stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 217/82 DECEMBER

D.S.H. ROSENTHAL

THE GKS INPUT FACILITIES AND HOW TO USE THEM

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

CR Categories and Subject Descriptors: I.3.4 [Computer Graphics]: Graphics Utilities — graphics packages; I.3.6 [Computer Graphics]: Methodology and Techniques — device independence; interaction techniques

The GKS Input Facilities and How To Use Them

David S. H. Rosenthal

ABSTRACT

The input facilities of GKS, the draft international standard for 2-D graphics software, are presented from an application programmer's viewpoint. The basic concepts are reviewed, concentrating on the differences between GKS and earlier systems.

These input facilities can be used in three distinct styles. One provides high portability by sacrificing control over details of the user interface. Another can exploit hardware capabilities by sacrificing portability. A third can provide portability *and* control over the user interface, at the cost of extra application code. All three styles are described, and illustrated with skeleton applications.

General Terms: Standardization

This paper will be submitted for publication.

1. Introduction

"Its so simple, so very simple,
that only a child can do it."
Tom Lehrer, *New Math.*

GKS[2], the draft international standard for 2-D graphics software, provides a set of input facilities based on the *virtual input device* concept[9], and broadly compatible with earlier device-independent graphics systems[7]. The differences that do exist are due in part to the special features of the GKS output pipeline, and in part to the refinements of the virtual input device model introduced during the international review of GKS.

The concepts underlying the GKS input facilities will be reviewed. These input facilities can be used in three distinct styles:

— A style providing high portability but sacrificing control over details of the user interface.

— A style capable of exploiting special hardware capabilities but which sacrifices portability.

— A style made possible by the combination of GKS' LOCATOR device with its multiple window-viewport capabilities. This provides both portability and control over the user interface, at the cost of extra application code.

All three styles are described, and illustrated with skeleton applications.

This paper is intended only as an introduction to GKS' input facilities. For general information on the system, the reader is referred to [1], and for a detailed discussion of the input facilities to [4].

2. Simple Usage

An application using GKS obtains input from one or more *logical input devices*. GKS *simulates* these logical devices using whatever physical input devices are available.

When an application using GKS needs to obtain input from the operator, the programmer is faced with two questions:

— What kind of input value is required?

— How should the operator supply it?

The first question determines the appropriate device *class*, and the second the appropriate device *attributes*. These concepts will now be reviewed.

2.1. Device Classes

GKS divides its logical input devices into classes according to the type of the input value that the device returns. The GKS classes, and the corresponding data types, are set out in Table 1.

Table 1 — GKS Device Classes

Class	Returns
LOCATOR	Single Position in World Coordinates Normalisation Transformation ID
STROKE	Sequence of World Coordinate Positions Normalisation Transformation ID
VALUATOR	Single Real Number
CHOICE	Non-negative Integer
PICK	Single Segment Name Pick Identifier
STRING	Sequence of Characters

These data types are the basic units from which GKS interaction sequences are built.

The simplest way for the application to access these devices is via the REQUEST functions, one for each class. The REQUEST VALUATOR function would be declared in PASCAL as:*

```
procedure RequestValuator (
    wsid: WorkstationID;
    dev : DeviceNumber;
var stat : RequestStatus;
var val  : real );
```

Invoking *RequestValuator* causes GKS to activate the *dev*-th VALUATOR device on workstation *wsid*, wait for the operator to input a value using it, and return OK in *stat* and the value in *val*. The operator may refuse to supply a value, by using the "break" facility.† In this case, *stat* will be NONE and *val*

* In the examples, GKS function and procedure names are in **this font**. Other functions and procedures are assumed to be defined elsewhere in the application, possibly using other GKS functions.

† GKS insists that this facility is available, but does

will be unchanged.

The REQUEST functions for the other classes differ only in the type of *val*; for example, the REQUEST LOCATOR function would be declared as:

```
procedure RequestLocator (
    wsid: WorkstationID;
    dev : DeviceNumber;
var stat : RequestStatus;
var val  : Location );
```

The programmer can choose freely among the logical device classes, without sacrificing portability, because the standard insists that at least one device in each class must be available. The implementor of a GKS system must provide suitable simulations for each of these classes on the hardware available at each site. For example, one site might simulate a VALUATOR using a control knob, another using a mouse to point at a bar, and a third by means of digits typed on the keyboard.

2.2. Using Device Classes

To illustrate typical uses for each of the device classes, consider a simple draughting application. It allows the operator to draw lines of various types, define symbols, store them in a library, select and place them in the drawing, and annotate the drawing with text in various fonts and sizes.

The application operates in several *modes*, say drawing, annotating, setting parameters, storing symbols, and recalling symbols. The top level will use a CHOICE device to select the operating mode:

```
selectmode := 1; { device to use }
repeat begin
    RequestChoice(ws, selectmode, stat, newmode);
    if (stat = OK) then
        case (newmode) of
        DRAWING:
            DoDrawing;
        ANNOTATING:
            DoAnnotating;
        SETTING:
            DoSetting;
        STORING:
            DoStoring;
        RECALLING:
            DoRecalling;
        end;
end until (stat <> OK);
```

Note how the CHOICE device in effect returns an element from an enumerated type.

DoSetting, implementing the parameter setting mode will use a CHOICE device to select the parameter to affect, and perhaps a VALUATOR to acquire the new value. For example:

```
repeat begin
    RequestChoice(ws, selectparam, stat1, param);
    if (stat = OK) then
        case (param) of
            .....
        LINEWIDTH:
            RequestValuator(ws, setlinewidth,
                stat2, linewidth);
            .....
        end;
end until (stat <> OK);
```

Note how this implements a hierarchical menu system, with the "break" facility used to pop up a level.

DoDrawing, implementing the drawing mode, would use a STROKE device to obtain the connected chains of lines:

```
repeat begin
    RequestStroke(ws, getlines, stat, newline);
    if (stat = OK) then begin
        StoreLine(newline);
        DrawLine(newline);
        end;
end until (stat <> OK);
```

DoAnnotating, implementing the annotation mode, would use two devices, a STRING device to obtain the text, and then a LOCATOR to position it. The LOCATOR would be invoked repeatedly to allow the operator to

---

not specify how it is to be implemented. On the UNIX GKS[5], a break is performed by typing the end-of-file character.

"shuffle" the label into place, before confirming the position using the "break".

```
repeat begin
    RequestString(ws, getchars, stat1, label);
    if (stat1 = OK) then repeat begin
        RequestLocator(ws, getposn, stat2, where);
        if (stat2 = OK) then
            ShowLabel(label, where, TEMPORARY)
    end until (stat2 <> OK);
    ShowLabel(label, where, PERMANENT)
end until (stat1 <> OK);
```

DoRecalling, implementing the symbol placing mode, would again use two devices. The symbol to be placed would be selected from a display of the defined symbols using a PICK device, and then a LOCATOR device would be used to determine the position of a series of instances in the drawing:

```
repeat begin
    RequestPick(ws, getsymbol, stat1, symb);
    if (stat1 = OK) then repeat begin
        RequestLocator(ws, getposn, stat2, where);
        if (stat2 = OK) then
            PlaceSymbol(symb, where)
    end until (stat2 <> OK);
end until (stat1 <> OK);
```

3. Advanced Usage

GKS systems provide one of three levels of output capability:

— minimal output (0)

— full output, basic segmentation (1)

— full output, full segmentation (2)

and one of three levels of input capability:

— no input (a)

— synchronous input (b)

— asynchronous input (c)

The previous section has described the use of the level (b) facilities. They are called *synchronous* because, when a REQUEST function is invoked, execution of the application is suspended until the input is available. The input devices are active only when the application is inactive, and *vice versa*.

The previous section concentrated on the interface between the application program and the logical devices. GKS also provides some control over of the interface between the operator and the logical devices. The major factor controlling this interface is clearly the hardware a GKS implementation uses to simulate the logical devices, and this cannot be standardised. Nevertheless, GKS has abstracted certain attributes common to physical devices and also provided a route for individual, non-standard attributes to be accessed.

3.1. Device Attributes

The device attributes are accessed via INITIALISE functions, one for each class. For example, the INITIALISE LOCATOR function would be declared as:

```
procedure InitialiseLocator(
    wsid: WorkstationID;
    dev : DeviceNumber;
    init : Location;
    pet : PromptEchoType;
    area: DCRectangle;
    stuff: DeviceDataRecord );
```

Invoking *InitialiseLocator* sets the atributes of the *dev*-th LOCATOR on workstation *wsid* to the values given. The effects of the various attributes are as follows.

Init is an initial value for the device. If possible, the device should behave as if the operator had input this value before being given control of the device. An example of the use of *init* is in *DoRecalling*:

```
repeat begin
    RequestPick(ws, getsymbol, stat1, symb);
    if (stat1 = OK) then repeat begin
        InitialiseLocator(ws, getposn, where, .....);
        RequestLocator(ws, getposn, stat2, where);
        if (stat2 = OK) then
            PlaceSymbol(symb, where)
    end until (stat2 <> OK);
end until (stat1 <> OK);
```

Re-setting the initial value to the last value received ensures that, if the hardware can do it, the LOCATOR will re-appear in the same place it was left.

Pet and *area* work together to give control over the appearance of the logical device. For each class, a number of Prompt/Echo types are defined by the standard, and others may be provided by the implementor. The types for LOCATOR are shown in Table 2. Note that an implementation is only required to provide

Table 2 - LOCATOR Prompt/Echo Types	
Type	Effect
1	use a device-dependent technique
2	use a crosshair intersecting at the LOCATOR
3	use a tracking cross at the LOCATOR
4	use a rubber-band line from *init* to the LOCATOR
5	use a rubber rectangle with diagonal from *init* to the LOCATOR
6	show a digital display of the LOCATOR coordinates within the rectangle *area*

type 1, other types may not be available.

Stuff is a data record that an implementation may use to pass parameters to non-standard device functions. For example, a common technique on bitmap displays is to vary the shape of the icon displayed at the cursor to provide information to the operator. This could be supported by including in the data record a cell array defining the shape of the cursor. Extensions of this kind are useful, but not portable.

The INITIALISE functions for other classes differ only in the type of *init*, and in the fields defined for the data record. Other classes define other Prompt/Echo types, such as text menus for the CHOICE class, or digit displays for the VALUATOR class.

3.2. Asynchronous Input

The level (c) facilities provide for the application and its input devices to be active simultaneously. Each device may be in one of two *modes*.

In SAMPLE mode, the device runs continuously, but input values are only supplied when the application asks (or *polls*) for them, using one of the SAMPLE functions:

```
procedure SampleLocator(
    wsid: WorkstationID;
    dev : DeviceNumber;
var val : Location );
```

SAMPLE mode provides the ability to handle many active devices simultaneously, with the computer supplying the timebase. It would be appropriate, for example, to access the controls of a video game:

```
while GoOn do begin
    SampleLocator(ws, player1, where1);
    newposition(player1, where1)
    SampleChoice(ws, player1, action1);
    newaction(player1, action1);
    SampleLocator(ws, player2, where2);
    .....
    sleep(delay)
end;
```

In EVENT mode, the application reads input values from a queue. The device runs continuously, adding input values to the queue whenever it is triggered to do so. The application obtains the value from the head of the queue with the function:

```
AwaitEvent(
    time: real;
var wsid: WorkstationID;
var class     : DeviceClass;
var dev : DeviceNumber );
```

to discover what kind of device caused the event, and then the appropriate GET function, for example:

```
GetLocator(
var val : Location );
```

EVENT mode again provides the ability to handle many devices simultaneously, but with the operator supplying the timebase. It can be used to implement mode-free applications, using different devices to select different operations:

```
while GoOn do begin
    AwaitEvent(timeout, ws, class, dev);
    case (class) of
    TIMEOUT:
        DoHelp;
    LOCATOR:
        begin
            GetLocator(where);
            DoPosition(where)
        end;
    VALUATOR:
        begin
            GetValuator(much);
            DoValue(much)
        end;
        .....
    end
end;
```

The mode of a device is initially REQUEST, but may be changed by the SET MODE functions, one for each class. The SET LOCATOR MODE function would be declared as:

```
procedure SetLocatorMode(
    wsid: WorkstationID;
    dev : DeviceNumber;
    mode    : DeviceMode;
    echo: boolean );
```

4. Alternative Style

Many graphics terminals, particularly current personal workstations, provide a keyboard and a pointing device, such as a mouse, a tablet, or a trackball. They do not provide the large numbers of button boxes, control knobs, and joysticks that are common with more expensive terminals. A GKS application wanting tight control over the user interface can, therefore, assume that at least the LOCATOR class will be closely related to the hardware.

So far, an important point about GKS' LOCATOR and STROKE classes has been obscured. It is their relationship to the GKS output pipeline. Output is described in a world coordinate system (WC), and is transformed to a normalised device coordinate system (NDC) to determine its placement on the display surface. Many previous systems have provided only one such normalisation transformation, but GKS provides many (see Figure 1).

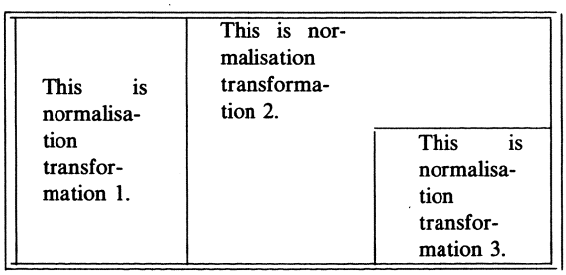

Figure 1 — GKS LOCATOR Input

The value returned by a LOCATOR or a STROKE device consists not merely of world coordinates, but also the identity of the normalisation transformation used to re-transform them from NDC.

```
type
    Location = record
        tran : TransformID;
        posn    : WCoord;
    end;
```

If the operator Locates in the left side of the screen in Figure 1, the *tran* field will contain 1, whereas from the lower right it will contain 3. This extra information allows for an alternative, window-based style of GKS application that centres its interaction on the pointing device. Example 6 in Annex F of the GKS document illustrates this style.

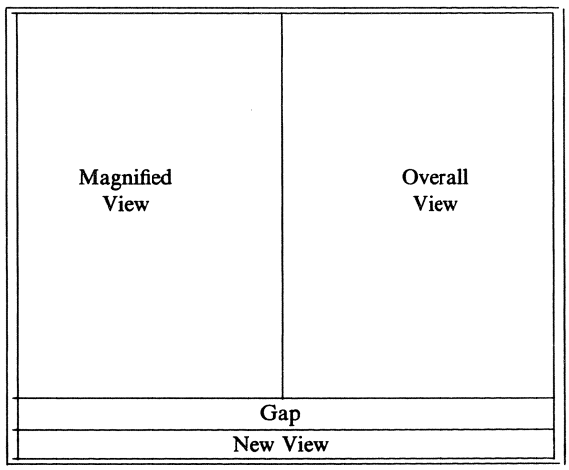

Figure 2 — GKS Example 6

A picture is being composed from positions input by a LOCATOR. Two *Locations* are used to define each line. The whole picture is displayed on the right side of the screen while

on the left side it is possible to display a sub-area at greater magnification (see Figure 2). Updates to the picture can be made in either drawing area with the left hand picture being used for detailed drawing. A *Location* defines which area is being used for inputting data as well as the data value itself.

To redefine the display in the left hand window, a LOCATOR input is required in a command area, containing NEW VIEW, at the bottom of the display area. The next two LOCATOR inputs define the lower left and upper right of the sub-area to be displayed on the left region. To cause a move to a new sequence of connected lines, a LOCATOR input is required in another command area, containing GAP, also at the bottom of the display area. The "break" facility is always used to terminate the series of interactions.

```
repeat begin
    RequestLocator(ws, pointer, stat1, where1);
    if (stat1 = OK) then case (where1.tran) of
    OVERALL, MAGNIFIED:
        { Either part of line or of New View }
        case (i) of
        0:
            AddPoint(where1);
        1:  begin
                where2 := where1;
                InitialiseLocator(ws, pointer,
                    where1, RubberRect, data);
                i := 2
            end;
        2:  begin
                NewView(where1, where2);
                i := 0
            end
        end;
    GAP:
        { Hit in Gap command }
        BreakLine;
    NEWVIEW:
        { Hit in New View command }
        i := 1;
    end
end until (stat1 <> OK);
```

This window-oriented style of interaction fits well with modern hardware[6], but also fits into current developments in interaction support software[3, 8]. It provides close control over the details of the user interface, without sacrificing portability, but in effect requires the application to implement the logical input devices it needs itself, using mainly the LOCATOR.

5. Comparison of Styles

An application intended to be highly portable, and cheap to develop, can ensure this by using:

— only REQUEST mode input.

— at most one device in each class.

— the default device attributes, in particular the default prompt/echo type.

When these programs are transported, they may encounter sites where not all devices are present on a *single* workstation; some may be supported on additional, input-only workstations. Allowance for this should be made in the code.

An application can use knowledge about a particular configuration to exploit hardware capabilities (at the expense of portability) by using:

— a range of prompt/echo types, corresponding to those supported by the hardware.

— extra, unspecified entries in the data record to address specific hardware device attributes.

— several devices in one class, corresponding to several hardware devices.

— SAMPLE and EVENT modes, if the operating system supports them.

An application can provide detailed control over the user interface without sacrificing portability by assuming that at least a hardware pointing device will be available, and using the LOCATOR device as a basis for constructing its own logical input devices. The use of multiple normalisation transformations can assist this; providing the pointing device with a built-in selection device, and permitting a window-oriented style of interaction.

Acknowledgements

My grateful thanks to Dr. van der List and the whole staff of the Traumatology Dept., Wilhelmina Gasthuis, Amsterdam, who coped with a broken ankle and a bedside terminal with equal aplomb.

REFERENCES

[1] P. R. Bono, J. L. Encarnação, F. R. A. Hopgood, and P. J. W. ten Hagen, "GKS — The First Graphics Standard," *IEEE Computer Graphics and Applications* **2**(5), pp.9-23 (July 1982).

[2] ISO, "Graphical Kernel System (GKS) — Functional Description," ISO DP 7942 (January 1982).

[3] D. H. H. Ingalls, "The Smalltalk Graphics Kernel," *BYTE*, pp.168-194 (August 1981).

[4] D. S. H. Rosenthal, J. C. Michener, G. Pfaff, R. Kessener, and M. Sabin, "The Detailed Semantics of Graphics Input Devices," *Computer Graphics* **16**(3), pp.33-38 (July 1982).

[5] D. S. H. Rosenthal and P. J. W. ten Hagen, "GKS in C," Proceedings of Eurographics, Manchester (September 1982).

[6] D. S. H. Rosenthal, "Managing Graphical Resources," *Computer Graphics* **16**(4) (1982).

[7] SIGGRAPH-ACM (GSPC), "Status Report of the Graphics Standards Planning Committee," *Computer Graphics* **13**(3) (August 1979).

[8] J. J. Thomas and G. Hamlin (eds), "Graphical Input Interaction Techniques: Workshop Summary," *Computer Graphics* **16**(4) (December 1982).

[9] V. L. Wallace, "The Semantics of Graphics Input Devices," *Computer Graphics* **10**(1) (Spring 1976).

69 K 34
69 K 36