GKS PROGRAMS - ARE THEY PORTABLE?

K.W. Brodlie

Department of Computer Studies

University of Leeds

Leeds, UK

1. INTRODUCTION

GKS (ISO, 1985) was established in 1985 as the first international standard for computer graphics. It has gained widespread acceptance by the graphics community: both DEC and IBM provide a GKS implementation on their principal machine ranges; graphics terminal manufacturers now provide GKS firmware; a number of software houses distribute GKS implementations; and (slowly) graphical applications software is being developed on top of GKS.

Yet GKS is not without its detractors. For example, there is a challenging article by Sanders (1987) from the leading US graphics software house Precision Visuals, that seeks to explode some myths about graphics standards: one myth is that 'writing applications on standards-based software will provide the best insurance that an application runs on all CPUs and devices'. Sanders claims that in reality GKS programs are not portable.

Standards are reviewed every five years, and so thought is now being given to a revision of GKS. This too has resulted in much criticism, with some of it again being directed at the lack of portability. For example, Pfaff (1987) argues that there is a lack of uniformity among different, standard-conforming implementations of GKS - there are too many 'implementation- and workstation-dependencies'.

In the meteorology community GKS has been used for some time at the UK Met Office. But again acceptance has been mixed with criticism. Little (1987) summarises the situation as follows: 'GKS is an appropriate standard for the UK Met Office. However, too many implementation dependent variations exist in implementations to enable portable software to be produced very easily.'

These comments are serious. The whole point of having a graphics standard is to provide a unique base on top of which portable graphical applications software can be written. Has the standard failed?

This paper takes an optimistic view. It begins by describing those features of the GKS standard which were included with portability in mind, and then with hindsight, it judges whether they do provide the guarantees that were hoped for. Two case studies are examined: both are examples of large GKS application programs with which the author has been involved and for which portability is a crucial factor. The GKS validation suite was developed as a means of testing GKS implementations for conformance to the standard; by its very nature, it is an excellent test of whether GKS programs are portable: it has to run with any (valid) GKS implementation on any host system and with any graphical device. The NAG Graphics Library is a set of high-level graphical routines that can be used, through an interface, with a variety of underlying graphics systems. The GKS version, like the validation software, has to run with different GKS implementations, on different hosts and different devices.

The conclusion, in both cases, is that GKS programs can be ported between different GKS implementations. However, it is firmly believed that the *key* to portability lies in the use of a good style of programming. Thus the paper finishes by describing a model for GKS programming - a model that gives a proper basis for writing portable GKS programs.

2. GKS PORTABILITY - THEORY AND PRACTICE

The second sentence of the GKS standard reads as follows:

The main reasons for standardizing basic computer graphics are:

> a) to allow application programs involving graphics to be easily portable between different installations;
> b) to aid the understanding and use of graphics methods by application programmers;

So GKS nails its colours firmly to the mast right at the start: portability is the aim. Reason (a) can be thought of as 'portability of programs'; reason (b) as 'portability of programmers'. Programmer portability has certainly been achieved. GKS has given the graphics community a common terminology for computer graphics: two experts can speak to each other without fear of misunderstanding through confusion of terminology. This benefit is easy to underestimate.

The designers of GKS went to some trouble to encourage program portability too. For example, GKS is divided into a set of levels, with three levels of increasing output capability (0,1,2) and three levels of increasing input capability (a,b,c) -giving a matrix of nine levels. The conformance rules of GKS state that an implementation shall support *exactly* the functions of one level. This was in spite of strong calls for a more relaxed view, namely that implementations shall support *at least* the functions of one level. Had this latter view been upheld, then implementations would have competed with each other to add extra features, programmers would have used these features, and the resulting programs would not be portable.

Still, the number of levels (nine in all) is quite large and has given rise to some criticism. Sanders (1987) argues that portability is ensured only if one uses the lowest level of GKS (level 0a). Certainly this is true in theory, but at worst a piece of applications software can be tagged with the lowest of the nine levels that will support the application. In practice the situation is far better: the great majority of implementations have homed in on level 2b and today software can happily be written under the assumption that level 2b support will be available. The only facility omitted from this level is asynchronous input (SAMPLE and EVENT modes) which requires concurrent processing. A few full (ie level 2c) implementations do exist, but these tend to be host-specific and are not widely available. Thus any application using asynchronous input should be regarded as special.

Natural forces have therefore concentrated GKS implementations at one level - level 2b - and it cannot be claimed that the level structure of GKS is a barrier to portability.

In defining GKS there was pressure to allow scope to 'go outside the standard'. For example, there is no circle drawing primitive in GKS, yet some graphics devices include firmware for circle generation. A programmer may reasonably wish to access such a facility for efficiency reasons... but how can this be achieved without jeopardising portability? GKS solves the dilemma in the form of the GENERALISED DRAWING PRIMITIVE or GDP. The GDP function has an identifier as one of its parameters, this identifier selecting from a set of implementation-dependent 'special' drawing facilities. Implementors can provide as many GDPs as they wish - or none at all. The ESCAPE function provides a similar mechanism for non-geometric facilities. Programmers can use these functions if provided, but they do so in the knowledge that the facilities may not be supported in another implementation. Likewise anyone receiving a GKS program knows that they must check whether GDPs or ESCAPEs have been used. Thus the scope

for going outside the standard is carefully controlled.

However, despite the controls, it has to be admitted that GDPs and ESCAPEs are a danger to portability. It was intended that a registration process would be set up, whereby commonly used GDP and ESCAPE functions would be assigned a unique identifier - so that, for example, a circle GDP if implemented would always have identifier 1, say. The National Bureau of Standards in the United States is ready to act as the registration authority, but in practice it has taken an exceedingly long time to get the registration procedures approved by ISO and the scheme is not yet in operation. Thus the advice at present has to be: for portable programs, avoid GDPs and ESCAPEs.

It is often said that GKS is a standard for 'device-independent' graphics. That is one myth that certainly ought to be exploded. Devices themselves are different: this is not something to be disguised, but rather something to be exploited. What is needed is not a device-independent graphics standard, but a standard that allows one to properly adapt programs to different environments. GKS does exactly that. A large number of inquiry functions are provided (nearly 50% of the total number of functions in fact), and these return a description of the graphics environment: properties of the implementation and the workstations it supports. Programs can be portable, yet produce output that is tailored to the device being used.

There is a strong suspicion that GKS inquiry functions are underused. It has been noticeable in running the GKS validation software, which makes considerable use of inquiry functions to configure itself, that errors are rarely found in the drawing parts of GKS, but quite regularly inquiry functions will fail. For example, an early version of DEC's VAX GKS (otherwise a reasonable implementation) contained errors in the language binding of a number of inquiry functions. Similarly Tektronix GKS (now corrected) omitted two inquiry functions completely. This suggests that inquiry functions are rarely being used - else surely the errors would have been discovered much sooner.

3. GKS VALIDATION SOFTWARE

It was recognised at an early stage in the development of GKS that the portability aim would only be achieved if all implementations adhered strictly to the standard. Thus the validation of GKS implementations is a crucial issue.

Work began in 1981 on the development of a large suite of test programs for GKS. These are written in Fortran and aim to check as many features of GKS as possible. The work was done mainly at the Universities of Leicester and Darmstadt, with financial support from the CEC. The software now forms the basis of a European-wide GKS validation service, run jointly by GMD in West Germany and NCC in the UK.

The GKS validation software must be able to run with different GKS implementations - that is its very purpose. It must also be easily transported between different host systems, and it should be able to produce output on different graphics devices. Thus it is an excellent measure of the portability of GKS applications software.

The validation suite has already been run successfully against a number of GKS implementations. Much use is made of GKS inquiry functions to determine the facilities available in an implementation, and hence to check them. Thus the software adapts itself to the surroundings it finds itself in.

Some manual configuration is required, chiefly to tailor the software to the operating system. It may be of interest to describe this, since it could act as a useful model for other large GKS applications. First of all, the validator is asked to supply the parameters of the OPEN GKS function (error file and buffer size). Next a table is constructed, in which the validator enters the different graphical devices or workstations to be included in the tests: for each workstation, the validator supplies the name by which he wishes to refer to the workstation, the connection identifier and workstation type. Workstation identifiers are allocated to each workstation giving a table of the form:

Device	Work'n Identifier	Connection Id	Work'n Type
Tektronix 4010	1	0	201
Sigmex 6130	2	0	121
CalComp 1012	3	0	710

When a test program is run, this table is used to present the tester with a menu of available devices; when one is selected, the workstation identifier, connection identifier and workstation type are taken from

the table.

4. NAG GRAPHICAL LIBRARY

The NAG Graphical Library (NAG, 1985) is a collection of graphical routines intended as a companion to the NAG Library of numerical and statistical routines. The software is written in Fortran, and includes routines for curve and function drawing, contouring, surface views and data presentation in the form of histograms, bar charts and pie charts. The Graphics Library is not a self-contained graphics system, but sits on top of different underlying graphics packages. The NAG routines are written in terms of a small set of interface routines (to draw a line, draw a character, etc), and these interface routines are implemented in terms of the graphics package available at each particular site. The Library is widely used in the UK and indeed throughout the world, with some 300 sites.

A fair number of these sites now use the software in conjunction with GKS. Only level 0a of GKS is required. The Graphics Library runs successfully with a variety of GKS implementations - indeed any difficulties have been traced to errors in the GKS implementations rather than any fundamental problems in writing portable GKS applications software. Again much use is made of inquiry functions.

5. A MODEL FOR GKS PROGRAMS

The two previous sections have shown that GKS programs can be ported between different implementations. However, the ease of portability can be increased significantly by writing GKS programs in a clear, structured manner. Some recommendations on how this can be done are given in this section.

GKS itself says little about the style in which programs should be written. This is a pity, because it is very easy to write bad GKS programs! Consider this very simple example. A couple of graphs are to be drawn, side by side on the display surface. The left-hand graph is to be drawn in the default colour, the right-hand graph is to be drawn in a different colour.

The following sequence of GKS functions will achieve this:

```
OPEN GKS (error_file, buffer)
OPEN WORKSTATION (1, con_id, wktype)
ACTIVATE WORKSTATION (1)
SELECT NORMALISATION TRANSFORMATION (1)
SET VIEWPORT (1, 0.0, 0.5, 0.0, 1.0)
POLYLINE (npts_1, pts_1)
SET VIEWPORT (1, 0.5, 1.0, 0.0, 1.0)
SET POLYLINE REPRESENTATION (1, 1, 1, 1.0, 2)
POLYLINE (npts_2, pts_2)
DEACTIVATE WORKSTATION (1)
CLOSE WORKSTATION (1)
CLOSE GKS
```

The above program has been written with no thought to structure, and no thought to portability. It is short and therefore it is possible to understand it by working through it line-by-line. Any larger program written to this style would be very hard to follow.

Consider some of the problems. Suppose a change to a different workstation is required, one without colour. The graphs could now be distinguished by linetype, but the programmer has to search for the SET POLYLINE REPRESENTATION function in the middle of the program, which will have to be altered. Imagine the difficulty if an application contained several thousand lines of code. Again suppose the layout is to be rearranged so that the graphs lie one on top of each other rather than side-by-side. The programmer has to search through the program for the SET VIEWPORT functions. More seriously, there are errors lurking even in this simple program. The SET POLYLINE REPRESENTATION function redefines the representation of index 1 - but index 1 (the default) has already been used to draw the first graph. This can have various effects in GKS depending on the properties of the workstation: it could well be that the left-hand graph will change to have the new representation as well - certainly not what is intended. The scattering of representation functions throughout a program makes it very hard to spot possible errors like this. Finally, there are two fundamental modes of attribute setting in GKS - bundled and individual. Bundled mode tends to be the default in Europe, individual in the US. This program assumes bundled mode as default, but any portable program must explicitly select the mode required by calling SET ASPECT SOURCE FLAGS.

Consider now how the style of the above program could be improved. It is helpful to split the construction of a picture into three separate stages. First, the *picture definition* stage, where the geometry of a picture is defined in terms of output primitives. A picture definition may be divided into several parts if

different world coordinate systems are being used, one part for each coordinate system. The second stage is *picture composition*, where the layout on the display surface is specified using different window-viewport transformations. The final phase is *picture representation*, where the 'abstract' picture defined in the first two stages is mapped to a particular workstation, or indeed workstations, using representation functions.

The picture definition is placed at the heart of a GKS program: it should never need to change when moving a GKS program from one environment to another. Indeed one can imagine libraries of picture definition modules being developed. The picture composition stage forms an outer layer: this may occasionally need to be changed, if a new layout of the picture elements is required. The picture representation layer will need to change when a program is ported, and so that quite correctly forms the outer layer. This change can be done manually, or inquiry functions can be used to do automatic tailoring to different environments.

This gives the following model for GKS programs:

```
OPEN GKS
SET ASPECT SOURCE FLAGS (bundled)

    OPEN WORKSTATION
    Define Picture Representation

        Define Layout

            Define Drawing

    CLOSE WORKSTATION

CLOSE GKS
```

The earlier example can now be rewritten, following the model that has just been developed:

```
OPEN GKS (error_file, buffer)
SET ASPECT SOURCE FLAGS (bundled)

    OPEN WORKSTATION (1, con_id, wktype)
    SET POLYLINE REPRESENTATION (1, 1, 1, 1.0, 1)
    SET POLYLINE REPRESENTATION (1, 2, 1, 1.0, 2)

        SET VIEWPORT (1, 0.0, 0.5, 0.0, 1.0)
        SET VIEWPORT (2, 0.5, 1.0, 0.0, 1.0)

            ACTIVATE WORKSTATION (1)
            SELECT NORMALISATION TRANSFORMATION (1)
            POLYLINE (npts_1, pts_1)
            SELECT NORMALISATION TRANSFORMATION (2)
            SET POLYLINE INDEX (2)
            POLYLINE (npts_2, pts_2)
            DEACTIVATE WORKSTATION (1)

    CLOSE WORKSTATION (1)

CLOSE GKS
```

Notice that the program now has a clean structure, it is easy to change the layout (multiple normalisation transformations have been used) and it is easy to change the representation of the graphs for different workstations.

6. CONCLUSIONS

Experience with two major pieces of GKS application software has shown that GKS programs are portable. Inquiry functions in GKS provide a powerful tool to enable application programs to adapt to the environment they find themselves in - yet there are indications these functions are rarely used.

The importance of a good model for GKS programming has been stressed, and a model that will provide a good basis for portable GKS programs has been described.

GKS was designed to support the development of portable graphical software in application areas such as meteorology. It is now well established as a standard, and it can be strongly recommended as a support graphics system for European meteorological applications.

REFERENCES

ISO, 1985: Information processing systems - Computer graphics - Graphical Kernel System (GKS) functional description. ISO 7942-1985(E).

Little, C., 1987: GKS at the UK Met Office. Proceedings of the GKS Review, Eurographics.

NAG, 1985: The NAG Graphical Supplement Manual - Mark 2. NAG Ltd, Oxford.

Pfaff, G.E., 1987: GKS Review position paper. Proceedings of the GKS Review, Eurographics.

Sanders, R., 1987: The myth of graphics standards. Systems International, October 1987, 92-94.