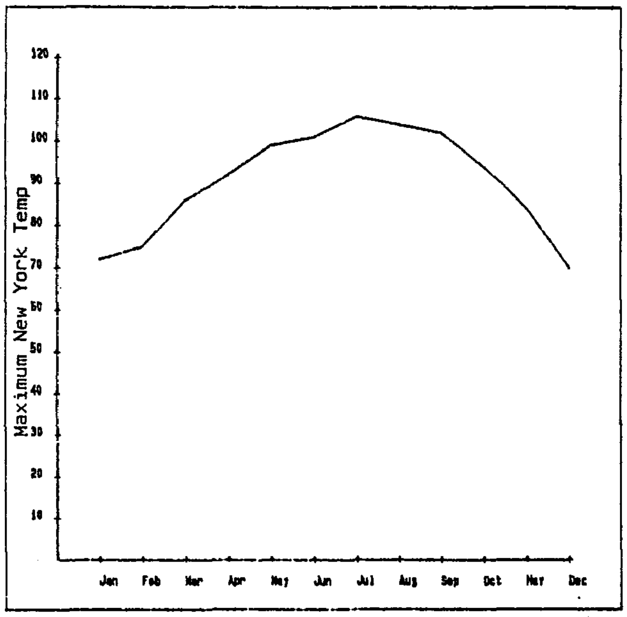

Figure 3-5

3.7. Clipping

A problem comes to light if we attempt to generate a graph of the minimum temperature by substituting the minimum coordinates for the previously used maxima. This is demonstrated in the output shown in Figure 3-6.

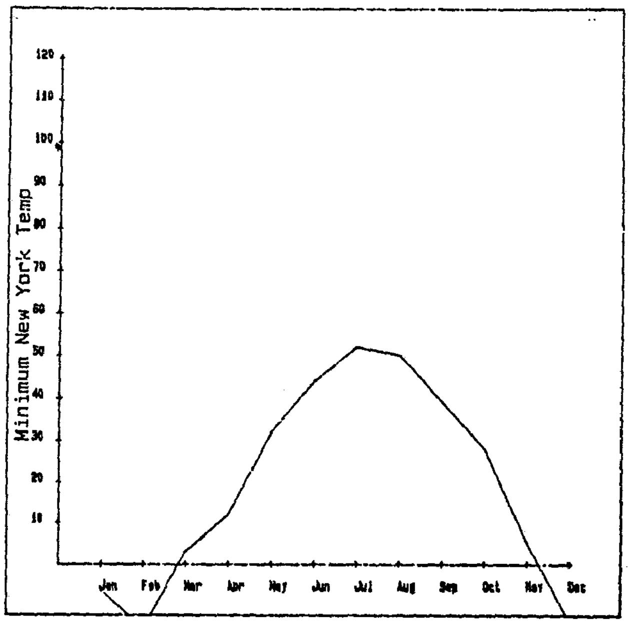

Figure 3-6

Because the Y coordinates of the minimum temperature go negative, the polyline is drawn outside the boundary of the viewport defined by the first normalization transformation. If the complete polyline is drawn, the above output is produced, resulting in a poor representation.

GKS, allows two possibilities for drawing outside of the viewport boundaries. In one mode the above is allowed, while in the other, drawing is <u>clipped</u> at the boundary, so that only those parts within the viewport are actually drawn. Both possibilities have their uses and it is up to the user to decide which is required. The default in GKS is to clip the output at the viewport boundary. The effect, therefore is as shown in Figure 3-7.

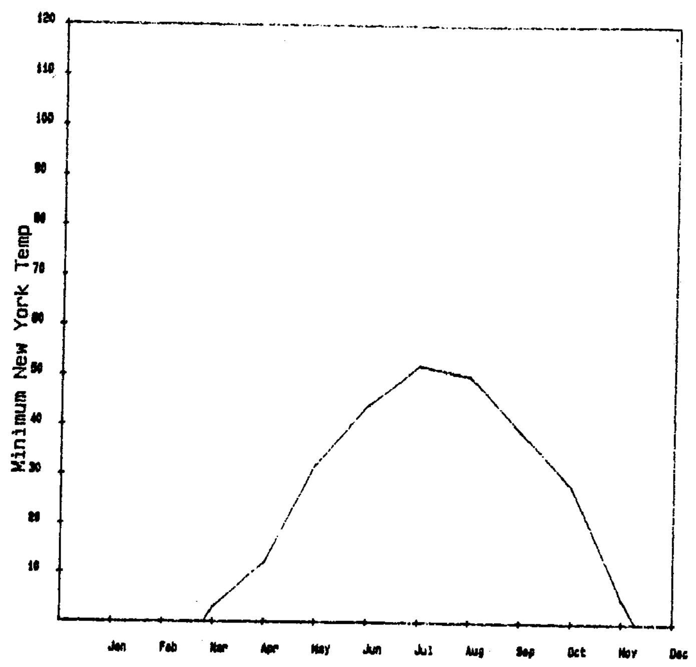

Figure 3-7

Clipping is switched on or off by the function:

    s_clip(clip)

where clip can take the values TRUE for CLIP or FALSE for NOCLIP as appropriate. The default value of an unset clipping indicator is CLIP. Note that there is only one clipping switch which is used for all normalization transformations.

If the clipping indicator is set to CLIP, polylines will be clipped at the boundary of the viewport. However, the effects on the other output primitives is not as straightforward. In the case of polymarkers, the individual markers will only be displayed if the center of the marker is within the viewport boundary (hence no display for marker positions just outside the viewport boundary). The clipping of text depends on the precision of the text being output. The lowest (STRING) precision text may be completely removed if the position of the start of the text is outside the viewport. For CHAR precision text, clipping is done on a character basis with the character only being output if the complete character box is within the viewport. For STROKE precision text, clipping is performed at the boundary of the viewport. The difference between CHAR and STROKE clipping is shown in Figure 3-8, the dashed lines indicate those parts that would be removed by clipping.

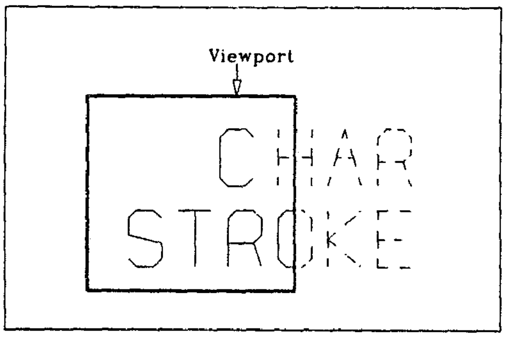

Figure 3-8

3.8. Normalization Transformation 0

In general, normalization transformations can be redefined at any time by either redefining the window or viewport. It is normal to define all the normalization transformations required in a picture prior to generating any graphical output. However, although this is regarded as good practice, it is not mandatory. In some circumstances, it may even be necessary to redefine the normalization transformation during output.

Normalization transformation 0 is special in that it cannot be redefined at all. Its value is equivalent to:

```
Wrect   wrect = {0.0, 0.0, 1.0, 1.0};
Nrect   vrect = {0.0, 0.0, 1.0, 1.0};

s_window(0, &wrect);
s_viewport(0, &vrect);
```

Any attempt to redefine it will cause an error. This is because it provides a mechanism whereby a user can define information directly in NDC rather than using world coordinates. Occasions may arise when this is preferable.

4. Segments

4.1. Introduction

The earlier chapters of this report have described how to construct a picture and position it on a virtual device. In many application areas, there is a need to display the same or similar graphical information many times possibly at different positions on the device or with minor changes. This leads to the need for some storage mechanism whereby pictures or sub-pictures can be saved for later use. GKS has the concept of a segment for this purpose. Graphical output can be stored in segments during the execution of a program and used at a later time. Segments have a number of segment attributes associated with them which allow the user to modify the appearance of the whole segment.

To take a simple example not using segments, consider a crude animation system in which the user may construct a scenic background and then display an object moving across the scene on successive frames, the initial frame being as shown below in Figure 4-1.

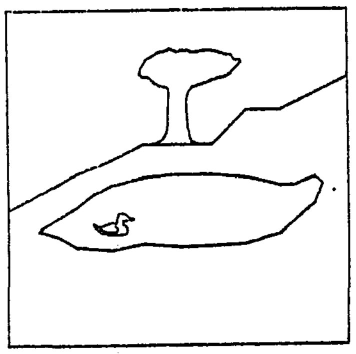

Figure 4-1

Suppose it is the duck that the user wants to move across the pond on successive frames; the position of the duck can be input by the operator on a keyboard and read by the program using a C language - "scanf" function. A sequence of frames with the duck in positions such as shown in Figure 4-2 might be generated.

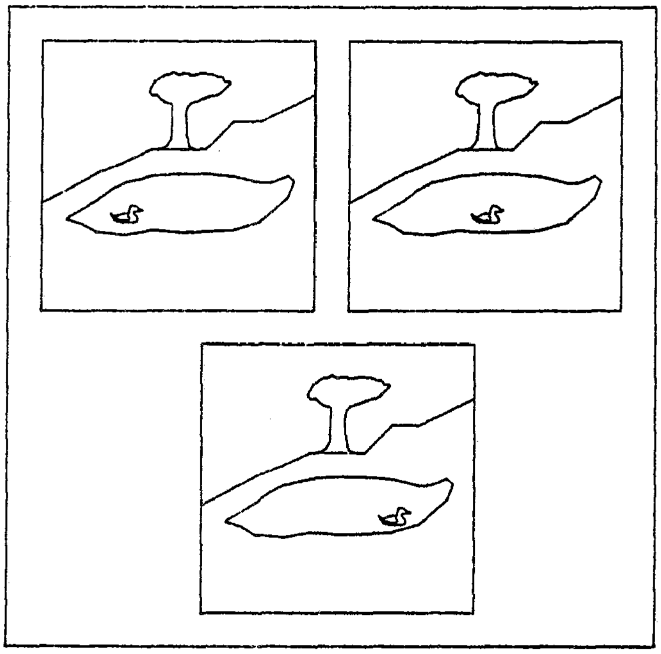

Figure 4-2

The outline of a program to do this may be:

```
        for (i = 1; ; i++)
        {
                scanf("%f %f", &x, &y);
                new_frame();
                draw_background();
                draw_duck(x, y);
        }
```

where the subroutine "draw_background" draws the outline of the tree, pond and landscape and the routine "draw_duck" draws the duck at the position (x, y). Note that "new_frame" clears the display but is not a GKS function.

This is, however, a very inefficient program because unnecessary work has to be performed. For example, normalization transformation has to be applied afresh to the background scene for each frame despite the fact that the background does not change from frame to frame. An obvious solution to this problem would be for GKS to store the background after the normalization transformation has been applied so that on each new frame it is only necessary to perform the minimum of computation to redraw the background on the display. This is of particular use when using display devices such as storage tubes or pen plotters.

However, more sophisticated devices, such as vector refresh displays and raster displays allow the user to selectively erase parts of the picture or move them about. Using such devices, the background can be drawn once with the duck in its initial position, then if necessary erase the duck and redraw it at the new specified position.

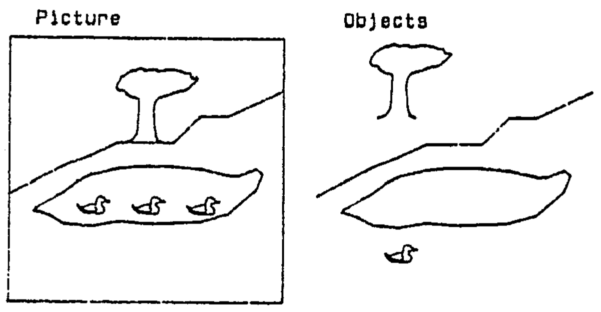

Figure 4-3

As another example, consider the left frame in Figure 4-3. The picture contains one tree, one pond, one landscape and three ducks. A program to draw such a picture may well be structured as four routine calls, each corresponding to one of the graphical objects (tree, pond, landscape and duck) where the routine to draw the duck would be parameterized on the position of the duck:

```
tree();
landscape();
pond();
for (i = 1; 1 <= 3; i++)
        draw_duck(x[i], y[i]);
```

This illustrates the correspondence between graphical objects and the picture. In particular, note that one object can have more than one instance in the picture.

In GKS, as mentioned earlier, output primitives can be grouped together into segments which are then associated with the output device. They can be manipulated in several ways as will be described later.

4.2. Creating and Deleting Segments

A segment is created by the function:

```
newseg(id)
```

where id is the integer identifier by which the segment is to be known. This segment is then the open segment. Subsequent calls to output primitive functions will result in those primitives being inserted into this open segment, as well as being output, until such time as the function:

```
        closeg()
```

is invoked. For example:

```
        newseg(1);
        duck();
        closeg();
```

will create a segment with the identifier '1' which contains the duck outline. It is important to note that at any time, only one segment can be open.

Once a segment is created, there are a number of operations which can be performed on segments. The simplest of these is deletion! A segment is deleted by calling the function;

```
        delseg(id)
```

where _id_ is the identifier of the segment to be deleted. After a segment has been deleted, the segment's identifier may be re-used.

4.3. Segment Attributes

4.3.1. Segment Transformation(*)

Earlier in this chapter, we discussed the need to move objects, or part of objects, around on a display in response either to some computation or to some operator request. _Segment transformation_ provides a way to do this efficiently. The segment transformation is a transformation operating on all coordinates in the segment definition. When a segment is created, the segment transformation is set to the null, or _identity_, transformation. The segment transformation can be subsequently changed by invoking the function:

```
        transeg(id, matrix)
```

where _id_ is the identifier of the segment whose transformation is to be changed. _Note_ that this identifier can be one of the open segments. The second parameter _matrix_ is a 2 x 3 transformation matrix.

Constructing transformation matrices can be a tricky task and thus GKS provides two utility functions to help the

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

application programmer. The first function:

```
        eval_transf_mat(fx, fy,tx, ty, r,
                        sx, sy, switch, matrix) (*)
```

calculates a transformation matrix which may then be passed directly into "transeg". In the above function fx and fy specify a fixed point around which the rotation and scaling takes place; tx and ty specify a translation or shift vector; r defines the angle of rotation in radians and sx and sy are scale factors. The resulting transformation matrix is returned via the parameter matrix. The coordinate switch, switch can take the value WC or NDC. With the former the values specified for the fixed point and shift vector are treated as being world coordinates, while in the latter case they are treated as normalized device coordinates.

If switch take the value WC, then the fixed point (fx, fy) and the shift vector (tx, ty) are transformed to normalized device coordinates by the current normalization transformation. The elements of the resulting transformation matrix are expressed in NDC. The transformation matrix is computed so that the order in which the transformations are applied to coordinate data is scale, rotate and shift. This is important because if rotation is applied before scaling different effects may be produced, for example in the case where scaling is not uniform. The function:

```
        accum_transf_mat(matin, fx, fy, tx, ty, r,
                         sx, sy, switch, matout) (*)
```

allows a more general transformation matrix to be composed. The parameters fx, fy, tx, ty, r, sx, sy, switch are the same as above. The resulting transformation defined by these parameters is concatenated with the transformation specified by matin and the resulting transformation matrix is returned via matout. The order in which the transformations are applied is input matrix, scale, rotate and shift.

One likely use of this function is to change the order in which the transformation is applied. This can be done by specifying each elementary (scale, rotate and shift) transformations separately using a call to "eval_transf_mat" followed by two calls to "accum_transf_mat" to build up the required matrix.

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

4.3.2. Segment Transformation and Clipping

The segment transformation is applied after the normalization transformation (mapping of world coordinates into normalized device coordinates) but before any clipping. Coordinates stored in segments are expressed in NDC. The elements of the transformation matrix which define the shift to be applied are also expressed in NDC.

A primitive in a segment is transformed by the normalization and segment transformations and then, if the clipping indicator is set to CLIP, is clipped against the viewport (of the normalization transformation) that was active when the primitive was put into the segment. Each segment contains, in effect, a clipping rectangle for each primitive it contains. However, the clipping rectangles are not transformed by the segment transformation.

4.3.3. Segment Visibility

The segment visibility attribute determines whether a segment is displayed or not. By default when a segment is created it is visible, thus:

```
newseg(1);
duck();
closeg();
```

will result in the duck picture being displayed. Normally the operator will see the picture being built on the display as each line is drawn. The visibility attribute may be changed by the function:

```
s_segvis(id, vis)
```

where vis specifies the visibility of segment id. vis may take the value TRUE for VISIBLE or FALSE for INVISIBLE. Hence, if the program is modified by the insertion of:

```
s_segvis(1, INVISIBLE)
```

after the call to "newseg", the duck will not be displayed on the screen. A subsequent invocation of:

```
s_segvis(1, VISIBLE)
```

will cause the duck to be made visible.

The visibility attribute is particularly useful for controlling the display of messages and menus. Typically each message will be put into a separate segment which is initially invisible. As a message is required, the segment containing it is made visible.

4.3.4. Segment Highlighting

Most vector refresh display hardware have the capability of highlighting a segment, eg. causing it to "flash" on the screen. The principle use of this facility is in drawing the operator's attention to some facet of the display. For example, in the animation program described above, when the operator has selected a new position at which the duck is to be displayed, he may highlight the duck segment in the new position and ask for confirmation that this is the position he really intends. The highlighting attribute is set by the function:

```
s_hilite(id, high)
```

where high specifies the highlighting for segment id. high may take the values TRUE for HIGHLIGHTED or FALSE for NORMAL. When a segment is created, the highlighting attribute is NORMAL and may be changed by the invocation of:

```
s_hilite(id, HIGHLIGHTED)
```

For a segment to appear highlighted, not only must the highlighting attribute have the value TRUE, but also the visibility attribute must have the value TRUE.

4.3.5. Segment Priority

When constructing complex pictures using segments quite often situations arise where segments or parts of segments overlap each other. Hence, there is a need to distinguish between segments in terms of priority of displaying segments or parts of segments when overlapping occurs. GKS provides a mechanism for doing this, segment priority, which can be set by the function:

```
s_segpri(id, priority)
```

where priority is a value in the range of 0.0 to 1.0, which specifies the priority of segment id. Thus, segment priority controls te order in which segments are drawn when the picture is changed.

There are a number of points which need to be stressed about segment priority because it is not as universally useful as it may at first sight appear to be. The first point is that segment priority only works for devices which have appropriate hardware capabilities. Secondly, the description of the "s_segpri" routine given above conveys the impression that it is possible to distinguish between an infinite number of segment priorities. In practice, the particular display being used is likely to have a finite range of priorities available, in which case the priority values specified is to be mapped onto this range. The final point to note is that if two segments of the same priority overlap, or if primitives within a given segment overlap, then the effects will be defined, but may vary from one implementation of GKS to another.

4.4. Renaming Segments

Segments can also be renamed by invoking the function:

```
renameseg(old, new)
```

where old and new are the old and new segment names respectively. Thus, if segment 1 contains the duck outline and we invoke:

```
renameseg(1, 2)
```

the duck outline will now be segment 2 and the segment name 1 can be reused for some other purpose. There are two points to note about the function. Firstly, there must be no segment called "new" (or "2" as in the case above) in existence when the function is invoked, otherwise an error results. Secondly, it is perfectly permissible to do the following:

```
newseg(1);
        .
<output primitives>
        .
        .
renameseg(1, 2);
        .
<output primitives>
        .
        .
closeg();
```

that is, renaming the open segment.

A typical usage of "renameseg" is the following. In the animation program at the start of this chapter, the operator wishes to determine a new position for the duck. Sometimes the placement is facilitated if the duck in the initial position is selected. However, the application program may wish to use a particular segment name for the duck, say 1. The duck would be created in segment 1 and a second duck in some other segment, say 2. When the new position has been determined by positioning segment 2, segment 1 is deleted and segment 2 is renamed segment 1. This shows that in some applications, a common technique is to use a set of segment names cyclically to hold a segment displayed in alternative positions. Once the desired position is found, the desired alternative is renamed and all others are deleted.

5. Graphical Input (*)

5.1. Introduction

So far, the primarily concern of this report has been with the output of graphical images. To a large extent, it has not been important to ask if someone had been looking at the results as they, (the images), were being generated. It was only in the last chapter that some interactive programming was done, and there, 'C's "scanf" function was used to input values to the program to control the graphical information being produced.

This chapter concentrates on the situation where a person sees an image being generated and reacts by rotating, moving, pressing or switching some piece of hardware in order to change the behaviour of the application program producing it.

Most displays will have hardware which allows graphical information to be entered directly to the system rather than always using the keyboard. For example, a storage tube will often have a pair of crosshairs to indicate positions on the display screen and these provide an alternative to the keyboard for the input of positional information. However, the variety of input hardware available is very great and hence, input data supplied from these are in many different forms. These different forms, therefore, must be organized in a uniform way, that does not depend on particular physical devices, to match the needs of application programs.

In GKS, the data that can be input to the program by the operator are divided into six different classes each of which have a logical input device associated with it. These six classes of logical input device in GKS are:

(1) locator: that inputs a position - an (X, Y) coordinate value.

(2) pick: that identifies a displayed object.

(3) choice: that selects from a set of alternatives.

(4) valuator: that inputs a numerical value.

(*)Because this implementation is a pre-released version of GKS, input facilities are restricted to REQUEST mode of interaction only. Also, due to the lack of graphical input hardware, these facilities have not been tested.

(5) string: that inputs a string of characters.

(6) stroke: that inputs a sequence of (X, Y) positions.

"Locator" and "pick" supply input data that is directly related to some feature of the display, while the others supply quite simple data not directly related to the display.

Note that these are logical input devices associated with the normalized device having the NDC unit square as its display area. Physical input devices are mapped into one or more of these logical device classes. The typical physical devices that are often used to map onto these logical devices are:

(1) locator: crosshairs and thumbwheels on a storage tube.

(2) pick: lightpen hitting an object dislayed on the screen.

(3) choice: button box or menu selection using a lightpen.

(4) valuator: potentiometers or input of values from a keyboard.

(5) string: keyboard input of a line of text.

(6) stroke: tablet input.

5.2. Request Mode

As well as the type of input to be read in by an application program, one also needs to specify when it takes place and under whose control. Discussion of the styles of interaction is deferred until a later section. For the present, the REQUEST mode of input is used which corresponds almost identically to the type of input encountered in the previous chapter.

In this mode, a program will REQUEST for an input from a logical input device and then waits while the logical input device takes over. Once the logical input device has obtained the required data, it returns it to the GKS program and effectively goes back to sleep. Hence, either the GKS program is executing or the logical input device is active but never both together (in REQUEST mode). One constraint this imposes is that, at any given time, no more than one logical input device may be active.

The form of the GKS function for REQUEST input is:

```
req_xxx(ws, dn, val)
```

The "xxx" part of the function call above defines the class of logical input device (locator, pick, etc) and the first two parameters specify the particular device of that class from which the input is requested; _ws_ specifies the workstation on which the logical input device is located and _dn_ specifies which of the devices of the class _ws_ it is. (This demonstrates that, it is possible to have more than one logical input device of a particular type on a workstation.) The resulting input value is returned to the program by the third parameter, _val_.

5.3. Locator

The LOCATOR logical input device returns a single position to the application program. The program may initiate this, in REQUEST mode, by calling the following GKS function:

```
req_loc(ws, dn, val)
```

This returns a position, via _val_, in WC along with the normalization transformation which is used to map the positional data back from NDC to WC.

Hence, the LOCATOR logical input device obtains a position in NDC space from the physical device and, using the normalization transformation, returns the positional data in terms of WC. However, it is possible that the picture being displayed has been constructed using a number of different window to viewport transformations. In this case, the logical input device then identifies which viewport the position lies within and uses this information to transform the position back to WC.

5.3.1. Overlapping Viewports

Requesting a locator position may return a position that is actually within more than one viewport due to overlapping viewports. In fact this always occurs, as normalization transformation 0 is defined to map the unit square in WC to the whole of the NDC space unit square (see Section 3.8). This ensures that there is always one viewport within which a locator input lies. In the case where two or more viewports overlap, one needs to define which normalization transformation is to be used to determine the mapping from NDC to WC.

In GKS, this is decided by associating a _viewport input priority_ with each transformation. If the input locator positions lies within more than one viewport, the viewport with the _highest_ viewport input priority is selected to perform the transformation back to WC.

The viewport input priorities are initially set up with normalization transformation 0 having the highest viewport priority so that all locator inputs are returned in NDC until the viewport input priority is changed. To change the viewport input priorities, the following GKS function is invoked:

```
s_vip(nt, rnt, hi)
```

This defines the viewport input priority of transformation _nt_ to be higher or lower than _rnt_ depending on the value of _hi_ which is set to TRUE for higher or FALSE otherwise.

5.4. Pick (*)

The PICK logical input device returns the name of a segment to the application program identifying the object that has been pointed to by the pick device. The program may initiate this form of input in REQUEST mode by calling the GKS function:

```
req_pick(ws, dn, val)
```

This returns, via _val_, the name of a segment and a more specific identification within the segment called the _pick identifier_.

5.4.1. Pick Identifier (*)

The pick identifier identifies the operator's action more closely within the segment by giving a second level of naming. The pick identifier is specified prior to the output of elements of a segment that may be "picked" by the GKS function:

```
s_pickid(n)
```

Once this function has been invoked, all subsequent output primitives will have a pick identifier value of _n_ associated with them until _n_ has been reset via another call to "s_pickid".

5.4.2. Segment Detectability

In Section 4.3, some segment attributes were described which may affect the appearance of segments. There is a further segment attribute, _segment detectability_, which determines whether or not a segment may be returned as a pick value. Segment detectability is set by the function:

```
s_detect(seg, det)
```

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

where det is the desired detectability set to TRUE for DETECTABLE or FALSE for UNDETECTABLE for the segment seg. A segment may only be picked if it is detectable. The default value for a segment at the time of its creation is undetectable. Making a detectable segment undetectable has no effect on its appearance. However, if a segment is invisible, it is also impossible to pick it. This reflects the behaviour of a physical device, such as a lightpen, which can only ´see´ images or objects that are visible.

5.5. Choice

The CHOICE logical input device returns an integer to the application program defining which of a number of possibilities has been chosen. The program may initiate this form of input, in REQUEST mode, by invoking the GKS function:

```
        req_cho(ws, dn, val)
```

where val returns the chosen integer. The actual realization of the CHOICE device on a physical device can vary quite significantly, from a menu hit by a lightpen, to a set of buttons to be pushed, to a name or number to be typed at a keyboard.

5.6. Valuator

The VALUATOR logical input device returns a real number to the application program. The program may initiate this, in REQUEST mode, by invoking the GKS function:

```
        req_val(ws, dn, val)
```

where val returns the numerical value (real number) that was input.

Clearly a single number could be input digit by digit from a keyboard without using a graphics system. However, if an interactive graphics system is in use, the operator may review the effects of a given input value without delay. For example, if the operator does not know, in advance, what the best value is, then several values must be input, with the operator reviewing each in turn until the best value is obtained. If successive values have to be input from a keyboard, the interactive dialogue is less likely to achieve an optimum value.

By contrast, "req_val" reads from whatever physical input hardware has been provided by a particular installation. Devices such as dials can continuously provide values and graphical echoes can, therefore, always be up to date.

5.7. String

The STRING logical input device returns a character string to the application program. In most cases the string is echoed character by character somewhere on the display. The program may initiate this, in REQUEST mode, by invoking the GKS function:

```
re_str(ws, dn, val)
```

where val returns the character string that was input. String input is most often used for unknown text strings such as choosing a filename or providing a title. However, selecting from a small number of strings already known to the application program, for example by using a menu,is actually closer to choice input.

5.8. Stroke (*)

The last logical input device, STROKE, returns a sequence of points to the application program, somewhat similar to a multiple "locator". The program may initiate this, in REQUEST mode, by invoking the GKS function:

```
req_str(ws, dn, val)
```

where val returns the sequence of points.

All points are transformed back using the same normalization transformation and lie within the window of that transformation. Viewport input priority is used to arbitrate between overlapping viewports in a way similar to "locator" input.

The particular implementation of STROKE input may provide a means of sieving out unwanted points. For instance, a particular implementation of STROKE may only select points occurring at selected time intervals. The idea in this case is to capture the data describing the motion of the operator's input, not just the points passed through. This is useful in animation and signature recognition, but is impossible to achieve if the individual points are input with "req_loc".

5.9. Styles of Interaction

5.9.1. Interaction Modes

In the previous sections, the different logical input devices were described using the REQUEST mode of input.

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

Attention is now focussed on the various different modes of input available in GKS and how they allow different styles of interaction to take place.

The interaction process, involving a logical input device, can be considered as taking place between two processes. One is the application program while the other is the input process, which looks after the input device and delivers data from a device to the application program. The relationship between the two processes can vary and, in doing so, will produce different styles of interaction.

There are three operating modes of logical input devices, each of which specify who (the operator or the application program) has the initiative, these being: SAMPLE input which is acquired directly by the application program; REQUEST input which is produced by the operator in direct response to the application program (as in the case of the "scanf" example using in Chapter 4) and EVENT input which is generated asynchronously by the operator.

These three modes of interaction work as follows:

(1) REQUEST mode: the application program and input process work alternatively. First the program requests an input and then waits for a response. The input process, started as a result of the request, delivers the resulting input to the application program and returns to the wait state. In the mode, one or the other process is active but not both together.

(2) SAMPLE mode: the application program and input process are both active together with the application program being the active partner. The input process works in the background providing the latest input data from the device which may or may not be used by the program. On the other hand, the application program continues executing, taking and using the current input data from a device when it is required.

(3) EVENT mode: the application program and input process are again active together but with the input process being the dominant partner. It delivers input data to the application program and expects the program to act, depending on the data received. The operator controls when input data is available and, effectively, drives the interaction.

All logical input devices can operate in each of the three modes. Of those in REQUEST mode, at most only one may active by responding to a request at a given time. However, any number may be active in SAMPLE mode at the same time, as with those in EVENT mode. This provides a rich environment for interaction within GKS.

5.9.2. Mode Setting (*)

Each device may be in only one mode of operation at a time. The mode is selected by calling a function of the type:

```
s_xxx_mode(ws, dn, mode, echo)
```

where "xxx", again is one of the six logical input devices. ws determines the workstation while dn determines device belonging to the class "xxx". The third parameter, mode, defines the mode of operation taking one of the three values REQUEST, SAMPLE, and EVENT, while the last parameter echo may take the value TRUE for ECHO or FALSE for NOECHO. If ECHO is enabled, the operator receives some indication of the input data that is being returned to the program.

The default mode of operating for all devices is REQUEST mode along with ECHO being enabled.

5.9.3. Request Mode

The call of "req_loc" causes the starting echo, the prompt, to appear on the screen if is echoing is selected. It is normally apparent to the operator which physical device is to be used. The operator moves the physical device (rotates a trackerball, moves a tablet stylus) which adjusts the screen echo. When satisfied with the position of the echo, the operator, then, activates a trigger to signal that the input is complete. (The physical trigger may be a button or a key or even the tipswitch of a tablet stylus, depending upon the particular logical input device.) This terminates the request locator interaction.

5.9.4. Sample Mode (*)

By contrast to requesting input, sampling returns a value immediately without waiting for a trigger. Sampling works for any of the logical input devices. A device (in this example a locator device) is put into SAMPLE mode by calling the function:

```
s_loc_mode(ws, dn, mode, echo)
```

where mode is set to SAMPLE. Unlike REQUEST mode, echoing begins immediately as the mode setting function starts up the input process to deliver values from the device. To SAMPLE from the locator device requires the following function to be invoked:

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

```
        spl_loc(ws, dn, val);
```

The SAMPLE functions return the current value of the locator device provided by the input process, via val. Although the SAMPLE mode is demonstrated with a locator device here, it must be noted that any of the logical input devices may be used in this mode, as with others.

5.9.5. Event Mode (*)

If an input device is put into EVENT mode, all input values, (or event, as they are called), are placed in the input queue for the application program to read from. The program reads each input value in order and deals with it before handling the next.

As the application program and input process are both active together, it is possible for the application program to look at the queue when no input have been made. Alternatively, it is possible for the input device to fill the queue completely before the application program examines the first item. This results in an overflow situation, which then causes an error message to be generated when the program attempts to remove the next event from the queue. Alternatively, the program may use an inquiry function (see Section 7.4) to detect input queue overflow.

Consequently, GKS contains functions to check the queue for events and to remove items from the queue.

It should be noted that there is a single input queue for all devices. And as it is possible to have several devices active in EVENT mode at the same time, all input from these devices are added to the queue as they are generated, together with information indicating which device produced the input. The GKS function to check on the queue is:

```
        await_event(timeout, ws, dclass, dv) (*)
```

The queue is examined to see if it is empty. If it is, the application program is suspended until an event is generated or a maximum of timeout seconds have elapsed. In the latter case, dclass returns NONE. timeout may be set to zero, in which case "await_event" always returns without suspending the execution of the application program.

If the queue is not empty, information concerning the first event (or the event at the head of the queue) is

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

returned. ws and dv identify the input device while dclass specifies the input class (locator, pick, etc). The input data are removed from the queue and transferred to the current event report.

Once it is known that an event has occurred from a particular class of device, the appropriate "get" function is called to read the input data from the current input report. To read locator input data, the function:

    get_loc(nt, val) (*)

is used, where is the transformation number and val contains the positional data (in the case of a locator event). Note that it is only necessary to return the data values specific to the device. The class of device from which the data originated is known via the information returned by the call to "await_event".

A third GKS function exists to provide a housekeeping function on the queue. Situations may arise where, due to erroneous input, the operator may need remove all events of a class, or all class. In GKS, the following function exists to remove all unwanted input from the queue:

    flush_queue(ws, dclass, dv) (*)

When called, it removes all input in the queue from the device whose class is dclass and which is specified by ws and dv.

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

6. Workstations

6.1. Introduction

It is now time to consider real devices. So far, the application program have used world coordinates which were mapped on to a hypothetical normalized device having a display surface with a range 0 to 1 in both X and Y directions with the origin being at the lower left hand corner. Instances of output primitives of the same type have been differentiated, so far, by associating different values of primitive indices with specific instances of primitives. However, the realization of how these primitives are differentiated on real devices have not been discussed.

In GKS, there is a clear distinction between definitions of a graphical application in a virtual or device independent way and the part which describes how the program will be interfaced to the specific devices available at a particular installation.

To understand this distinction, consider an example to illustrate picture composition using normalized device coordinates. The scene to be generated consists of a house, tree and a duck each of which is defined with its own world coordinate system. Data defined in chapter 2 will be used to describe the duck. The coordinates to define the tree are as follows:

```
Tre->w_x = 14.0  18.0  21.0  ....   40.0  42.0
Tre->w_y = 10.0  11.0  13.0  ....   11.0  10.0
```

The house with a window and a door could be defined by:

```
Hse->w_x = 10.0  10.0  50.0  90.0  90.0  10.0
Hse->w_y =  0.0  60.0  90.0  60.0   0.0   0.0

Wnd->w_x = 30.0  30.0  70.0  70.0  30.0
Wnd->w_y = 40.0  60.0  60.0  40.0  40.0

Dor->w_x = 40.0  40.0  60.0  60.0  40.0
Dor->w_y =  0.0  30.0  30.0   0.0   0.0
```

Composing the picture in NDC space can be done by:

```
Wrect wrectl = {0.00, 0.00, 30.0, 20.0};
Nrect vrectl = {0.25, 0.05, 0.55, 0.25};
s_window(1, &wrectl);
s_viewport(1, &vrectl);

Wrect2 = {0.00, 0.00, 70.0, 70.0};
Nrect wrect2 = {0.05, 0.05, 0.40, 0.40};
s_window(2, &wrect2);
s_viewport(2, &vrect2);

Wrect wrect3 = {0.00, 0.00, 10.0, 10.0};
Nrect wrect3 = {0.55, 0.05, 0.90, 0.75};
s_window(3, &wrect3);
s_viewport(3, &vrect3);


sel_ntran(1);
s_pl_i(1);
duck();

sel_ntran(2);
s_pl_i(2);
polyline(58, Tre);

sel_ntran(3);
s_pl_i(1);
polyline(6, Hse);
s_pl_i(2);
polyline(5, Wnd);
s_pl_i(3);
polyline(5, Dor);
```

This example shows that normalization transformation 1 maps the window defined by coordinates (0,0) and (30,20) in WC space onto the viewport defined by (0.25,0.05) and (0.55,0.25) in NDC space while transformation 2 maps the window defined by (0,0) and (70,70) in WC space to the viewport defined by (0.05,0.05) and (0.40,0.40) in NDC. Similarly, transformation 3 maps the window (0,0) to (10,10) onto the viewport (0.55,0.05) to (0.90,0.75) in NDC space. Said in another way, the picture generated within the coordinates (0,0) to (30,20) in WC space will be reproduced within the coordinates (0.25,0.05) to (0.55,0.25) in NDC space by the first transformation, while the picture within WC (0,0) and (70,70) will be reproduced within the NDC (0.05,0.05) to (0.4,0.4) by the second transformation, and so on. As before, the point of origin is the lower left hand corner.

Hence, selecting the normalization transformation 1, the duck is drawn using polyline representation indicated by the index 1. Following this, the second normalization transformation is selected along with the polyline representation 2 to draw the tree. Finally, the third normalization transformation is selected to draw the house using the various different polyline representations. Schematically, the picture is shown in Figure 6-1. Labels have been added to the polylines to indicate the polyline index associated with each one.

Now consider how the picture that has been composed can be output to a real device and what will be the appearance of the various polylines on the display or plotter.

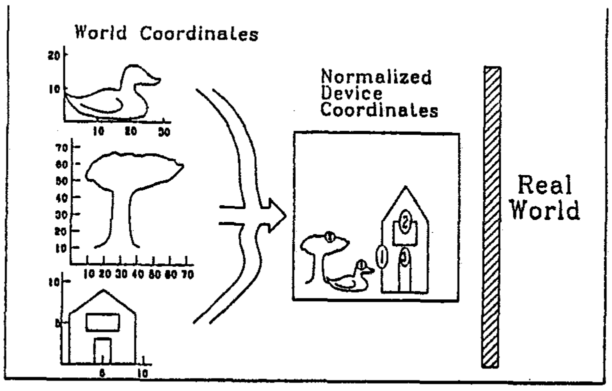

Figure 6-1

6.2. Workstations

GKS has been defined so that it is equally applicable as a graphics system in a wide range of different environments. Depending on whether the application environment is simple or complex, an operator may be working on one device or using a number of devices. For example, GKS might be used for off-line graphical output to a remote plotter or it might be used in a very simple interactive mode on a storage tube, with text commands from the keyboard causing pictures to be displayed on the screen. At the other end of the usage spectrum, an operator may have a refresh display with lightpen, tablet, keyboard and button box together with a small plotter and a large digitizer to provide a sophisticated workbench for a Computer Aided Design (CAD) application. The aim of GKS is that all these environments should be catered for in a similar way while still allowing the application program to make the best use of the total environment available and of the specific characteristics of each device.

Instead of individual input and output devices, GKS has introduced the concept of workstation, which in simple environments closely approximates a graphical display and associated input peripherals attached via a single line to the host computer. The main characteristic of a workstation is that it is to have at most a single display surface for graphical output. However, it may also have a number of input devices associated with it. Thus, for example, a storage tube with keyboard and thumbwheels would be regarded as one workstation in GKS. It is not necessary, however, for a workstation to be capable of both input and output. Consequently, both digitizers and plotters can be regarded as separate workstations. In this case, they are classified as INPUT and OUTPUT workstations respectively. Workstations capable of doing both output and input are classified as OUTIN workstations. OUTPUT, INPUT and OUTIN are referred to as workstation categories.

If an operator has a complex workbench involving a pair of displays or a display and plotter, this environment cannot be described to GKS as a single workstation. It is described as several workstations clustered together under the control of one operator and one application program. It is possible that the same device may be configured as part of more than one workstation type. It is therefore necessary in GKS to allow several workstations to be used together.

6.3. Defining Workstations To Be Used

So far, the output facilities and coordinate systems of GKS have been explained along with their usage in various situations. However, if graphical information were to be generated using the output primitives discussed in chapter 2, GKS would return an error message and terminate execution rather than displaying the picture. This is because no workstation (output device) has been specificed, opened or activated. Each application program, therefore, must identify the workstation(s) that it is to use, indicate their categories and identify when it is in use. If several workstations are used together, they will most likely be connected to the computer over different lines. The function used to specify a particular workstation to be used is:

```
open_ws(ifile, ofile, ws)
```

The first two parameters, ifile and ofile identify files that may be used in conjunction with the workstation. As the names suggest, ifile identifies the input file from which the program is to obtain its input while ofile specifies the file name to which all subsequent output is to be sent. Their usage is illustrated in situations where Metafiles are being used either to read from or to store graphical information to. If no external files are being used, the above function may be calling using "NULL" as the first and second arguments. ws specifies the name of the workstation to be used. As the user may be require several workstations to complete the task, the "open_ws" function must be called separately for each workstation being used specifying their _ws_'s values.

So far, even if the above function was invoked one or more times, for each workstation to be used, the workstation(s) would have been opened but none would have been activated. Consequently, if some output primitives were created, GKS would exit with an error indicating that no workstations are active. To activate a workstation so that it can receive output, the function:

```
activate(ws)
```

is called. _ws_ specifies the workstation to be activated and it is possible to have one or more workstations active while others are left open. From the time that a workstation is activated, all output is sent to its associated device for display. It is possible for several devices to be active together in which case they all will receive the graphical output. As the program executes, workstations may be activated and deactivated so that only some fraction of the complete output appears on each workstation. Deactivation of a workstation is performed by the function:

```
deactivate(ws)
```

Before creating output, it is sometimes necessary to move to the next frame on film or to the next sheet on the plotter or to clear the screen of a display. For example, it is sensible to move to the next sheet on the plotter only if the previous sheet has been used. GKS provides both these alternatives with the function:

```
clear(ws, flag)
```

where _flag_ may take the values FALSE for CONDITIONALLY or TRUE for ALWAYS . If _flag_ has the former, ie. CONDITIONALLY, the screen is cleared only if something is drawn. If, however, the value is the latter, the screen is cleared unconditionally (ie. on every occasion).

As an example, the program below outputs a set of frames to the first workstation while only outputting every 10th frame to the second workstation.

```
        open_ws(NULL, NULL, tek);
        open_ws(NULL, NULL, ser);

        activate(tek);

        for (i = 1; i <= 100; i++)
        {
                j = i % 10;
                if (!j)
                        activate(ser);
                polyline(15, pts);
                clear(tek, ALWAYS);

                if (!j)
                {
                        clear(ser, CONDITIONALLY);
                        deactivate(ser);
                }
        }
```

At the end of the program any workstation still active should be deactivated and all workstations should be closed down using the function:

```
        close_ws(ws)
```

Thus the above program should be terminated by:

```
        deactivate(tek);
        close_ws(ser);
        close_ws(tek);
```

This will ensure that all output has been transferred to the devices. Following this the files used are closed in an orderly manner and "clean" disconnections are made from all the workstations in use.

6.4. Workstation Transformations

The section above defines when output may be sent to a workstation and, therefore, to its associated display screen. As described at the beginning of this chapter, a picture has been composed in NDC space but it must be output to active workstations. The simplest approach would have been to insist that all devices viewed the whole of the NDC space within the unit square. This is what happens by default. However, there are distinct advantages if some control is allowed over what part of NDC space is to be seen on a particular device.

A good method of visualizing what is going on is to use the analogy of a camera focussed on the normalized device coordinate space. Each workstation can focus on a specific part of the complete square and can zoom in or out. Consequently, only a part of the output to NDC space may appear on a display screen.

This is shown schematically in Figure 6-2. In this example, the first workstation is focussed on the trunk of the tree while the second is focussed on the house. The method of defining which part of NDC space is to appear on the workstation is very similar to the way positioning of output on NDC space is achieved. A workstation window to viewport mapping defines which part on NDC space will be seen at the workstation and where it will appear on the display screen.

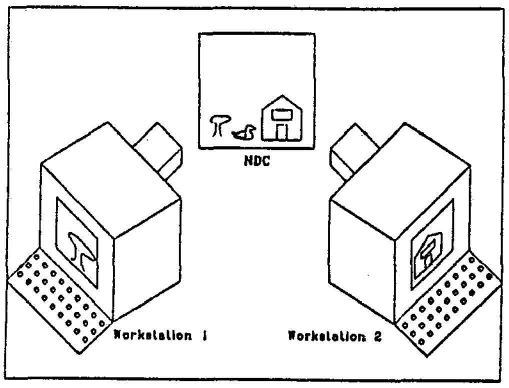

Figure 6-2

The two functions are:

```
s_w_wind(ws, nrect)
s_w_view(ws, drect)
```

The "s_w_wind" function specifies that area of the NDC space, (contained as the lower left and upper right corner points), in the range 0 to 1 which will be output to workstation ws. The window is specified in NDC coordinates.

The "s_w_view" function specifies where on the display screen the view of NDC space will appear (again as the lower left and upper right corner points). The viewport is specified in the appropriate device coordinates.

However, unlike the normalization transformation, there is only one workstation transformation (per workstation) which is used for the whole picture. If the workstation transformation is changed, the whole picture is displayed with the new transformation.

There is a major difference between the normalization transformation used to compose the picture in NDC space and the workstation transformation defined above to view the picture on the display. Whereas the normalization transformation allows differential scaling, this is not the case for the workstation transformation. Only normal scaling is permitted, hence the aspect ratio of the picture composed in NDC does not change when reconstructed on a device. In the example above (see Figure 6-1), the house is elongated in the Y direction as it is defined in NDC space by the normalization transformation. Displaying this picture on a device will still show the elongation in the Y direction even if one displays blown up parts (zoomed) of the house. Thus, the aspect ratio is preserved.

This raises the question of what happens if the aspect ratio of the workstation window and viewport are defined differently. In this case, the complete window is displayed on the device in the correct aspect ratio using a rectangle with its bottom left corner equal to the bottom left corner of the viewport specified and is as large as possible. Some examples in Figure 6-3 show areas of the viewports used when the window and viewport have different aspect ratios.

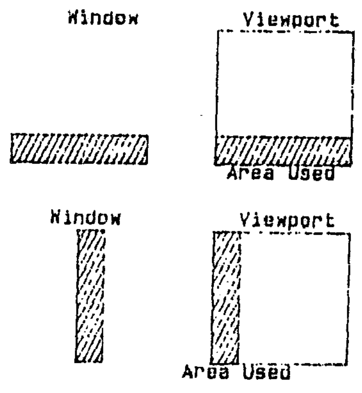

Figure 6-3

Once the window is mapped onto the viewport, the unused parts of the display surface cannot be used for subsequent displays unless the whole picture is reconstructed using a more compatible aspect ratio. To avoid confusion, therefore, it is good practice to always specify the workstation window and viewport with the same aspect ratio.

Default settings are available for each workstation and, if not changed, these will map the whole unit square of the NDC on to the largest square on the device with the lower left corner of NDC space at the lower left corner of the dispay screen. Specific default settings, for different display aspect ratios are shown in Figure 6-4.

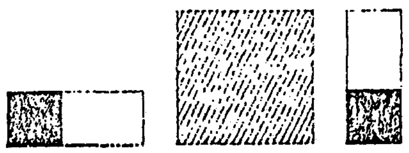
Area of display used as default
viewport equivalent to NDC space

Figure 6-4

6.5. Polyline Representation

For polylines, GKS identifies three aspects that a workstation could use to differentiate between polylines, these being the linetype, linewidth and colour. A particular workstation may use one of these aspects or a combination of them to ensure that polylines with different index values are differentiated on display. Not all of these possibilities may exist on a particular workstation, so it is up to the application program to define the mapping unless the defaults provided by the local installation are accepted. For each polyline index value, its representation on a particular workstation is defined by the function:

```
pl_rep(ws, pli, &lr)
```

The ws parameter specifies the workstation in question while pli specifies which polyline index is being defined. The third parameter specifies the address of the representation associated with the polyline index pli and has the following format:

```
lr.lr_ix        /* representation index */
lr.lr_lt        /* linetype     */
lr.lr_lw        /* linewidth    */
lr.lr_ci        /* colour index */
```

"lr.lr_ix" specifies the representation number and is the same as pli. "lr.lr_lt" specifies the linetype, for this representation and can have one of the following possible values are:

GKS linetype	Device Representation
SOLIDL	solid line
DOTTED	dotted line
DASHED	dashed line
DOTDASHED	dotted - dashed line

Other values for this parameter may be defined at a later stage.

"lr.lr_lw" specifies the linewidth scale factor. This is a real number giving the width of the line relative to the width of a standard line on this device. Values less than 1 specify lines thinner than the standard line. The workstation will use the closest available linewidth to the specified one when drawing the line.

"lr.lr_ci" specifies the polyline colour factor. This does not define a colour directly but instead points to a colour table for the workstation where the colours are actually specified as Red-Green-Blue intensity values. For pen plotters, it will be usual for the colour table to be set up as follows:

Colour Index	Colour Pen
0	White
1	Black
2	Red
3	Blue
4	Green
5	Yellow
6	Orange
7	Brown
8	Grey

If colour is unavailable on a device, it is usual for all the colour indices to point to the the foreground colour of the device with the exception of 0 which points to the background colour (if possible).

GKS differentiates different polyline index values on all devices thus providing true device independence. If necessary, the user can define the representations corresponding to the index values so that they appear as nearly the same as possible on all devices that are used. GKS gives the user the opportunity to define the mapping, unlike many systems where the mapping is made within the system.

An expert user defining a system to run with several simultaneous workstations has to specify the appearance of primitives on each workstation. However, it can be seen that the description of the graphics program in terms of the world coordinates defines one part of the system and this can be kept quite independent of the other part defining the realization of the system on particular devices.

6.6. Colour Table

For a colour display, it is possible to redefine the entries in the colour look up table pointed to by the colour index. Each entry in the table consists of:

Colour	Red	Green	Blue
index	intensity	intensity	intensity

The intensity values are specified in the range 0.0 to 1.0. Particular entries in the table can be redefined by:

```
s_col_rep(ws, ci, &col)
```

where ci is the colour index and col specifies the address of the representation being specified. This consists of a structure defining Red, Green and Blue intensities expressed in real numbers.

It should be noted that, when other primitives are described there is a single colour table for each workstation. If polyline index values and text index values use the same colour index in their representation, the same colour will be used as they both point to the same colour table.

6.7. Polymarker Representation

The aspects of the polymarker primitive controlled by the workstation via the polymarker index are defined by the function:

```
mk_rep(ws, pmi, &mr)
```

In a similar way to polyline, ws specifies the particular workstation while pmi specifies which polymarker index is being defined. mr specifies the address of the representation associated with the polymarker index pmi and has the following format:

```
mr.mr_ix        /* representation index */
mr.mr_mt        /* marker type  */
mr.mr_ms        /* marker size  */
mr.mr_ci        /* colour index */
```

"mr.mr_ci" is the polymarker colour index, which is treated in the same way as the polyline colour index.

"mr.mr_mt" specifies the marker type, which defines the form of the marker to be displayed. The possible marker types are:

Marker Type	Marker Form
1	.
2	+
3	*
4	O
5	X

(Note: these are not symbols (ie. characters) but markers which in most cases are drawn using polylines.) The first marker type defines the smallest displayable point while the other markers have a size that is workstation dependent. Marker types greater than 5 may be defined for a particular workstation at some later stage.

"mr.mr_ms" specifies the marker scale factor, which is a real number giving the size of the marker relative to the workstation standard marker size (size depict by the marker scale factor of 1). A value of 2 will specify a marker twice the size of the standard. The workstation will output markers as close to this as possible.

6.8. FillArea Representation

The aspects of the fillarea primitive, for drawing of enclosed areas, is controlled by the workstation via the fillarea index are defined by the function:

```
fa_rep(ws, fai, &fa)
```

where as before, ws specifies the workstation while fai specifies the fillarea representation being defined. fa specifies the address of the representation associated with the fillarea index fai and has the following format:

```
fa.fa_ix        /* representation index */
fa.fa_is        /* interior style */
fa.fa_si        /* style index  */
fa.fa_ci        /* colour index */
```

"fa.fa_ci" specifies the fillarea colour index to be used. "fa.fa_is" and "fa.fa_si" specify the fillarea interior style and fillarea style index respectively.

The possible values for the fillarea interior style are HOLLOW, SOLID, PATTERN and HATCH. (*)

(*)Not all workstations provide all interior styles and it will be necessary for the user to consult the local installation manual to decide what is available. Only

HOLLOW will cause the boundary to be drawn in a colour defined by the colour index. SOLID will cause the interior to be completely covered using the colour selected via the colour index specified.

PATTERN will shade the area by repeating a pattern, using the size and positioning defined in a workstation independent way by the pattern size and pattern reference point attributes. These attributes are by the respective functions:

```
s_patsiz(sz)
s_patref(rp)
```

where sz, consisting of two fields sz->w_x and sz->w_y, is the size of each instance of the pattern in the X and Y directions and rp, consisting of rp->w_x and rp->w_y, is the coordinate of the pattern reference point. Note that for the interior style the colour index is not used.

Entries in the pattern table may be already preset by the installation manager or they may be set by the user using:

```
s_patpt(ws, pi, nx, ny, na)
```

where pi specifies the pattern index associated with the entry to be defined. nx and ny specify the number of cells of the pattern in the X and Y directions while na list, which is nx long, specifies the colour table index for each cell.

Note that the colour index of the fillarea representation is used for the SOLID, HOLLOW and HATCH interior styles but only for PATTERN interior style . The colours in the pattern are derived from the elements of the pattern array.

The fourth method of shading the fillarea is by interior style HATCH. In this case, the style index defines the different styles of hatching available on the workstation being used.

6.9. Text Representation

Text is the most complex of GKS primitives and certainly the one with the most attributes (eg. height, writing direction, alignment and orientation), all of which could be defined in a device independent way. The appearance of a text string in terms of its overall size and shape is defined in NDC space. For example:

style HOLLOW should always be provided.

```
Wrect   wrectl = {0.0, 0.0, 100.0, 100.0};
Nrect   vrectl = {0.0, 0.0, 1.0,   1.0};

s_window(1, &wrect);
s_viewport(1, &vrect);
sel_cntran(1);

s_ch_ht(1.5);
ang->w_x = 1.0;
ang->w_y = 1.0;
s_ch_up(ang);
s_tx_i(1);

pts->w_x = 50.0
pts->w_y = 50.0
text("Example", pts);
```

This defines output on NDC space of the form given in Figure 6-5.

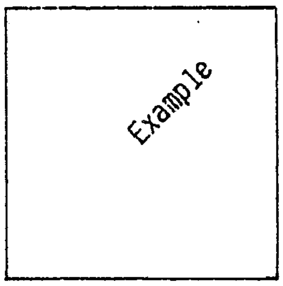

Figure 6-5

However, a number of characteristics associated with the text string have not been set globally but are controlled by the text index in the same way that linetype, linewidth and colour are controlled by the polyline index in the workstation. The text aspects controlled by the workstation independent text index are defined by the function:

```
tx_rep(ws, ti, &tr)
```

where as before <u>ws</u> specifies the workstation, <u>ti</u> the text representation <u>index</u> (and is the same as "tr.tx_ix") and <u>&tr</u> the address of the representation itself having the following format:

```
tr.tx_ix        /* representation index         */
tr.tx_fp        /* text font and precision      */
tr.tx_chef      /* character expansion factor   */
tr.tx_chsp      /* character spacing            */
tr.tx_ci        /* text colour index            */
```

Text font and precision (tr.tx_fp) (*): a workstation may have access to a number of different fonts. The index defines the number of the font to be used, if this is possible, while the precision specifies how well the font type can describe the output in NDC space.The three possible precision values are:

(1) STRING: of the global attributes only character height need be used. The minimum requirement is that the text string will be drawn horizontally left to right starting from the position as close to the position specified as possible.

(2) CHAR: the global attributes are used to define the position of the individual characters accurately. The characters may still be horizontal.

(3) STROKE: the font will display the text string precisely using all attributes.

Character expansion factor (tr.tx_chef): the assumed width of the characters relative to the height can be increased or decreased by redefining the character expansion factor.

Character spacing: the character spacing can be used to increase or decrease the space between characters. It is defined in terms of the character height.

Text colour index (tr.tx_ci): as for the other primitive colour indices, this is a pointer to an entry in the colour table defining the colour to be used.

If two workstations "ser" and "hpp" are active with text index 1 specified by

(*)Present implementation restricts the use of text font to those provided by the workstation hardware themselves. As a result of this, text precision is limited to STRING only.

```
          ser                          hpp
          ---                          ---
tr.tx_ix = 1;                tr.tx_ix = 1;
tr.tx_fp.ft = 2;             tr.tx_fp.ft = 2;
tr.tx_fp.pr = STROKE;        tr.tx_fp.ft = STRING;
tr.tx_chef  = 2.0;           tr.tx_chef  = 1.0;
tr.tx_chsp  = 0.5;           tr.tx_chsp  = 0.1;
tr.tx_ci    = 1;             tr.tx_ci    = 1;

tx_rep(ser, 1, &tr);         tx_rep(hpp, 1, &tr);
```

the output on the two workstations may be shown in Figure 6-6. Thus, unlike polyline, text is output by the workstation in significantly different ways depending on the hardware facilities supported.

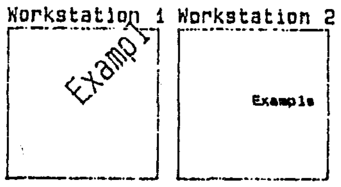

Figure 6-6

6.10. Segment Storage on Workstations

Conceptually, segments are stored in the active workstations when the segment is created. A particular implementation of GKS could store segments centrally but, for intelligent workstations with local storage, the aim would be to store the segments in the workstation to improve the performance seen by the user in terms of segment manipulation. Even if stored centrally, the implementation must be such that the user can imagine that segments are stored on the workstation. For example:

```
open_ws(tek);
open_ws(ser);
activate(tek);

newseg(8);
pond();
closeg();

activate(ser);
newseg(9);
duck();
closeg();

deactivate(tek);

newseg(10);
tree();
closeg();

zapseg(9);

deactivate(ser);
```

The pond (segment 8) would be stored on workstation "tek", while the duck (segment 9) would be stored on both "tek" and "ser". The tree (segment 10) is only stored on "ser" as "tek" was deactivated by the time segment 10 is defined. Deleting a segment removes all knowledge of it from the system so deleting segment 9 will delete the duck from both workstations even though the deletion takes place when workstation "tek" is inactive.

The function for manipulating a segment, by assigning a value to a segment attribute, affects all copies of the specified segment. It is possible to delete a segment from a single workstation by calling:

```
delseg(ws, id)
```

The effect is as if workstation _ws_ had not been active when the segment _id_ had been created.
The function

```
clear(ws)
```

deletes all segments stored in the specified workstation.
The function:

```
redraw(ws)
```

can be used to clear the display surface of the specified workstation without deleting the segments stored from it.

Simple uses of GKS are likely to have nearly the same set of segments defined on each workstation. Some skill is required to define segments on a specified subset of workstations that are open. It would normally only be used in complex programs.

Closing down a workstation deletes from that workstation all the segments currently defined in it. If it is reopened, it will be initialized with an empty segment storage.

If a segment is deleted from each workstation on which it is stored, either by using "delseg" or by closing the workstation, then the segment itself is deleted as if "zapseg" had been invoked.

6.11. Deferring Picture Changes (*)

If an intelligent refresh display is being used as a workstation, the operator would expect that picture changes would take place immediately. For example, if a polyline is output by the program, the operator would expect to see the polyline image on the display screen immediately. However, if a less powerful system is used or if there are inefficiencies in the link between the host computer running GKS and the display, this may not be possible and may, in some cases, not be desirable due to the high overhead it places on the system.

Two examples where it may be undesirable to update the picture immediately are an off-line plotter and a storage tube connected to the host via a network. In the first case, output for the plotter may be accumulated as records on a magnetic tape. Rather than output a record for each primitive output, it would be more efficient to buffer the output into reasonable size blocks before sending to the magnetic tape. In the second case, sending small packets of output across the network is inefficient and it would be more of value to save them up.

There are occasions, however, particularly in an interactive mode of working, where, if the display is not up to date, it may be difficult for the operator to continue. An example might be where the operator is defining coordinates to be joined by lines. In this case, if the last position does not have the associated line drawn, it becomes difficult for the operator to know where the starting point of the next line is.

GKS allows the user to specify the mode (deferral mode)

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

in which a particular workstation should perform in terms of updating the display when modifications are made to the already present display. The possible deferral mode are:

ASAP:
: the effect of the visual change must be achieved As Soon As Possible.

BNIG:
: the effect of the visual change must be achieved Before the Next Interaction Globally - that is on any workstation currently open. Thus, the display screens may not always be up to date but they are always made current as soon as input is requested from any workstation.

BNIL:
: the effect of the visual change must be achieved Before the Next Interaction Locally - that is on this workstation. Thus, the display screens may not always be up to date but it is always made current as soon as input is requested from this workstation.

ASTI:
: the effect of the visual change must be At Some TIme but there is no guarantee when this will be.

Another source of inefficiency on some displays is that certain functions cause <u>implicit</u> <u>regeneration</u> of all displayed information. For example, changing the representation on a workstation, for a particular polyline index already used in the picture, should cause all polylines with that index as attribute to take on the new representation. On a storage tube, as well as a pen plotter, this could only be done if the whole picture on the screen was redrawn, while on other displays, such as refresh displays, this could be performed immediately.

Another example where implicit regenerations are caused is segment manipulation. Deleting a segment from a picture or changing its transformation should have an immediate effect but can only be achieved on a storage tube workstation by redrawing.

GKS recognizes that implicit regeneration may be inefficient on some workstations and allows the user to delay such picture changes if it is not a problem in one of two modes:

SUPPRESSED:
: implicit regenerations of the picture are suppressed.

ALLOWED:
: implicit regenerations of the picture are performed.

An implicit regeneration consists of clearing the display surface and redrawing all the segments stored on the workstation. This may result in a loss of information if parts of the picture are not stored in segments.

The user specifies the setting of the deferral and implicit regeneration modes by :

```
        s_def(ws, dm, igm)
```

where dm and igm are the deferral mode and implicit regeneration mode for the workstation ws. If the deferral mode has a value other than ASAP or the implicit regeneration mode has the value SUPPRESSED, it must be possible to force the display to be current and that is achieved by the function:

```
        update(ws, regen)
```

where regen is the regeneration flag which may take the values TRUE for PERFORM or FALSE for SUPPRESS. The function performs all deferred actions, but only does an implicit regeneration if one is necessary and the regeneration flag is PERFORM. In contrast, the function "redraw(ws)" performs all the deferred actions, clears the display surface and redraws all the segments stored on this workstation unconditionally.

It is anticipated that refresh displays would normally work in the modes (ASAP, ALLOWED) while local storage tubes might work in the mode (BNIL, SUPPRESSED). For an off-line plotter, the mode would most likely be (ASTI, SUPPRESSED).

6.12. Input Devices (*)

The previous chapter, Chapter 5, describes the logical input devices defined by GKS in the virtual world. Thus the application program defines its output in terms of the output primitives and also defines input to the program using the logical input devices Locator, Stroke, Valuator, Choice, Pick and String. GKS ensures that the operator of an OUTIN workstation has at least one logical input device of each type available to him. If the operator uses a single workstation, this implies that the workstation must be capable of providing inputs of each type.

A particular installation when it defines workstation types must indicate which logical input devices they support and how they are achieved. It is quite possible for different installations to define the same logical input device on a GKS workstation in significantly different ways.

The implementation of the logical input devices is much simpler if the installation supports REQUEST input only, since only one input device may be active at any one time. It is then unnecessary for the physical device to include a

(*)Due to the unavailability of graphical input devices at this is installation, this facility is presently restricted to input using keyboards which are read in via 'C's input functions.

device identification with the input data.

7. GKS Environment

7.1. Initialization

Nothing has been said, so far, as to how the user initializes GKS at the start of the program. In fact, none of the GKS functions so far described may be called before GKS is initialized and this is done by the function:

```
open_gks()
```

The function "open_gks" is called once only by an application program for an invocation of GKS.

Opening of GKS initializes a number of variables to default values, some of which are implementation specific, for example, the maximum number of workstations that may be opened simultaneously may vary from implementation to implementation. Others are defined as standard for all implementation. For example, GKS always sets the current "polyline index" and other similar indices to 1. The current normalization transformation number is set to 0 and all normalization transformations are initialized so that both the window and the viewport are the unit square with the origin being the lower left-hand corner. For example:

```
open_gks();
open_ws(ser, 6, 3);
activate(ser);

p2 = pts;
p2->w_x = 0.0;     p2->w_y = 0.0;     p2++;
p2->w_x = 1.0;     p2->w_y = 1.0;
polyline(2, pts);
```

This will draw a single line from the lower left corner to the upper right corner of the NDC unit square. The line will be drawn in the style specified by polyline index 1.

Just as GKS needs to be initialized before any calls can be made to GKS functions, it is also necessary to close GKS down at the end of session. This is achieved by calling the function:

```
close_gks()
```

The error file is closed and other housekeeping functions required by the implementation are performed. The above program should, therefore, be terminated by :

```
deactivate(ser);
close_ws(ser);
close_gks();
```

Note that it is important to deactivate and close all workstations that are open prior to closing GKS. The first step of deactivating all active workstations is necessary as this will terminate the existence of their "workstation state lists" (see below) while the second step of closing all open workstations is needed to terminate the links established between the workstations and GKS thus releasing the physical devices themselves.

7.2. GKS Operating States

To aid error handling in GKS, a number of operating states are identified. When GKS is in a specified state, certain GKS functions can be called while others are invalid. Prior to GKS being opened, it is in the state "GKS CLOSED". As soon as the function OPEN GKS has been invoked, GKS moves to the state "GKS OPEN". The five states of GKS are:

(1) GKS CLOSED (GKCL): the initial state of any application program demonstrating that GKS has not been initialized as yet and hence cannot be used until initialization has been achieved. It is also the final state of an application program showing that the use of the system has been completed.

(2) GKS OPEN (GKOP): the result of a call to "open_gks". Once this state is achieved, initialization of all GKS variables, attributes, State and Description lists occurs. To revert back to GKCL a call to "close_gks" is needed.

(3) WORKSTATION OPEN (WKOP): at least one workstation has been opened as a result of a call to "open_ws". In this state, certain attribute setting functions can be called upon by the application program. To revert back to GKOP a call must be made to "close_ws" for each workstation that is currently open.

(4) WORKSTATION ACTIVE (WKAC): at least one of the workstations is active as a result of a call to "activate". In this state, GKS output primitives can be created and certain segment manipulation functions can be performed. To revert back to WKOP a call must be made to "deactivate" for each workstation that is currently active.

(5) SEGMENT OPEN (SGOP): a segment is opened. This state is reached by creating either a new segment on an activate workstation or opening an existing one. Calling "closeg" cause GKS to revert to the former state WKAC.

There is a strict ordering of the states as shown in Figure 7-1 below. For example, the state WORKSTATION OPEN cannot be changes to "SEGMENT OPEN" directly - GKS has to go through the state "WORKSTATION ACTIVE". Effectively, this ensures that a segment cannot be created or opened unless at least one workstation is active. Similarly, GKS cannot revert directly from the state "WORKSTATION ACTIVE" to "GKS OPEN". This prevents a workstation from being closed if it is still active; the program must first deactivate it.

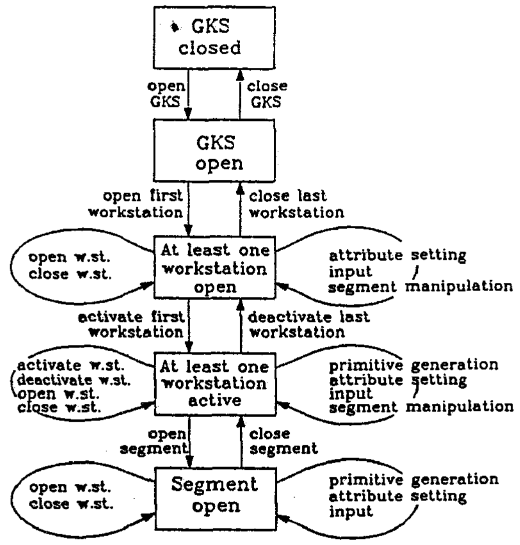

Figure 7-1

GKS precisely defines which states each GKS function can be called in and which GKS functions cause the state to change. Most of these are quite obvious as can be seen by the above examples.

7.3. GKS State Lists

Internal to GKS and hidden from the operator are a number of state lists which contain relevant information about the current state of GKS and the implementation itself. Although the operator need not know of their existence, the application programmer can improve the performance of the application program by inquiring about the features and the facilities available at a particular installation or on a workstation and adapting the program accordingly. The main state lists are:

(1) Operating State: this is a single value indicating the current state of GKS.

(2) GKS Description Table: a table giving information about the particular implementation of GKS. For example, it contains information such as the GKS implementation level, information on the available workstations (ie. number, names and types) and the maximum number of normalization transformations allowed.

(3) GKS State List: the main list containing all information about the status of GKS at any given instance.

This includes information about the number of open and active workstations, the current settings of all attributes including those of the output primitives, the currently selected normalization transformation (and its associated transformation number), segments that are open, etc.

(4) Workstation Description Table: these are set up by the installation manager for the site and contain information about the characteristics of the workstations available. There is one table for each workstation type available.

(5) Workstation State List: these contain information about the workstations that are open. Each workstation that is open has its own state list and it is initialized to certain values (screen size, etc) from the Workstation Description Table. The Workstation State List contains information concerning the device specific side of GKS. It includes the window and viewport settings and various tables described in the previous chapter, which control how a particular primitive appears on this workstation.

These tables contain all the current relevant knowledge of GKS and are accessed by the application program through a set of inquiry functions.

7.4. Inquiry Functions(*)

Inquiry functions in GKS allow the application programmer access to the information in the various GKS State Lists. They are useful in finding out what the state of the system is when an error occurs and also when writing library routines where it may be necessary to save the current state of the system and restore it later. A typical inquiry function has the form:

```
inquire_name_of_segment(id, segname) (*)
```

The inquiry functions in GKS have been arranged so that it is not possible to get a GKS error when calling them. This allows the inquiry functions to be called when in an error condition without resulting in further errors being generated which would obscure the original error. Also, it allows inquiry functions to be called with no likelihood of them having any impact on the state of GKS.

This is achieved by the inquiry functions all having their initial output parameter as an indication as to whether the information returned by the inquiry is valid or not.

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

If id has the value 0 on return, the segname parameter does contain the name of the open segment, but if id returns any other value, it indicates what error situation has occurred. In the above example, id could return a non-zero value which would indicate that there is no open segment and, therefore, it is not possible to give a name of an open segment.

These inquiry functions may be looked up when they are required in the GKS manual as there are quite a few of them (about 75).

7.5. Error Handling(*)

GKS has a well defined set of errors which are reported back to the application program. The philosophy adopted by GKS is that all errors are reported by putting details of the error in the error file which has been specified when GKS was opened. A typical error will record the following information:

(1) Error Number: a number indicating which error has occurred.

(2) GKS Function: the name of the GKS function that was being executed when the error was detected.

There is a full set of error numbers so that a particular error can be precisely identified. What happens when an error has been recognized depends on the application program. If no special action is taken, GKS calls the installation supplied error handling procedure:

```
error_handler(no, fid) (*)
```

The parameters supplied to it are the error number (no) and function name in which the error was detected (fid). The action of this procedure is to record the error information in the error file by calling the procedure:

```
rep_er(no, fid)
```

and then return to the GKS function where the error was detected. Normally a GKS function causing an error has no effect on GKS in the sense that on returning from the GKS function, the state of GKS is as if the function had not been called. In some cases, GKS may not be able to take the clean up action to achieve this and then the effects are unpredictable.

This rather cumbersome approach of organizing the error handling has been set up to allow the application program to

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

specify its own error handling procedure which takes a specific action. For example:

```
        open_gks();
        open_ws(ser, 6, 3);
        activate((ser);
        closeg();
```

This program would cause an error as the GKS State would be "WORKSTATION ACTIVE" when "closeg" was called rather than the expected "SEGMENT OPEN". As a result, the installation supplied error handling procedure would be called. If the application program required to know which workstations were active when the error occurred, it would substitute its own error handling procedure as follows:

```
        error_handler(no, fid)
        Int      no;
        Fcts     fid;
        {
                 if (nmber = 4)
                 {
                         inquire_set_of_wksts(12, id, nws, ws);
                                  .
                                  .
                 }
                 rep_er(no, fid);
                 return;
        }
```

The main point to note is that the user supplied error handling routine should still call "rep_er" before returning. In this example, error number 4 indicates a segment was not open. The inquiry function returns the number of active workstations in nws and the workstation identifications in ws(1) to ws(nws). The user supplied error handler may write more information into the error file. The only GKS functions which can be invoked in the error handling procedure are:

```
        rep_er(no, fid)
        inquiry_functions (*)
        emergency()
```

There are occasions when it will not be possible to recover from an error and then the requirement is to save as much of the graphical information produced as possible, for example ensuring that any output information buffered within GKS is transmitted to the device. GKS provides the function:

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

emergency()

for this purpose. GKS itself will invoke this function in response to some classes of error. This function may be invoked by a user specified error handling procedure also.

7.6. Levels of Implementation

It is clear from the previous sections that GKS is a comprehensive graphics system containing most of the features required by the applications programmer. This in itself will mean that GKS is quite large and, for simple applications, it is feasible that a great deal of unnecessary complexity is available in GKS. In this case, the user would like to use only a subset of the facilities available in GKS. This is achieved through the level structure.

GKS has nine levels defined by three different choices on each of two axes. The two axes are approximately defined by input and by all other functions. The definition of the input choices are:

(a) No input functions allowed.
(b) Only REQUEST input allowed.
(c) REQUEST, SAMPLE and EVENT input are all allowed.

The definitions of the choices on the other axis are:

(0) Minimal output - all the output primitives are available but the meaning of primitive index values cannot be changed. The installation default settings must be used. There is at least one settable normalization transformation. Only one output workstation is allowed at a time.
(1) The main additions over (0) are the ability to specify the primitive index representations, the ability to have more than one output workstation active at a time, the ability to use segments and multiple normalization transformations.
(2) This allows the workstation independent segment storage facilities to be used.

The simplest GKS implementation is, therefore, Level 0a which allows output to a single workstation at any given time.

The current status of the GKS implementation, at the time of writting this report, roughly corresponds to that of 1b.

8. References

Enderle, G; Kansy, K. & Pfaff, G. Computer Graphics Programming, GKS - The Graphics Standard, Springer-Verlag, Berlin, 1984.

Hopgood, F.R.A., Duce, D.A., Gallop, J.R. & Sutcliffe, D.C. "Introduction to the Graphical Kernel System (GKS)", Academic Press, London, 1983.

"ISO/DIS 7942 Information Processing - Graphical Kernel System - Functional Description: GKS Version 7.2", ISO/TC97/SC5/WG2 N163 (1983).

9. Appendix A - Glossary

This section gives the definitions of the most important terms of the Graphical Kernel System (GKS). As far as possible, commonly accepted graphical terminology is used.

acknowledgement:
: Output to the operator of a logical device indicating that a trigger has fired.

Aspect of primitives:
: Ways in which the appearance of a primitive can vary. Some aspects are controlled directly by primitive attributes and some indirectly through a bundle table. Primitives inside segments have an aspect controlled through segments containing them, eg. highlighting; primitives outside segments do not.

attribute:
: A particular property that applies to a display element (output primitive) or a segment. Examples: highlighting, character height. _Note_: In GKS, some properties of workstations are called workstation attributes.

bundle index:
: An index into a bundle table for a particular output primitive. It defines the workstation dependent aspects of the primitive.

bundle table:
: A workstation dependent table associated with a particular output primitive. Entries in the table specify all the workstation dependent aspects of a primitive. In GKS, bundle tables exist for the following output primitives: polyline, polymarker, text and fill area.

cell array:
: A GKS output primitive consisting of a rectangular grid of equal size rectangular cells, each having a single colour. _Note_: These cells do not necessarily map on-to-one with pixels.

choice device:
: A GKS logical input device returning a non-negative integer to the application program defining which of a number of possibilities has been chosen. _Note_ : a returned value of 0 (zero) refers to a "no choice" situation.

clipping:
: Removing parts of display elements that lie outside a given boundary, usually a window or viewport.

colour table:
: A workstation dependent table, in which the entries specify the values of the red, green and blue intensities defining a particular colour.

coordinate graphics;
line graphics:
: Computer graphics in which display images are generated from display commands and coordinate data.

device coordinates (DC):
The coordinate system local to a particular device. Note: DCs expressed that lie outside the limits specified on a device are not displayed.

device driver:
The device dependent part of a GKS implementation intended to support a graphics device. The device driver generates device dependent output and handles device dependent interaction.

device space:
The space defined by the addressable points of a display device. Coordinates expressed within the device space but that do not lie within the display space are not displayed.

display device;
graphics device:
A device (eg. refresh display, storage tube display, plotter) on which display images can be represented.

display element:
A basic graphics element that can be used to construct a display image.

display image;
picture:
A collection of display elements or segments that are represented together at any one time on a display device.

display space:
That portion of the device space corresponding to the are available for displaying images. In GKS, display space is also used to refer to the working space of an input device as a digitizer.

display surface;
view surface:
In a display device, that medium on which display images may appear.

echo:
The immediate notification of the current value provided by an input device to the operator at the display console.

escape:
A function in GKS used to access implementation or device dependent features, other than for the generation of graphical output, which are not otherwise addressed by GKS.

feedback:
Output indicating to the operator the application program´s interpretation of a logical input value.

fill area:
A GKS output primitive consisting of a polygon (closed boundary) which may be hollow or may be filled with a uniform colour, a pattern, or a hatch style.

fill area bundle table:
A table associating specific values for all workstation dependent aspects of a fill area primitive with a fill

area bundle index. In GKS, this table contains entries consisting of interior style, style index and colour index.

Generalized Drawing Primitive (GDP):
An output primitive used to address special geometrical workstation capabilities such as curve drawing.

GKS level:
Two values in the range 0 to 2 and a to c which together define the set of functional capabilities provided by a specific GKS implementation.

GKS Metafile (GKSM):
A sequential file that can be written or read by GKS; used for long term storage (and for transmittal and transferral) of graphics information.

highlighting:
A device independent way of emphasising a segment by modifying its visual attributes. (A generalization of blinking.)

implementation mandatory:
Implementation mandatory describes a property that must be realized identically on all workstations of all implementations of the standard.

input class:
A set of input devices that are logically equivalent with respect to their function. In GKS, the classes are: LOCATOR, STROKE, VALUATOR, CHOICE, PICK and STRING.

locator device:
A GKS logical input device providing a position in world coordinates and a normalization transformation number referring to the currently selected window-to-viewport mapping.

logical input device:
A logical input device is an abstraction of one or more physical devices which delivers logical input values to the program. Logical input devices in GKS can be of type LOCATOR, STROKE, VALUATOR, CHOICE, PICK and STRING.

logical input value:
A value delivered by a logical input device.

marker:
A distinguishable mark which is used to identify a particular location.

measure:
A value (associated with a logical input device), which is determined by one or more physical input devices and a mapping from the values delivered by the physical devices. The logical input value delivered by a logical input device is the current value of the measure.

normalization transformation;
viewing transformation;
window-to-viewport transformation:
A transformation that maps the boundary and interior of a window to the boundary and interior of a viewport.

Note: In GKS, this transformation maps position in world coordinates to normalized device coordinates.

normalization transformation number:
A number associated with each normalization transformation which helps in identifying that particular instance of window-to-viewport transformation.

normalized device coordinates (NDC):
A coordinate specified in a device independent intermediate coordinate system, normalized to some range, typically 0 to 1. Note: In GKS, during an intermediate state the coordinates may lie outside the defined range, but associated clipping information ensures that the output does not exceed the coordinate range [0,1] x [0,1].

operator:
Person manipulating physical input devices so as to change the measures of logical input devices and cause their trigger to fire.

output primitive;
graphical primitive:
A display element. Output primitives in GKS are POLYLINE, POLYMARKER, TEXT, FILL AREA, CELL ARRAY and GENERALIZED DRAWING PRIMITIVE.

pick device:
A GKS logical input device used to identify a displayed object. It returns the segment name of the picked object along with a pick identifier.

pick identifier:
A name attached to individual output primitives within a segment. Used to identify individual components of the picked segment.

pixel;
picture element:
The smallest element of a display surface that can be independently assigned a colour or intensity.

polyline:
A GKS output primitive consisting of a set of connected lines.

polyline bundle table:
A table associating specific values for all workstation dependent aspects of a polyline primitive with a polyline bundle index. In GKS, this table contains entries consisting of linetype, linewidth scale factor and colour index.

polymaker:
A GKS output primitive consisting of a set of locations, each to be indicated by a marker.

polymarker bundle table:
A table associating specific values for all workstation dependent aspects of a polymarker primitive with a polymarker bundle index. In GKS, this table contains entries consisting of marker type, marker size scale factor and colour index.

primitive attribute:

Primitive attribute values (for output primitives) are selected by the application in a workstation independent manner, but can have workstation dependent effects.

prompt:
Output to the operator indicating that a specific logical input device is available.

raster graphics:
Computer graphics in which a display image is composed of an array of pixels arranged in rows and columns.

rotation:
Turning all or part of a display image about an axis. Note: In GKS, this capability is restricted to segments.

scaling;
zooming:
Enlarging or reducing all or part of a display image by multiplying the coordinates of display elements by a constant value. Note: For different scaling in two orthogonal directions two constant values are required. Note: In GKS, this capability is restricted to segments.

segment:
A collection of display elements that can be manipulated as a unit.

segment attributes:
Attributes that apply only to segments. Note: In GKS, segment attributes are visibility, highlighting, detectability, segment priority and segment transformation.

segment priority:
A segment attribute used to determine which of several overlapping segments take precedence for graphical output and input.

segment transformation:
A transformation which causes the display elements defined by a segment to appear with varying position (translation), size (scale) and/or orientation (rotation) on the display surface.

string device:
A GKS logical input device providing a character string as its result.

stroke device:
A GKS logical input device providing a sequence of points in world coordinates and a normalization transformation number referring to the currently active normalization transformation.

text:
A GKS output primitive consisting of a character string.

text bundle table:
A table associating specific values for all workstation dependent aspects of a text primitive with a text bundle index. In GKS, this table contains entries consisting of text fonts and precision, character expansion

factor, character spacing and colour index.

text font and precision:
An aspect of text in GKS, having two components font and precision, which together determine the shape of the characters being output, on a particular workstation. In addition, the precision describes the fidelity with which the other text aspects match those requested by an application program. In order to increase fidelity, the precisions are: STRING< CHAR and STROKE.

translation;
shift:
The application of a constant displacement to the position of all or part of a display image. Note: In GKS, this capability is restricted to segments.

trigger:
A physical input device or set of devices which an operator can use to indicate significant moments in time.

valuator device:
A GKS logical input device providing a real number.

viewport:
An application program specified part of normalized device coordinate space. Note: In GKS, this definition is restricted to a rectangular region of the world coordinate space used for the definition of the normalized transformation.

window:
A predefined part of a virtual space. Note: In GKS, this definition is restricted to a rectangular region of the world coordinate space used for the definition of the normalized transformation.

workstation:
GKS is based on the concept of abstract graphical workstations which provide the logical interface through which the application program controls physical devices.

workstation mandatory:
Workstation mandatory describes a property that must be realized identically on all workstations of a standard implementation.

workstation transformation:
A transformation that maps the boundary and interior of a workstation window into the boundary and interior of a workstation viewport (part of display space), preserving aspect ratio. Note: In GKS, this transformation maps positions in normalized device coordinates to device coordinates. The effect of preserving aspect ratio is that the interior of the workstation window may not map to the whole of the workstation viewport.

workstation viewport:
A portion of display space currently selected for output of graphics.

workstation window:

A rectangular region within the normalized device coordinate system which is represented on a display space.

World Coordinates (WC):

A device independent Cartesian coordinate system used by the application program for specifying graphical input and output.

10. Appendix B - GKS Error List

10.1. States :

1	GKS not in proper state; GKS shall be in the state GKCL
2	GKS not in proper state; GKS shall be in the state GKOP
3	GKS not in proper state; GKS shall be in the state WSAC
4	GKS not in proper state; GKS shall be in the state SGOP
5	GKS not in proper state; GKS shall be either in the state WSAC or in the state SGOP
6	GKS not in proper state; GKS shall be either in the state WSOP or in the state WSAC
7	GKS not in proper state; GKS shall be in one of the states WSOP, WSAC, or SGOP
8	GKS not in proper state; GKS shall be in one of the states GKOP, WSOP, WSAC or SGOP

10.2. Workstations

20	Specified workstation identifier is invalid
21	Specified connection identifier is invalid
22	Specified workstation type is invalid
23	Specified workstation type does not exist
24	Specified workstation is open
25	Specified workstation is not open
26	Specified workstation cannot be opened
27	Workstation Independent Segment Storage is not opened
28	Workstation Independent Segment Storage is already opened
29	Specified workstation is active
30	Specified workstation is not active

31	Specified workstation is of category MO
32	Specified workstation is not of category MO
33	Specified workstation is of category MI
34	Specified workstation is not of category MI
35	Specified workstation is of category INPUT
36	Specified workstation is Workstation Independent Segment Storage
37	Specified workstation is not of category OUTPUT
38	Specified workstation is neither of category INPUT nor OUTIN
39	Specified workstation is neither of category OUTPUT nor OUTIN
40	Specified workstation has no pixel store readback capability
41	Specified workstation type is not able to generate the specified generalized drawing primitive

10.3. Transformations :

50	Transformation number is invalid
51	Rectangle definition is invalid
52	Viewport is not within the Normalized Device Coordinate unit square
54	Workstation veiwport is not within display space

10.4. Output Attributes

60	Polyline index is invalid
61	A representation for the specified polyline index has not been defined on this workstation
62	Linetype is less than or equal to zero
63	Specified linetype is not supported on this workstation
64	Polymarker index is invalid

65	A representation for the specified polymarker index has not been defined on this workstation
66	Marker type is less than or equal to zero
67	Specified marker type is not supported on this workstation
68	Text index is invalid
69	A representation for the specified text index has not been defined on this workstation
70	Text font is less than or equal to zero
71	Requested text font is not supported for the specified precision on this workstation
72	Character expansion factor is less than or equal to zero
73	Character height is less than or equal to zero
74	Length of character up vector is zero
75	Fill area index is invalid
76	A representation for the specified fill area index has not been defined on this workstation
77	Specified fill area interior style is not supported on this workstation
78	Style (pattern or hatch) index is less than or equal to zero
79	Specified pattern index is invalid
80	Specified hatch style is not supported on this workstation
81	Pattern size is not positive
82	A representation for the specified pattern index has not been defined on this workstation
83	Interior style PATTERN is not supported on this workstation
84	Dimensions of the colour array are invalid
85	Colour index is less than or equal to zero
86	Colour index is invalid

87 A representation for the specified colour index has not been defined on this workstation

88 Colour is outside the range [0,1]

89 Pick identifier is invalid

10.5. Output Primitives

100 Number of points is invalid

101 Invalid code in string

102 Generalized drawing primitive identifier is invalid

103 Content of generalized drawing primitive data record is invalid

104 At least one active workstation is not able to generate the specified generalized drawing primitive

10.6. Segments

120 Specified segment name is invalid

121 Specified segment name is already in use

122 Specified segment does not exist

123 Specified segment does not exist on specified workstation

124 Specified Segment does not exist on Workstation Independent Segment Storage (WISS)

125 Specified segment is open

126 Specified priority is outside the range [0,1]

10.7. Input (*):

140 Specified input device is not present on workstation

141 Input device is not in EVENT mode

142 Input device is not in SAMPLE mode

143 EVENT and SAMPLE input mode are not available at

the level of GKS

144 Specified prompt and echo type is not available at this level of GKS

145 Echo area is outside display space

146 Contents of input data record is invalid

147 Input queue has overflowed

148 Input queue has not overflowed since GKS was opened or the last invocation of INQUIRE INPUT QUEUE OVERFLOW

149 Input queue has overflowed, but associated workstation has been closed

150 No input value is of the correct class is in the current event report

151 Timeout is invalid

152 Initial value is invalid

153 Length of initial string is greater than the implementation defined maximum

10.8. Metafiles (*):

160 Item type is not allowed for user items

161 Item length is invalid

162 No item is left in GKS metafile input

163 Metafile item is invalid

164 Item type is not a valid GKS item

165 Content of item data record is invalid for the specified item type

166 Maximum item data record length is invalid

167 User item cannot be interpreted

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

10.9. Escape (*):

180	Specified function is not supported
181	Contents of escape data record are invalid

10.10. Implementation Dependent :

300	Specified function is not supported in this level of GKS
301	Storage overflow has occurred in GKS
302	Storage overflow has occurred in segment storage
303	Input / Output error has occurred while reading
304	Input / Output error has occurred while writing
305	Input / Output error has occurred while sending data to a workstation
306	Input / Output error has occurred while receiving data from a workstation
307	Input / Output error has occurred during program library management
308	Input / Output error has occurred while reading workstation description table
309	Arithmetic error has occurred

10.11. Other Errors :

??? Other errors are numbered from 900 onwards.

(*)Because this implementation is a pre-released version of GKS, this facility is unavailable as yet.

11. Appendix C - A Sample Program

The sample program below shows the basic structure of an application program using GKS. If executed as it stands, this program will not produce any output (graphical or otherwise) because no application has been incorporated into it.

However, the sequence of actions performed by the program are as follows :

(1) GKS is set to the initial state of closed (GKCL);

(2) error file is opened;

(3) GKS is then opened thus changing the state to open (GKOP);

(4) workstation "ser" (*) is then opened and activated. The state now changes from GKOP to "at least one workstation open" (WKOP) and then to "at least one workstation active" (WKAC);

(5) the workstation transformation is achieved by mapping the unit square (0 -> 1) of NDC to the complete device display space;

(6) working area is then allocated;

(7) a normalization transformation (numbered 3) is defined to map the unit square (0 -> 1) in WC to the equivalent square in NDC;

(8) room has been left now for the creation of the application itself which may include calls to user defined functions and/or to GKS functions themselves. As nothing is included here, no output is therefore generated;

(9) following this, the workstation is deactivated and then closed. Hence, the GKS state now goes from WKAC, through WKOP to GKOP;

(10) finally, GKS is closed itself, thus returning to the initial state of GKCL.

The sample program:

(*)Workstations available to this point are: "hpp" for the Hewlett-Parkard 7475A plotter; "ser" for the Servogor 281 plotter and "tek" for the Tektronix 4006 terminal.

```
#include "cgksincl.h"   /* list of other #includes       */
#include "cgks.h"       /* all typedefs                  */
#include "cgksconst.h"  /* all constant definitions      */

Gks     *open_gks();
Wss     *open_ws();

extern  Erarea  er_d_a;
Ercode  er_handle();    /* error handler                 */
Size    gks_size;

extern  Gks     *gkss;
extern  GksState gksst;
extern  header;         /* contains title of the output */

Wss     *wss_ser;       /* workstation to be used        */

extern FILE *errfp;     /* error file                    */

* GLOBAL VARIABLES */
/* ====== ========= */

Drect   drect0 = {100, 100, 2700, 2700};
                        /* device coordinates (see footnote) */
Nrect   nrect0 = {0.0, 0.0, 1.0, 1.0};
                        /* normalized device coordinates     */
Wrect   wrect0 = {0.0, 0.0, 1.0, 1.0};
                        /* workstation's window coordinates  */

Wc      *st;
Int     k, n;

/* USER DEFINED FUNCTIONS */
/* ==== ======= ========= */

/* MAIN */
/* ==== */


main()
{
        /*      interaction prior to opening gks     */
        /*      ----------- ----- -- ------- ---     */
```

(*)The range of workstation device coordinates are: "hpp" = x: 0 - 16000, y: 0 - 11000; "ser" = x: 0 - 3300, y: 0 - 2790 and "tek" = x: 0 - 1020, y: 0 - 760.

```
/*      set up error file        */
/*      --- -- ----- ----        */
gksst = GKCL;
errfp = fopen("errorf", "w");
setbuf(errfp, NULL);

/*      open GKS & initialize workstation    */
/*      ---- --- - ---------- -----------    */
gkss   = open_gks(er_handle,&er_d_a, gks_size);
wss_ser = open_ws(NULL, NULL, "ser");
activate(wss_ser);

/*      set workstation window & viewport    */
/*      --- ----------- ------ - --------    */
s_w_wind(wss_ser, &nrect0);
s_w_view(wss_ser, &drect0);
update(wss_ser);

/*      allocate memory work area       */
/*      -------- ------ ---- ----       */
n = 2000;
if ((st = (Wc *) malloc(n * sizeof(Wc))) == NULL)
{
        rep_er(302, MAL);
        return;
}

/*      set window & viewport    */
/*      --- ------ - --------    */
k = 3;
s_window(gkss->gk_ntran[k], &wrect0);
s_viewport(gkss->gk_ntran[k], &nrect0);
sel_cntran(gkss->gk_ntran[k]);

/*      create an application    */
/*      ------ -- -----------    */


/*      close up workstation & GKS      */
/*      ----- -- ----------- - ---      */
deactivate(wss_ser);
close_ws(wss_ser);
close_gks();
```