A STANDARD COMPUTER GRAPHICS PACKAGE - GKS

THE IMPLEMENTATION OF A STANDARD COMPUTER GRAPHICS PACKAGE - GRAPHICAL KERNEL SYSTEM

By

DEH-CHANG CHEN, B.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster Universty

March 1984

MASTER OF SCIENCE (1984) (Computation)

McMASTER UNIVERSITY Hamilton, Ontario

TITLE: The Implementation of a Standard Computer Graphics Package - Graphical Kernel System

AUTHOR: Deh-Chang Chen, B.Sc. (Chemistry, Fu-Jen University, Taiwan)

SUPERVISOR: Professor W.H. Fleming

NUMBER OF PAGES: viii, 84

ABSTRACT

Computer graphics is a field whose time has come. In the past, it was an esoteric specialty involving expensive display hardware and idiosyncratic software. Recently, hardware has become more readily available, and efforts have been made to develop graphics software standards, which help make graphics programming rational and straightforward.

The Graphical Kernel System (GKS) is rapidly gaining acceptance as a worldwide standard for computer graphics. The International Standards Organization (ISO) is in the final stages of converting GKS from its current status as a Draft International Standard (DIS) to an International Standard.

This report presents an overview of GKS and also discusses a subroutine library, that has been developed for use at McMaster University and is equivalent to "0a" GKS (the lowest level of GKS). This library, called GKSLIB, is written in FORTRAN 77, and could be used by a programmer to support a wide range of two-dimensional, passive graphics applications.

ACKNOWLEDGEMENTS

My acknowledgements begin with a sincere thank to Professor W. H. Fleming for his direction, encouragement, and patience during the preparation of this project. I also wish to express my appreciation to the members of the McMaster University Unit for Computer Science for the many forms of help provided during my studies with them. Finally, my family has been behind me all the way on this project, and I dedicate it to them.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 What is Computer Graphics?

Computer graphics may be defined as the creation, storage, and manipulation of models of objects and their pictures via computer [FOL82]. It is an extremely effective medium for communication between man and computer. Most people enjoy interacting graphically more than they do the more traditional and more limited alphanumeric communication techniques. With computer graphics, we are largely liberated from the tedium and frustration of looking for patterns and trends by scanning many pages of linear text on line printer listings or alphanumeric terminals.

In the past, however, the high cost of computer graphics technology has prevented its widespread use. Recently, the cost is dropping rapidly, and computer graphics is becoming available to more and more people.

1.2 Some Representative Uses of Computer Graphics

Computer graphics is used today in many different areas of industry, business, government, education, entertainment, and, most recently, in the home. The following list gives an idea of the areas of use to which graphics has already been put [FOL82].

- Plotting in business, science, and technology. This describes probably three-quarters of graphics application programs. Examples include graphs of mathematical, physical, and economic functions, histograms, bar and pie charts, task scheduling charts, inventory and production charts, and a profusion of other plots. All are used to present trends and patterns in data in a meaningful and concise fashion in order to increase understanding of complex phenomena and to facilitate informed decision making.

- Cartography. Computer graphics is used for the production of highly accurate representations on paper or film of geographical and other natural phenomena. Exmaples include geographic maps, weather maps, and population density maps.

- Computer-aided drafting and design. In computer-aided design (CAD), computer graphics is used to design components and systems of mechanical, electrical, and electromechanical devices. These systems include structures (such as buildings, automobile bodies, and chemical plants), and telephone and computer networks. The engineer can interact with a computer-based model of the component or system being designed in order to test, for example, its mechanical, electrical, or thermal properties.

- Simulation and animation. The most familiar example in this area is the flight simulator, in which computer graphics helps train the pilots of our airplanes on the ground. It has many advantages over real aircraft for training purposes, including fuel savings and safety.

- Process control. In some industrial applications, the user can interact with some aspects of the real world itself, rather than a simulation of the real world. Status displays for refineries, power plants, and computer networks display data values from sensors attached to critical components in the system; the operator then responds to exceptional conditions.

- Office automation. In the office and even the home, people now use the alphanumeric and graphic terminals to create and disseminate information which contains not just text but also tables and graphs.

- Art and commerce. Computer art and advertising have the common goal of expressing a "message" and attracting the attention of the public with aesthetically pleasing pictures.

1.3 Classification of Applications

The areas listed above can be categorized in a variety of ways. An obvious one is based on the type of picture generated: for example, whether two- or three-dimensional, or whether portraying an abstract or a real entity.

A categorization that is less drawing-oriented and more programming-oriented divides the application areas into three distinct ones in a spectrum: offline plotting with a predefined data base produced by other application programs, where an observer has little control over the appearance of the images; interactive plotting in which the user dynamically controls the pictures' content, format, size, or colors on a display surface by means of interaction devices such as a keyboard, lever, or joystick; and interactive design in which the user, starting from a blank screen, defines an object, typically from predefined components, and then alters it at will, panning and zooming to get the desired view [FOL82].

CHAPTER 2

OVERVIEW OF A GRAPHICS SYSTEM

2.1 A Programmer's Model of a Computer Graphics System

A graphics system consists of hardware and software components.

HARDWARE. Figure 1 symbolizes the hardware view of a graphics system. Two major hardware components are the host computer and the display unit. The display unit itself consists of a display processor (or DPU) and a CRT. It may contain some other input devices such as keyboard or joystick for interactive purpose.

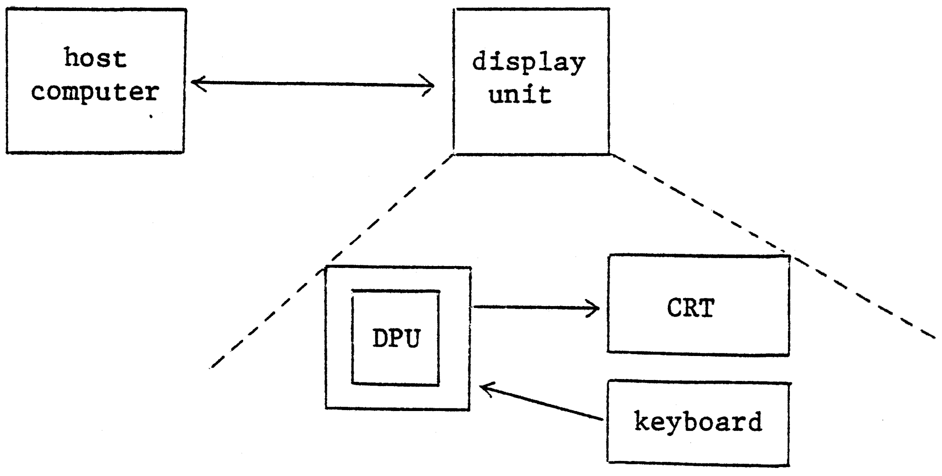

Figure 1 Hardware view of a graphics system

SOFTWARE and DATA MODULES. Figure 2 shows two important software components - application program and graphics package, and two data modules. The first data module is the DPU display program which is written by the graphics package and read by the DPU. The second data module is the application data structure which contains, among other things, a description of the objects whose images are to be displayed.

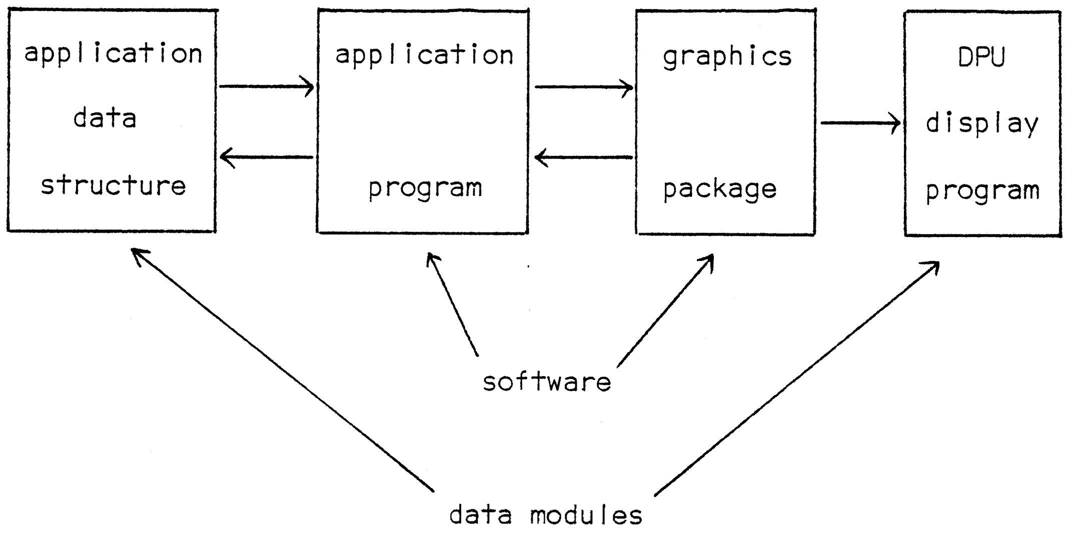

Figure 2 Software and data modules of a graphics system

Therefore, the application program retrieves data from the application data structure, and sends graphics commands to the graphics package, which, in turn, produces a display program for DPU to draw picture on CRT.

2.2 Graphics Hardware

From Figure 1, there are four major subsystems in a typical graphics system: computer, DPU, display device, and input devices. The computer is the heart of the system. The display device is usually a cathode ray tube (CRT). The DPU can be viewed as a special purpose CPU, with its own set of commands, data formats, and an instruction counter. It executes a sequence of display instructions (the display program), to create a drawing on the display device. Individual DPU instructions typically draw a point, line, or character string. Input devices, with which the user inputs commands and other information, are attached to the DPU.

There are two basic types of CRTs: refresh and storage. With a refresh CRT, the DPU interprets the display instructions and converts digital values to analog voltages which displace an electron beam writing on the phosphor coating of the CRT. Since the light output of the phosphor decays in tens or at most hundreds of microseconds, the DPU must cycle through this display program to refresh the phosphor at least 30 times per second to avoid flicker. In a storage CRT (also known as DVST), the image is stored (until erased) as an internal charge distribution, and thus a refresh cycle is not necessary.

The DPU can be organized to create a drawing either by random scan or raster scan. In a random-scan (or vector) system, parts of the drawing can be depicted on the display in any order. In a raster-scan system, the display primitives, such as lines, characters, and solid areas are stored in a refresh buffer in terms of their component points, called pixels (short for picture elements). The image is formed from the raster, a set of horizontal raster lines each made up of individual pixels. The raster is thus simply a matrix of pixels covering the entire screen area. The entire image is scanned out sequentially, 30 times per second, one raster line at a time top to bottom, by varying only the intensity of the electron beam for each pixel on a line. The storage needed is thus greatly increased in that the entire image of, say 512 lines of 512 pixels, must be stored explicitly in a bit map containing only points that map one-for-one to points on the screen. The more familiar hard-copy devices also operate with either a random or raster scan. The printer is a simple raster-scan hard-copy device. The print head moves from left to right, top to bottom. The pen plotter, in which a pen can be moved in any direction over a piece of paper, is a random-scan device. A raster system makes possible the display of solid areas, typically in color, which is an especially rich means for communicating information.

Input devices attached to the DPU are for the user to interact with the application program. Some more common examples include alphanumeric keyboard for entering text, programmable function keyboard (PFK) for invoking predefined options or functions, light pen for pointing at information displayed on the screen, data tablet with stylus for specifying screen coordinates, and control dials for entering the scalar values. The DPU contains a number of registers in which input devices store the appropriate values. Event (interrupt-) generating devices load their device registers and interrupt the CPU. The sampled devices load their register with data whenever they are interrogated by the CPU; the CPU then reads the associated registers.

2.3 What is a Graphics Package?

From Figure 2, a graphics package may be defined as a high-level programming interface between the application program and the graphics devices. The application program uses the graphics package much as it uses the I/O subsystem of the operating system to read and write records in file. The graphics package offers the application programmer a range of functions to use within his program and hides some less important details of the display's construction from the programmer, such as specific low-level architecture of the DPU and the xy coordinate system of the physical screen. By this way, the graphics package can simplify the writing of the graphics application program.

The functional facilities of a graphics package available to the application programmer can usually be divided into some distinct classes. Below is a typical classification:

1. graphic output primitives,
2. viewing specification,
3. attribute-setting,
4. segment control,
5. input,
6. control.

Figure 3 is an expanded version of Figure 2, and gives a more detailed view of the structure of a graphics system. It shows three major processors of a graphics package:

1. viewing operation processor,
2. DPU code generator,
3. input handler.

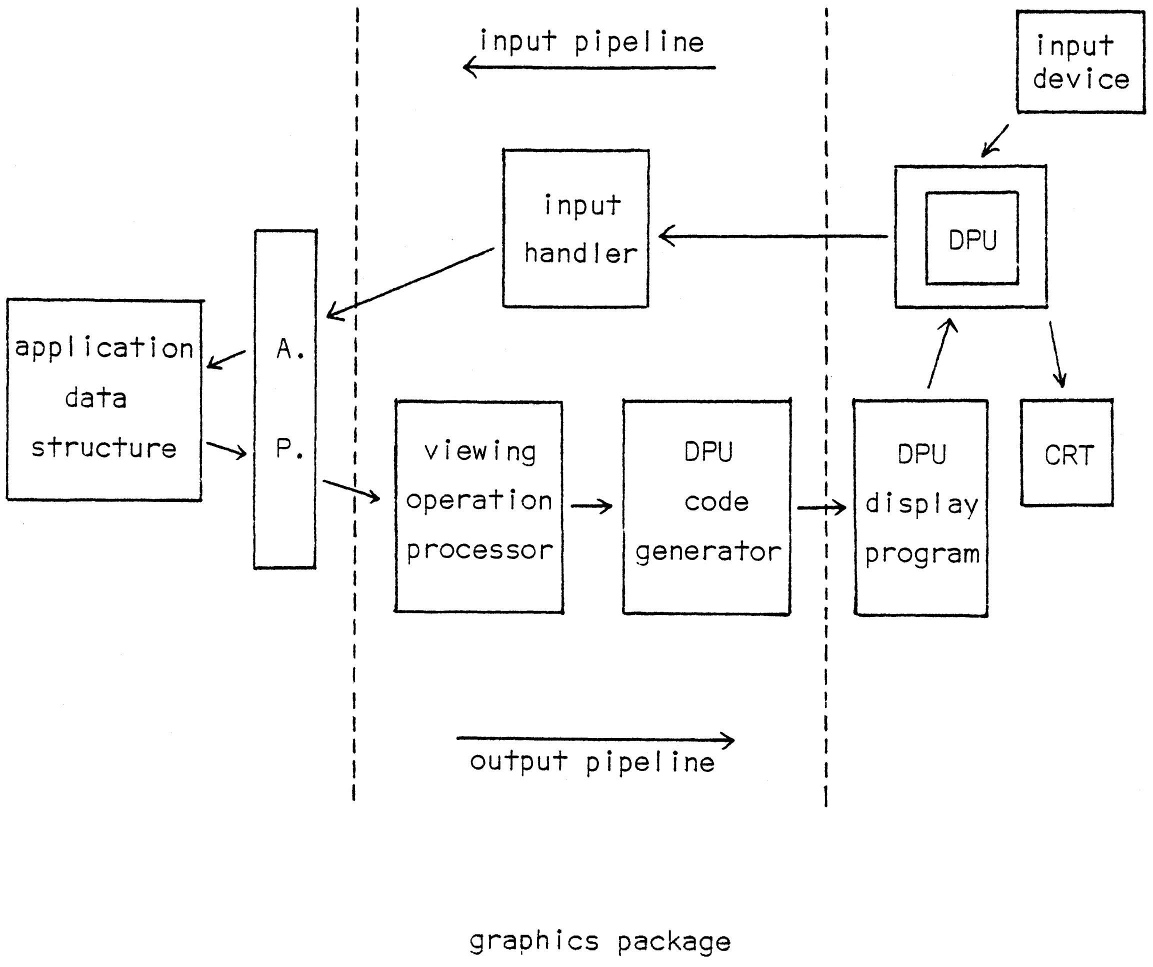

Figure 3 Detailed view of a graphics system

With this diagram, it should become easier to explain how the six classes of functions provided in a typical graphics package can be invoked and used in an application program. There are two data flows going in opposite directions within the graphics system: one is from the object description in an application data structure to an image (or picture) on the screen, the so-called output pipeline, and the other is from the user-supplied input to the data structure and/or display program, the so-called input pipeline.

2.4 Viewing Operation Processor

At the first stage in the output pipeline, the application program retrieves a piece of data from an application data structure. The application data structure contains geometric data and connectivity relationship data. The former defines the shape of components of the object, the latter defines how the components fit together. The application programmer must first construct this data structure before he can use it and describe it to the graphics package for viewing purposes.

The application program describes the data structure to the graphics system by transforming it to a sequence of function calls to the graphics subroutine package as Figure 4 shows. One of the most important function classes is the graphic output primitives. They are the functions an application programmer can use to display straight lines, text strings, and other simple graphical items. For example, we can use a function in this class called POLYGON to draw a triangle on the screen:

```
POLYGON(x_array,y_array,n),
```

where x_array and y_array contain the coordinates of the vertices of the polygon, and n gives the number of vertices.

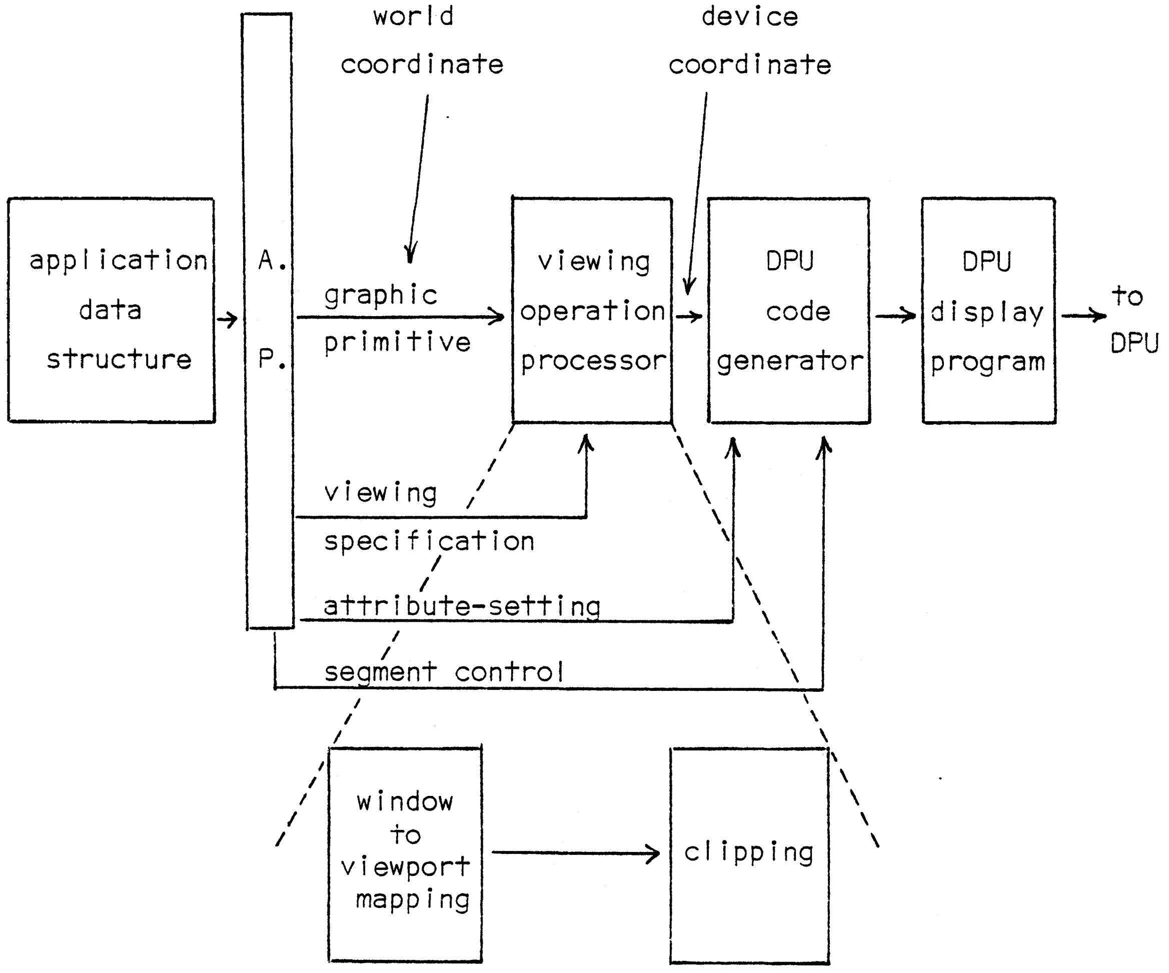

Figure 4 Function calls to the graphics package

One very important concept here is that the vertices of the polygon are specified in the user's world coordinate system. That means, we can define objects in terms of units that are natural to the application and to the user. These world coordinates must be converted into the appropriate coordinates of the physical display device. To make this conversion, we have to tell the graphics system what portion of the essentially unbounded world coordinate space contains the information we want to display at this time. Figure 5 shows an example.

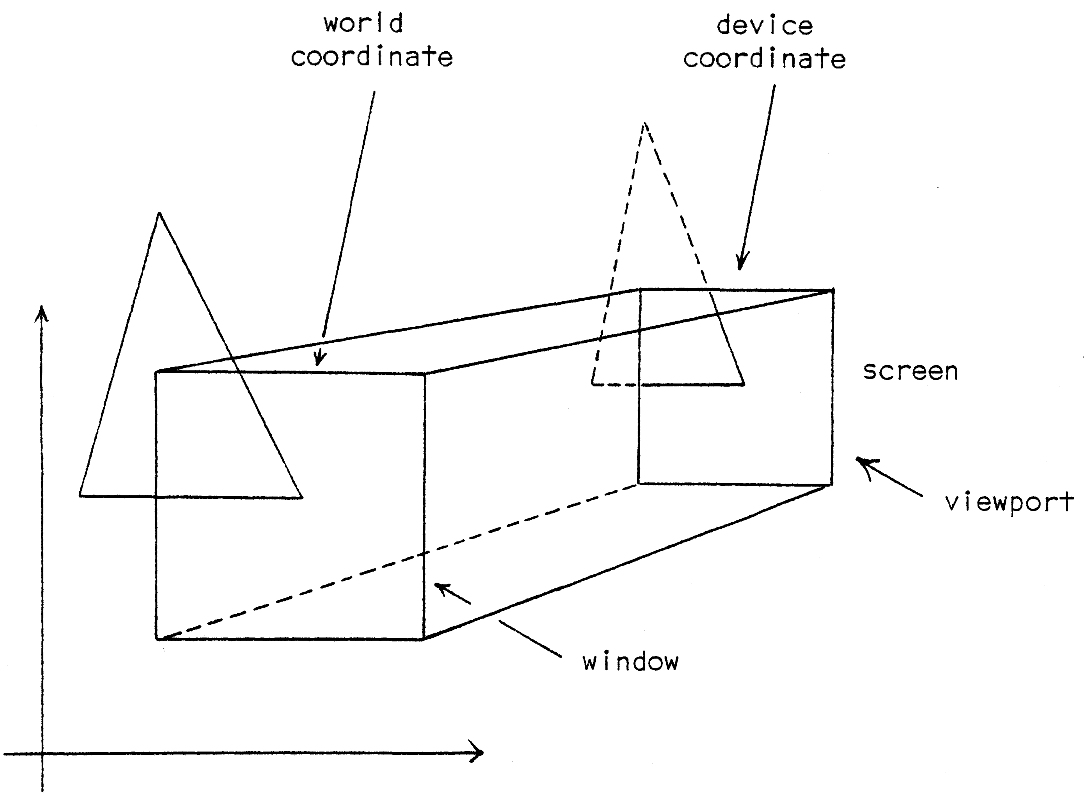

Figure 5 Conversion from the world to the device coordinates

The rectangular region in the world coordinate space is called a window. It can be specified to the graphics system by calling one of the viewing specification functions:

SETWINDOW(x1,y1,x2,y2),

where (x1,y1) is the lower left corner and (x2,y2) is the upper right corner.

In addition to displaying the picture on the entire screen, we can also map a window onto some portion of the screen. This rectangular portion is called a viewport, which can be defined by invoking another viewing specification function:

SETVIEWPORT(x1,y1,x2,y2).

Figure 6 gives an example.

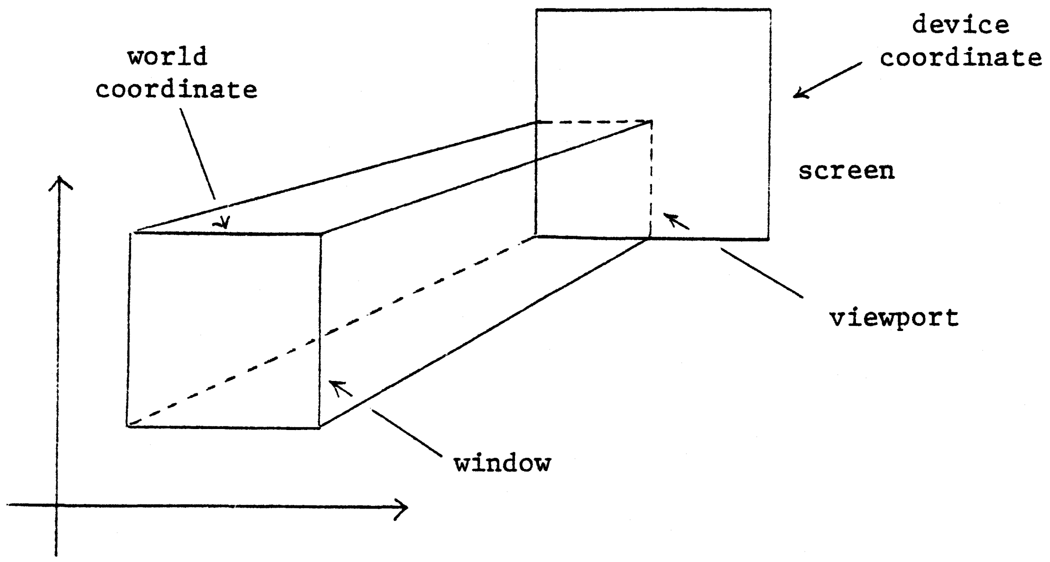

Figure 6 Mapping the window onto some portion of the screen

With these two viewing specification functions, SETWINDOW and SETVIEWPORT, we can choose any portion in the world coordinate space to be displayed on any portion of the screen. The viewing operation processor of the graphics package will do the necessary transformation to all graphic output primitives it receives.

From Figure 4, there is one other important task the viewing operation processor must perform beside window-to-viewport mapping, namely clipping. Clipping is a technique used to make any parts of the object outside the window invisible. Figure 5 shows one case where clipping is needed. If this is not done, the results on the screen would not be well-defined, because we try to display points that overflow the coordinate addressing scheme of the display.

Clipping can be done after mapping as shown in Figure 7. In this case, pictures (or graphic output primitives) are clipped against the viewport. That means, the viewport is used as clipping rectangle. The disadvantage of this method is that all the primitives must be transformed whether they are visible or not.

One alternative way is "clipping before mapping" as shown in Figure 8. Here, pictures are clipped against the window.

2.5 DPU Code Generator

The function of this processor is to produce a display program for DPU. The DPU display program is a sequence of point and line plotting commands and character plotting commands which are encoded in a

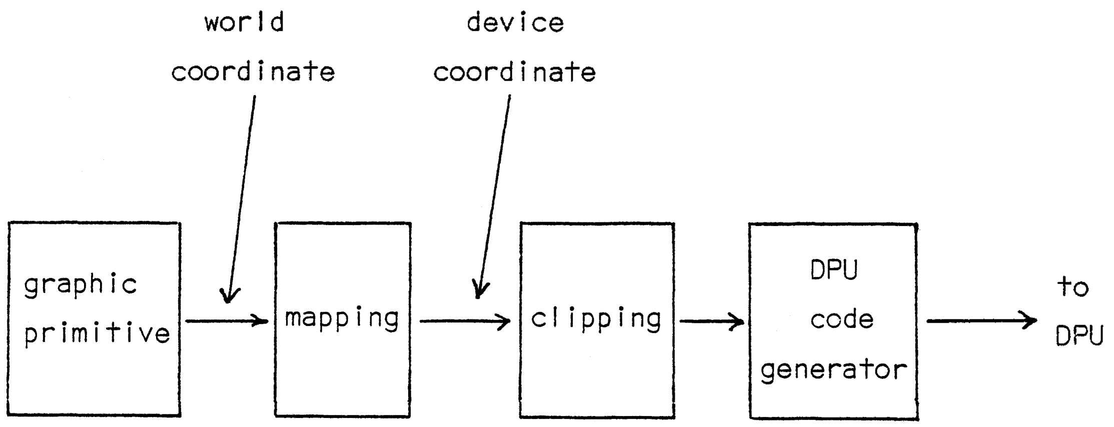

Figure 7 Mapping, then clipping

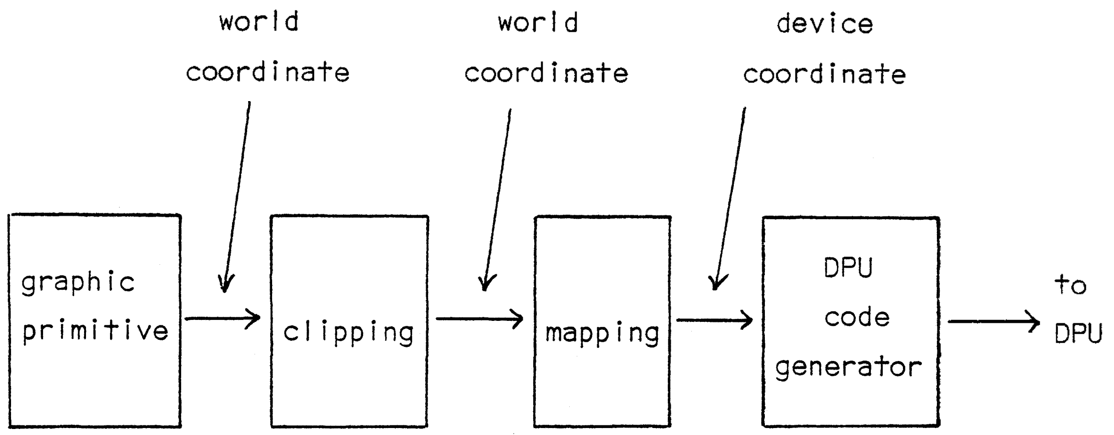

Figure 8 Clipping, then mapping

specific format suitable for DPU to interpret in order to draw points, lines, and character strings on the screen. Actually, it can be compared to the "machine code" from the normal compilation. Then, the graphics package can be thought of as a "display program compiler". Two significant differences between the compilation of the DPU code and the normal compilation are: (1) The compilation of the DPU code takes place at the run time of the application program, while normal compilation is usually finished before application program starts executing; (2) The ordinary source code is compiled to equivalent target code in its entirety, while graphics package's "source" code may be clipped to a subset before being compiled to equivalent DPU "target" code.

From Figure 4, two classes of functions provided in a graphics subroutine package are relevant to DPU code generation. They are attribute-setting and segment control. Attribute-setting functions control the appearance of the graphic output primitives. For example, line style and line colour can be set for all following primitives until reset. Segment control functions are used to group logically related output primitives into segments. Segments are the units of selective modification of the display program. Functions are also available to delete, rename, or change the visibility of segments.

2.6 Input Handler

While picture plotting is handled by the graphics package's output routines, input handling is controlled by its input routines that pass user-supplied input data to the application program as part of an interaction sequence. The input data is first collected from the DPU by the input handler, which typically then passes it to the application program. The data changes the state or flow of control of the application program. It may also cause the application program to modify either the data structure or to change the viewing operation parameters. The input may also be used directly by the code generator to perform segment manipulation operations.

A major goal of the input facilities of a graphics package, as of output devices, is device-independence. This is achieved by organizing all the physical input devices into five basic logical devices:

1. button, to select an option;
2. pick, to point to a displayed entity;
3. keyboard, to enter a character string;
4. valuator, to input a scalar value;
5. locater, to specify screen coordinates.

Program requests for input functions specify a logical device name which the input handler maps to the available physical device with the most naturally corresponding characteristics. This mapping of logical to physical devices is analogous to an operating system's mapping of logical unit numbers or logical file names to appropriate physical file storage devices.

Each logical device has a natural prototype in a specific physical device or class of devices. However, any of these logical devices can be simulated by any input device. This concept again is rather like that of logical files in an operating system. A sequential input file may be implemented physically by means of a card reader, a magnetic tape drive, a disk drive, or a terminal keyboard. The application programmer doesn't care which one it is - the operating system makes them all "look alike" functionally, despite their physical differences.

CHAPTER 3

GRAPHICS STANDARDIZATION

The aim of graphics system design is to simplify the writing of graphic application programs. The earliest applications were written without the benefit of graphics systems, and were very difficult to write. Nowadays it is universal practice to use a graphics package as the basis for applications development. With this approach, applications take less time to write, and their development demands less skill on the part of the programmer [NEW79].

However, almost all of the eralier graphics systems were restricted to certain mainframe computers, to a given host language, and to specific graphics devices. Moreover, most of the systems addressed a single application area. These graphics subroutine packages were usually supplied by manufacturers for their unique display devices, and varied from the very simple package for two-dimensional pictures on a storage display, up to the most complex systems supporting three-dimensional graphic data structures. Since each of these systems supports a different set of functions and requires the use of different programming conventions, they are all very low-level and machine and device dependent. The result is that if an application program is written to use one of these systems, the chances are very remote that it can be run in conjunction with another system. This is the problem of program portability. Another disadvantage is the need to retrain the programmers to use different graphics systems. This problem concerns programmer portability [NEW79].

3.1 Device Independence

In order to achieve application program (and programmer) portability, the graphics packages must present a uniform interface to the application programmer, no matter what equipment is being used. Whether the output device is a plotter, a storage display, or a high-performance refresh display, the programmer should be able to use the same set of graphics functions to generate images. These packages are thus device independent at the level of the programmer's interface.

One other form of device independence also needs consideration: device independence within the package. This permits the package to drive different devices with the minimum of modification for each new device. Device independence in a graphics package can be achieved by carefully separating those components of the graphics system that are inherently device-dependent from the remaining common software, and by giving equally careful attention to the interface between the two parts. Large sections of the package, including transformation and clipping software, can usually be included in the common software. The most device-dependent parts of the package are likely to lie in the input device polling routines and in the DPU code generator. They can be partitioned in a module and called the logical device driver. Device dependence can be kept to a minimum, if interfaces from the common software to these routines are well-designed.

The interface to the DPU code generator may take the form of either an intermediate data structure, or a set of functions within the display code generator, called by the common software. Many plotter-oriented graphics packages use an intermediate data structure, in which the entire image for plotting is stored in a device-independent format by using normalized device coordinates (NDC); the data structure is then translated to the format required by the device. NDC can be thought of as a logical coordinate system used for describing the view surface of a logical output device. They are real numbers in the range from 0 to 1 in both x and y, with the origin in the bottom left corner of the view surface. In an interactive environment, the use of an intermediate data structure amounts to an extra buffering step, impacting response and requiring additional memory. It is therefore rarely used in interactive graphics package, except as "pseudo display files" for off-line plotting, or for storage of images for later re-use. A pseudo display file is often referred to as a metafile.

3.2 Standardization Aspects

A significant development that started in the mid-seventies was a general awareness of the need for standards in such device-independent graphics packages. Anyone who has been involved in building graphics system knows that the lack of unity and maturity in the field has been a source of discourgement to many potential users, and has slowed progress towards wide acceptance of computer graphics. Although standards in programming languages were common very early on, the standards in computer graphics are long overdue. The lack of standards is not due to the youth of the technology. By 1965, most hardware technologies used today were already in existence. One clue to the delay is the wide diversity among graphics hardware devices and among graphics packages developed for them [BON82].

3.2.1 Why is the Standardization Necessary?

The single strongest justification for standardization in computer graphics is the promotion of program portability and programmer portability [NEW78]. Portability, in turn, reduces software costs and personnel training costs. It was also found that standardization improved communications from the user's point of view.

From the manufacturer's viewpoint, improved portability increases the size of the market. The standard itself gives guidance as to the right directions for hardware innovation.

Moreover, a standard that encourages machine and device independence also protects the hardware and software investment of the end user. New computer systems and new hardware devices may be added as technology advances and as the demands of the application change.

In short, a standard serves as the base for a common understanding and a common terminology for creating computer graphics systems, for using computer graphics, for talking about computer graphics, and for educating students in computer graphics methods, concepts, and applications.

3.2.2 Standardization Requirements

In order to meet the needs of most users, a computer graphics system should separate the basic graphical capabilities from those functions that are related to a specific application area. A system realizing the basic graphical capabilities is called a "core system", the application dependent systems using the functions of a core system are referred to as "modelling system". For example, in a geometric modelling system, the handling (definition, transformations, calculations, storage) of the geometrical models of the design parts is done by the modelling system, whereas the graphical presentation of the models and the interactions with an operator is the task of the core system. A core system should have the following properties [END83]:

- independence of a specific computer,
- independence of specific graphical devices,
- independence of a specific programming language,
- independence of a specific application area.

Thus, a standardized computer graphics system should define a standard functional interface for all kinds of applications, a standard device interface to all kinds of graphics devices, and a standard interface for storage and transfer of graphical information ("graphics metafile"). Figure 9 shows a graphical core system with the application interface and the device interface.

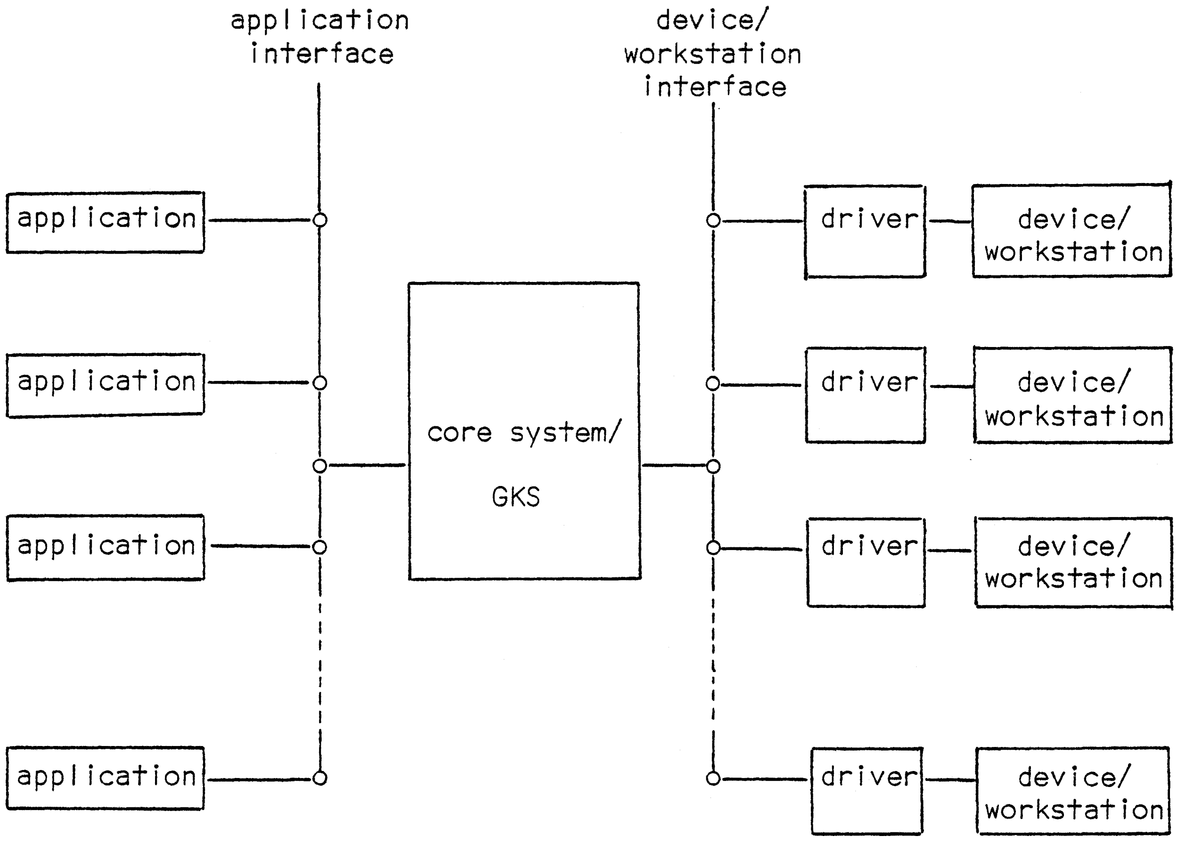

Figure 9 Core system with application and device interfaces

3.2.3 The Development of Graphics Standards

The development of graphics standards began in 1976 following an extremely successful international "Workshop on Graphics Standards Methodology" in Seillac, France. In the US, the results of this workshop stimulated ACM SIGGRAPH′s Graphics Standards Planning Committee (GSPC). This committee designed a proposal for a 3D graphics core system. Two versions of this proposal were published in 1977 and 1979. The GSPC′s proposal has become known as GSPC-Core or just "the Core".

Efforts to develop graphics standards were also underway in a number of other countries. In particular, the German Standardization Institute, DIN (Deutsches Institut fuer Normung), established a group aimed at designing a graphics core system. This group produced several versions of the Graphical Kernel System (GKS) [BON82].

In 1979, GKS was selected as the base for the international standardization effort in the computer graphics field by the working group Technical Committee 97 (Information System)/Sub Committee 5 (Programming Languages)/Working Group 2 (Graphics), normally abbreviated TC97/SC5/WG2, of ISO (International Standardization Organization). Up to 1982, GKS was subject to an extensive international reviewing process. In several revisions it finally reached a state that agreement could be reached that GKS should be an international standard. It presently has the status of a DIS (Draft International Standard). ANSI, the American National Standards Institute, is also in the process of adopting the Graphical Kernel System as an ANSI standard [STR83].

3.2.4 GKS vs GSPC-Core

Whereas GSPC-Core was intended to be a comprehensive and comfortable 3D standard, GKS, a 2D system, was aimed at the basic graphical functions. This was one of the reasons why the ISO, after evaluating both GSPC-Core and GKS, decided to use GKS as the base for an international graphics standard [END83]. A basic system is also referred to as a "basic core", in contrast to a "rich core" which contains a broader spectrum of functionality. A more detailed comparison between GKS and

GSPC-Core will be given in Appendix I.

CHAPTER 4

AN OVERVIEW OF GKS

The GKS standard specifies a set of functions for computer graphics programming in a way that is independent of particular graphics devices, computers, programming languages, or applications. A fundamental concept in GKS is the workstation, consisting of a number of input devices and a single output device. The workstation concept is important for achieving device independence, while still allowing full control of physical device characteristics. The capabilities provided by GKS include the following [ISO82]:

* two-dimensional line and raster graphics,
* graphics input and output at one or more graphics workstations simultaneously,
* provision for storage and dynamic modification of pictures,
* storage and retrieval of graphics information from a long-term, external graphics file (metafile),
* means for adapting application program behaviour to suit workstation capabilities,
* several upwardly compatible levels of the standard with increasing functional capabilities.

4.1 Layer Model of GKS

GKS defines a language independent nucleus of a graphics system. However, in an implementation of the system, these functions have to be realized as subroutines (or procedures) in a given programming language. Such a language specific realization, in which the language-independent system nucleus is embedded, is called a language layer. The functions provided by the language layer can be used by the application programmer, together with operating system functions. Special application dependent layers can be built on top of the GKS language layer (e.g. a layer for data representation graphics). The layer model represented in Figure 10 illustrates the role of GKS in an application. Each layer may call the functions of the adjoining lower layers. So an application program will have access to a number of application oriented layers, the language dependent GKS layer, and operating system resources.

4.2 GKS Workstation

A GKS workstation consists of a single display area and a number of input devices. The whole workstation is treated in GKS as one logical unit and operated in a coordinated fashion by an operator at a given site. An operator can have a number of GKS workstations under his control at the same time. For example, he may be interacting at a refresh display while taking occasional copies of output at a plotter. Different workstations may be set to view different parts of the complete virtual picture.

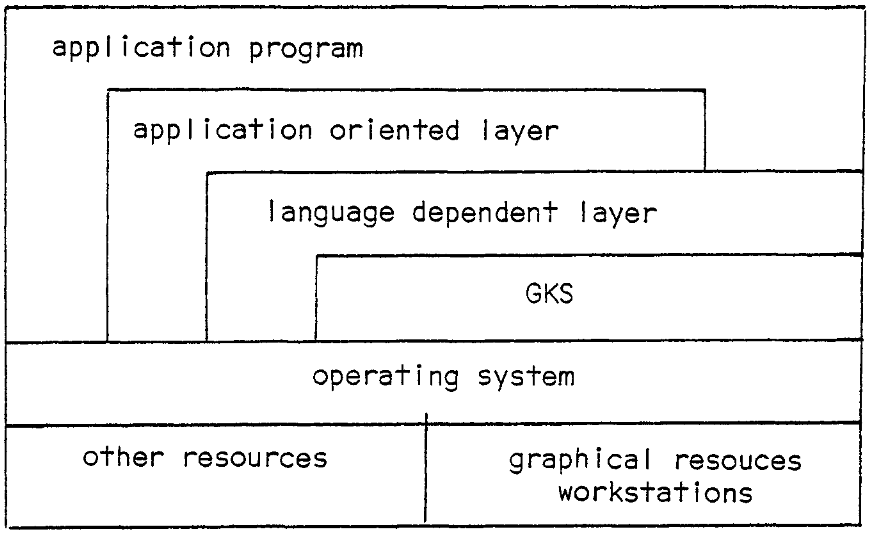

Figure 10 Layer model of GKS

Each workstation has a type. Workstation types are similar to the facilities that would be available at a plotter, storage tube, or refresh display. Each workstation type falls into one of six categories:

OUTPUT	Output,
INPUT	Input,
OUTIN	Output and input,
WISS	Workstation Independent Segment Storage,
MO	GKS Metafile (GKSM) output,
MI	GKSM input.

For every type of workstation present in a given GKS implementation, an entry exists in a workstation description table. It describes the capabilities and characteristics of the workstation. The workstations are identified by the application program by use of a workstation identifier. Whenever the application program wants to use a workstation, it must first request the opening of this workstation by GKS, which associates the workstation to the corresponding graphical terminal and gives the application access to all its capabilities except output of graphic primitives. For output the workstation must be explicitly activated. Output primitives are sent to all active workstations. Segment manipulation and input can be performed with any open workstation.

4.3 GKS Attribute Bundles

Graphics primitives such as line drawing can have associated attributes such as color, thickness, and line style. There are basically two approaches to specifying such attributes. The first is to have a set of modal attributes that are in effect until the next setting of the attribute. For example:

```
COLOR (RED)
WIDTH (THICK)
STYLE (SOLID)
DRAW LINE
COLOR (GREEN)
STYLE (DASHED)
DRAW LINE
```

This would draw a thick, red, solid line followed by a thick, green, dashed line. Each modal attribute remains in effect until reset. Thus, thickness is an attribute to both lines. A disadvantage of this approach is the need to map this attribute specification onto a number of devices that may not be able to implement a particular attribute. How can we draw red lines on a storage tube? Usually the implementor of the device driver makes an arbitrary decision. A second disadvantage of this approach is the specification of library routines where particular lines must be differentiated but the application programmer is left to specify the particular attribute to use. For example, a contour routine might need every third contour to be highlighted. The application programmer might wish to use color, thickness, or broken lines to highlight the effect. With modal attributes, the body of the algorithm becomes complex, with many attribute settings depending on user's requirements.

The solution adopted in GKS is to have one workstation-independent attribute per primitive designated as the primitive index. Each primitive may have one of a number of representations associated with it, running from 1 up to an implementation maximum. The equivalent GKS program to the one above would look like this:

```
SET POLYLINE INDEX (1)
POLYLINE
SET POLYLINE INDEX (2)
POLYLINE
```

The first line would be drawn with the bundle corresponding to index 1 and the second with the bundle corresponding to index 2. The representations of bundles 1 and 2 are workstation-dependent and can be set by the application programmer. Thus, he can set representation 1 as red, thick, and solid, while representation 2 is green, thick, and dashed. The advantage of making the pen specification workstation-dependent is that the characteristics of representation 1 can be quite different on two workstations. Many of today's application programs suffer greatly from the problems associated with producing appropriate output on workstations with diverse capabilities. GKS workstation model makes it easy to use different graphics devices, and particularly easy to use the best features of each device. However, for the single workstation environment, GKS has given up the simpler, more direct approach of workstation-independent, unbundled primitive attributes in exchange for increased flexibility. A set of indicators, called Aspect Source Flags (ASFs), can be used for this purpose. They control whether the values of the associated attributes are obtained from a bundle table or from individual specifications.

4.4 Output Primitives

GKS has defined six output primitives:

POLYLINE,

POLYMARKER,

TEXT,

FILL AREA,

CELL ARRAY,

GENERALIZED DRAWING PRIMITIVE.

However, unlike many present-day systems, GKS does not use the concept of current position. Each primitive has its coordinates fully defined internally. Furthermore, for line drawing, a polyline, which generates a set of connected lines given an array of points as a parameter, is the fundamental line drawing primitive. A polyline primitive is more practical than a single line, since a set of lines is more useful in forming a shape. Given that polyline rather than line is the basic primitive, attributes such as linestyle apply to the complete polyline rather than a single line segment. Thus, dotted or dashed curves are easily drawn.

A mechanism must also be defined for identifying points. GKS extends the point primitive to that of a marker that can output one of possible forms centered on a specified position. The basic primitive is a polymarker that output a sequence of markers and is the obvious primitive to choose, once polyline has been defined.

Text similarly produces a string of characters rather than a single character, to ensure some equivalence of level among the three main output primitives.

The remaining three primitives show the increasing importance of raster graphics and the need to allow hardware facilities to be used even within a device-independent standard. Fill area defines a boundary whose interior can be hollow, filled in solidly, or filled with either a pixel pattern or a hatching pattern. The cell array primitive is a means of specifying an array of colors or intensities and is particularly useful in image processing applications. The final primitive, GDP, is an escape function to allow special geometric primitives such as circle or curve to be defined in a well-defined, implementation-specified way - a standard way of being nonstandard.

4.5 Attributes

Polyline and polymarker have a single attribute, which selects a bundle as follows:

polyline	: linetype, linewidth scale factor, color index;
polymarker	: marker type, marker size scale factor, color index.

The color index selects an entry in a workstation-dependent color table, which specifies the RGB values to be used when drawing the primitive.

Text differs from the other primitives in splitting the attributes into two classes:

geometric attributes	: character height, character up vector, text path, text alignment;
nongeometric attributes	: text font and precision, character expansion factor, character spacing, text color index.

The first class controls the geometric aspects of the text. These attributes are workstation-independent and are expressed in world coordinates where appropriate (eg character height). The second class controls the nongeometric aspects of text such as font, precision, and color. The motivation for this split is that the overall form and shape of the text must fit with the graphic output on all devices and so should be device-independent. However, the particular character forms and quality of characters drawn may differ among workstations and should, therefore, be part of the bundle table. Workstation independent attributes are set modally. There is a current value for each workstation independent attribute.

Fill area also has two sets of attributes. The first comprises the following:

(1) Interior style defines the mode of filling: hollow, solid, pattern, or hatch;

(2) Style index specifies for pattern an entry in a pattern table, which is used for filling. If the interior style is hatch, the index is used to determine which of the predefined hatch styles is used;

(3) Color index is used for hollow and solid and is a reference to the color table.

The second set comprises two workstation-independent attributes: pattern size and pattern reference point. They define the size and position of the start of the pattern.

GDP has no separate bundle table, but uses the one most appropriate to the type of primitive it most closely resembles. Cell array has also no separate bundle table, but its definition includes an array of color indices.

4.6 Segments

It is possible to generate output primitives so that they are displayed on all active workstations. However, in an interactive environment the complete picture frequently needs to be split into a number of objects or segments that can be manipulated independently. For example, highlighting a particular part of the picture, or removing it for some reason. In working with a refresh display, a user must often move parts of a picture around. This is achieved via a segment transformation matrix, which may be altered after the segment is defined.

Segments are stored on only those workstations that are active when the segment is defined. This is adequate for most purpose, but occasionally we need to see a segment on a workstation that was not activated when the segment was created. For example, the user may be defining a picture made up of segments on a refresh display and then at some stage may wish to copy the current display to a plotter. GKS allows this through a workstation-independent segment storage (WISS), which can keep copies of segments as they are formed. When a copy is required, the segments can be sent from WISS to a specified workstation. In the more complex implementation levels of GKS, a segment can also be inserted into another segment.

4.7 Viewing

The typical graphical package has a single window/viewport transformation that allows the application programmer to define his own coordinate system, some part of which is mapped onto an area of the display screen. The situation is complicated in GKS by having several workstations active at the same time. Should all workstations, then, be forced to use the same viewport? An application might require one display to give an overall view of the picture being displayed, while another looks at the detail of the picture.

GKS achieves this flexibility through three different two-dimensional Cartesian coordinate systems and two distinct window/viewport mappings. The applications programmer defines his output in terms of a world coordinate (WC) system mapped onto some part of the normalized device coordinate (NDC) plane. This first-stage mapping is called normalization transformation. The set of active workstations can then take separate views of the NDC space and map these onto workstation-dependent parts of the display, expressed in device coordinates (DC). This second-stage mapping is called workstation transformation.

Any complex picture probably consists of several distinct parts, which are most appropriately defined in different coordinate systems. A conventional package would do this by allowing the user to continually redefine the window/viewport mapping from WC to NDC. For example,

```
SET WINDOW(XMIN,XMAX,YMIN,YMAX)
```

```
DRAW PICTUREA
SET WINDOW(X2MIN,X2MAX,Y2MIN,Y2MAX)
DRAW PICTUREB
```

In this hypothetical package, PICTUREA is drawn with the first coordinate system, whereas PICTUREB is drawn with the second coordinate system. The user effectively sees a display made up of two parts with different coordinate systems. The user's view of the system is that both coordinate systems must be known to the system as pictures. However, in reality only the second coordinate system is known in a conventional package. When the user needs to point to a particular position in either coordinate system, the system cannot deliver the position in the correct coordinate system.

To ensure that the user's view of the system is correct, GKS allows the definition of multiple window/viewports, all existing simultaneously. The GKS equivalent of the above program would look like this:

```
DEF WINDOW(1,XMIN,XMAX,YMIN,YMAX)
DEF WINDOW(2,X2MIN,X2MAX,Y2MIN,Y2MAX)
SECLECT WINDOW (1)
DRAW PICTUREA
SECLECT WINDOW (2)
DRAW PICTUREB
```

The form of the program in GKS has a tendency to define all the coordinate systems at the start of execution and then select the particular transformation when required. The other program form has transformation definitions scattered throughout.

4.8 Input

GKS input is defined in terms of a set of logical devices:

choice,

locator,

pick,

string,

valuator.

A logical device may be implemented on a workstation in a variety of ways. For example, a string may be input using a keyboard, by freehand drawing on a tablet, or by hitting a set of light buttons indicating particular characters on a display. The exact form of the implementation is up to the workstation.

Input can be obtained in three distinct ways:

(1) Request. This is rather like a Fortran READ. The system waits until the input event has taken place and then returns the appropriate values.

(2) Sample. The current value of a GKS input device is examined. This input mode is most frequently used for devices that have a continuous readout of their value. For example, the current position of the stylus on the digitizer can be sampled.

(3) Event. This mode allows a user to generate input data asynchronously. He may adjust input devices to special values and hit specific "triggers" so that the system can take over the adjusted values. For example, a light-pen hit normally generates an event.

4.9 GKS Levels

GKS has a level structure, which defines three input levels and three output levels, such that one implementation can choose any input level and any output level and combine the functions in each to define a valid level of GKS. By this way, the GKS system can be implemented to be usable by a wide range of applications, from static plotting to dynamic motion and real time interaction.

The output level axis has the three possibilities:

0: Minimal output,

1: Basic segmentation with full output,

2: Workstation Independent Segment Storage (WISS).

The input level axis has the three possibilities:

a: No input,

b: REQUEST input,

c: Full input.

CHAPTER 5

IMPLEMENTATION

The primary objective of this project was to develop a graphics subroutine package, based on the capabilities described in the GKS specification, that could be used by a programmer to support a wide range of 2-D passive graphics applications. The McMaster University Cyber 170/730, running NOS 2, was used for developing this GKS implementation. The language chosen for this purpose is FORTRAN 77. At present, the library includes nearly 85% of functions provided in the level "0a", which is the lowest level of GKS. Moreover, instead of generating pictures directly on a graphics device by using a device driver, the current status of the GKS implementation uses a metafile generator to produce a metafile for off-line plotting, or for storing graphic images.

The metafile produced from an application program on the Cyber can be transported to the Digital Equipment Corporation PDP 11/23 minicomputer, running UNIX. There a metafile reader, written in C, has been developed to interpret the graphical information stored in the metafile, and then draw the pictures on an AED raster display. The modification and expansion of this metafile reader is still in progress.

5.1 Language Consideration

GKS is defined independently of any particular programming language, so it is necessary to bind the abstract functions and data types of GKS to actual functions and data types in the language to be used for implementation. The language chosen for this project is FORTRAN 77, partly because it is a suitable, widely-used language for scientific programming. A FORTRAN implementation is suitable for a larger number of users, because an interface between FORTRAN subroutines and programs in higher languages like PASCAL may be provided rather easily - in contrast to the reverse direction, which was never solved in a satisfactory manner.

GKS has a published binding to FORTRAN. This binding specifies the actual names and argument sequences for graphics functions, and can thus promote application program portability. If an application is written for one vendor's GKS package, and another vendor's version is substituted later (perhaps to run on a different host computer, or to use a new device not supported by the first vendor's GKS), the original code has an excellent chance of being able to run without any modification. Although the current GKS implementation doesn't employ this FORTRAN binding, it should be easy to perform the necessary conversion in the future.

FORTRAN 77, however, presents some problems for the language binder. First, FORTRAN naming conventions restrict all the variable and subroutine names to a maximum length of 6 characters (7 on Cyber). This prevents maintaining mnemonic content in the names. Second, the data abstraction facilities in FORTRAN are not very sophisticated. For example, it does not support enumerated data types, although this can be fixed by using integer data type and defining constants with integer values, so the GKS programmer can use the same nice names he would have had with an enumerated data type.

5.2 Level 0a

The ISO GKS document is 285 pages long. Because the entire sections on segment functions and input functions are not needed by "0a" GKS, the majority of those pages can be ignored.

Full GKS (the highest input and output levels combined) includes 110 functions plus 75 inquiry functions. The lowest level ("0a") requires 53 functions plus 38 inquiry functions, of which 44 functions plus 33 inquiry functions have been included in this project. They are listed in Appendix II, together with their actual names used in this FORTRAN implementation of GKS.

The data structures required by "0a" GKS are also significantly smaller than those defined for full GKS in the ISO GKS document. The segment state lists and input queue disappear completely. What "0a" GKS requires for GKS state list, workstation state list, and workstation description table are given in Appendix III, together with the variable and array names used. The list of errors that can be generated by "0a" GKS is given in Appendix IX.

5.3 Device Drivers

In most computing environments an application program generates pictures on a graphics device as shown in Figure 11 [GSP79].

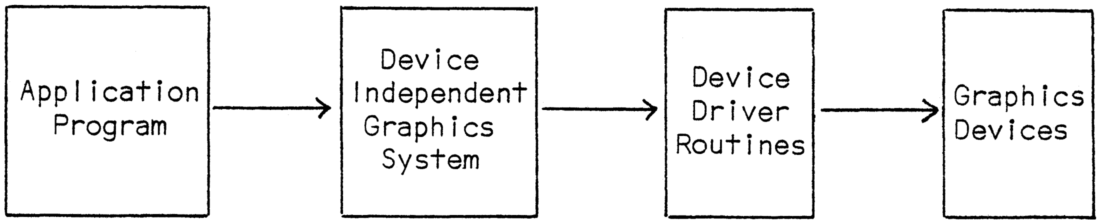

Figure 11 Picture generation by using a device driver

Metafile generation capabilities could be included in such a system as shown in Figure 12.

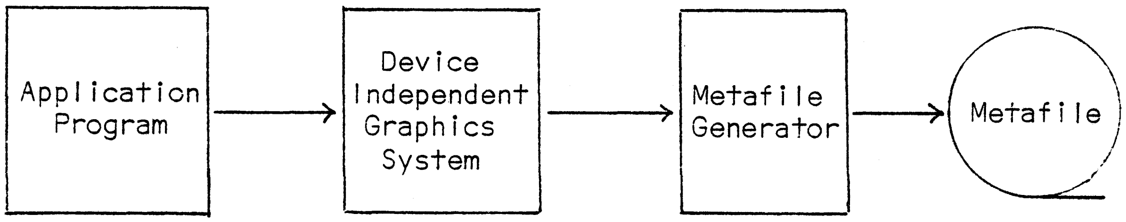

Figure 12 Metafile generation capability

The metafile generator can be thought of simply as another device driver for a virtual device. However, it should be stressed that the metafile generator can assume nothing about the graphics devices onto which metafile pictures will eventually be output.

Each computing facility that wishes to generate graphics output from a metafile input must provide a Metafile Reader to interpret the device independent metafile commands and either invoke routines in the Device Independent Graphics System (Figure 13) or directly call routines in a Device Driver (Figure 14).

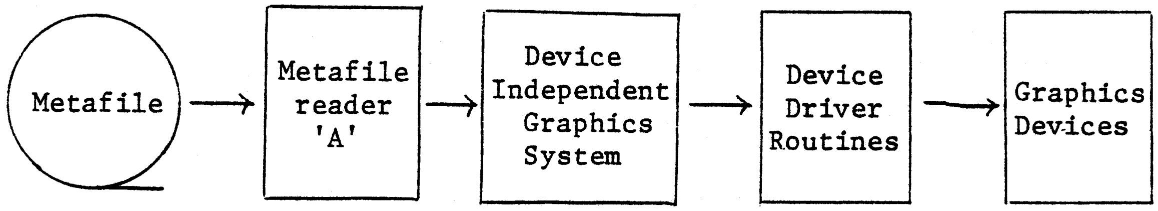

Figure 13 Indirect metafile interpretation

At the present stage of implementing "Oa" GKS in this project, the graphics package only includes a virtual device driver (metafile generator) for writing graphical information on the metafile as depicted in Figure 12. Then, a metafile reader, developed on another computing facility, PDP 11/23, can be used to interpret the metafile, to call routines in a device driver, and to generate pictures on the graphics device as depicted in Figure 14.

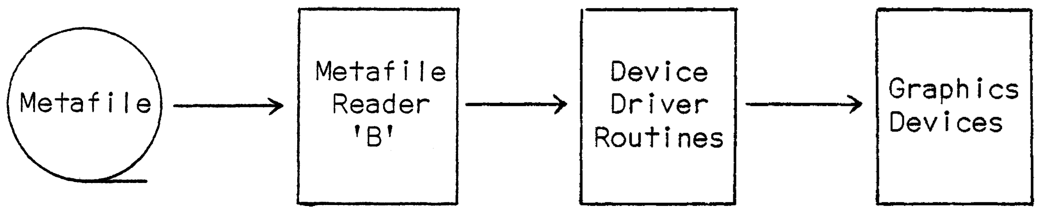

Figure 14 direct metafile interpretation

The metafile format employed in this implementation stems from a working document of the ANSI X3H33 Virtual Device Interface Task Group, published in December 1982. It is not the final draft proposal and is subject to change. A detailed description of the metafile format is the subject of next chapter.

5.4 Other Decisions

The number of simultaneously open workstations and the number of normalization transformations are constants that have to be chosen for each GKS implementation, regardless of level. Choosing the minimum in either case can result in a smaller, simpler implementation.

Supporting only one open workstation at a time can affect an implementation in many ways. The loops that check whether a specified workstation is open or active degenerate to a single if test. Likewise, the loops that send commands to each active workstation are no longer needed. Arrays of open and active workstations become single variables.

Despite all these advantages of a single workstation implementation, this project includes multiple simultaneous workstation capabilities by setting the entry "maximum number of simultaneously open workstations" in GKS Description Table to 2. The purpose of this is to accommodate future expansion more easily.

GKS requires a minimum of two normalization transformations, one of which is the unity transformation and can't be changed. By supporting this minimum, only a single settable transformation needs to be saved. Explaining to the programmer becomes easier, too. He has the simple choice of using NDC (transformation 0), or setting up his own arbitrary world coordinates (transformation 1).

Currently, this GKS minimum requirement of two normalization transformations (0 and 1) is employed in this project, but we can easily add more later if that requirement should arise.

CHAPTER 6

GRAPHICS METAFILE

A graphics metafile is a device-independent representation of a picture intended for subsequent display on a graphics output device. Two important concepts are contained in this definition. First, the metafile is a device-independent representation of a picture that can be displayed on a wide variety of graphics devices. Second, the metafile is intended for subsequent display; thus, it is passive in nature.

The specification of the format and content of a metafile is not part of GKS. At present, an ISO metafile standard does not exist. ANSI, however, has been working on this area with the encouragement of ISO's WG2 and this could lead to another international standard. In 1980, the ANSI X3H3 Computer Graphics Technical Committee formed the X3H33 Virtual Device Interface Task Group to standardize a computer graphics metafile. This task group published a draft proposal in December 1982 [ANS82], on which the metafile generator of this GKS implementation was based.

6.1 Metafile Elements

In order to provide for the description, storage, and communication of graphical information in a device-independent manner, this X3H33 proposal defines the syntax and semantics of a set of elements that may occur in a metafile. These elements are:

- Descriptor Elements: describe the functional content, default conditions, identification, and characteristics of a metafile;
- Control Elements: control initialization, termination, definition of address space, picture initialization, and format descriptions of the metafile elements;
- Graphical Elements: describe images in a metafile;
- Attribute Elements: describe the appearance of a graphical element;
- Escape Elements: used to construct a picture, but not otherwise standardized;
- External Elements: communicate information not directly related to the generation of a graphical image.

A metafile is a collection of elements from this standardized set and must be interpreted or translated in order to present its pictorial content on a graphics device.

6.2 Character Coded Graphics

The encoding scheme for the metafile used in this proposal is called character coded graphics. In the 7-bit character coded method of describing alphanumeric characters and pictorial information, particular character codes are identified by an 8-bit code sequence in which seven of the bits are used as an index into a 128-character code table and the eighth bit is used as parity or for extension to another code table of

128 characters.

The character code table is normally represented as a table of eight columns and sixteen rows with b7, b6, and b5 addressing the columns and bits b4, b3, b2, and b1 addressing the rows. This code table is also called in-use table and is structured into 32-code position C-set and 94- or 96-code position G-set as shown in Figure 15.

6.3 Code Extension

In most applications, there are not enough characters available in the in-use table, so code extension techniques are needed to permit C- or G-sets to be switched, and thus providing a virtual address space larger than the 128-code positions available in a 7-bit environment.

There are four G-sets and two C-sets that are designated at any one time; that is, any one of the four sets G0, G1, G2, or G3 could be invoked into the in-use table by an invocation sequence. In the default state, G0 contains the primary character code set, G1 contains the Metafile code set, G2 contains the supplementary character set, G3 is reserved for future standardization. Furthermore, in the 7-bit environment, the default state has G1 as the current in-use G-set. The C0 set is always in-use since it contains the code extension control codes.

Each incoming bit combination is either decoded according to the current contents of this table or is used to change the content of this table. The in-use table contains, in columns 0 and 1, the C0 set. Three characters of this set, ESCAPE (ESC or 1/11, that is, column 1,

				b7	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1
b4	b3	b2	b1		0	1	2	3	4	5	6	7
0	0	0	0	0								
0	0	0	1	1								
0	0	1	0	2								
0	0	1	1	3								
0	1	0	0	4								
0	1	0	1	5								
0	1	1	0	6								
0	1	1	1	7	C0					G		
1	0	0	0	8	SET					SET		
1	0	0	1	9								
1	0	1	0	10								
1	0	1	1	11								
1	1	0	0	12								
1	1	0	1	13								
1	1	1	0	14								
1	1	1	1	15								

Figure 15 7-bit in-use table

row 11), SHIFT IN (SI or 0/15), and SHIFT OUT (SO or 0/14), are used to control the contents of the remaining six columns of the in-use table.

The SI character is used to invoke the current G0 set into the in-use table where it remains until further control action is taken (that is, it is invoked in a locking manner). The SO character is used to invoke the current G1 set into the in-use table in a locking manner.

The sequences, ESC 6/14 and ESC 6/15, are used to invoke the G2 set and G3 set, respectively, into the in-use table in a locking manner.

A single additional control set, the C1 set, is defined. In a 7-bit environment, it is never invoked into the in-use table in a locking manner. Rather, single characters from the C1 set are accessed via two-character escape sequences and are treated as a single control character. These sequences take the form, ESC Fe, where Fe represents the desired character from the C1 set. This character, by definition, must have a bit combination corresponding to column 4 or 5 of the 7-bit in-use table. The in-use table automatically reverts to its former state after the C1 command is executed and is thus not changed by these two character escape sequences.

6.4 Encoding of the Metafile Elements

6.4.1 The Descriptor and Control Elements

The Metafile descriptor and control elements are encoded in the C1 character set. If a C1 control function is represented by a 2-character escape sequence in a 7-bit code, this combination of the final character is specified by taking A=4 and B=5 in the following table.

A/0	Reserved	B/0	VDM escape
A/1	Begin metafile	B/1	Reserved
A/2	End metafile	B/2	Reserved
A/3	Begin picture	B/3	Reserved

A/4	End picture	B/4	Reserved
A/5	Message	B/5	Reserved
A/6	Clear view surface	B/6	Reserved
A/7	Clip on	B/7	Reserved
A/8	Clip off	B/8	Reserved
A/9	Begin VDM Elements/Defaults	B/9	Reserved
A/10	End VDM Elements/Defaults	B/10	Reserved
A/11	Reserved	B/11	MCSI
A/12	Reserved	B/12	ST
A/13	Reserved	B/13	VDM description
A/14	SS2	B/14	VDM version
A/15	SS3	B/15	Application data

The elements VDM VERSION, VDM DESCRIPTION, APPLICATION DATA, MESSAGE, and ESCAPE consist of a control string that may occur in the data stream as a logical entity for control purposes. The control string consists of an opening delimiter and a command string, and is terminated by STRING TERMINATOR (ST). The opening delimiter indicates which control element is being specified. The command string consists of a sequence of bit combinations in the range of 0/8 through 0/13 and 2/0 through 7/14.

Begin VDM Elements/Defaults initiates a list of parameters, the value of each equaling one of the opcodes used in the Metafile. Default values immediately follow the appropriate opcode values. The parameter list is terminated with the bit combination representing End VDM Elements/Defaults.

Control functions that include numeric parameters are encoded using the Metafile Control Sequence Introducer (MCSI). Multibyte control functions are represented by control sequences. A control sequence consists of the coded representation of MCSI followed by one or more bit combinations that identify the control function and represent the parameters of the control function. The format of a control sequence is:

MCSI Pl...Pn I F

where:

- MCSI is represented by ESC 5/11 in a 7-bit code;
- Pl...Pn are bit combinations representing one or more parameters to complete the control function specification. The parameters are either real or integer, or enumerated type. Each parameter sub-string is separated from other parameter sub-strings by the 3/11 character. The parameter string for all three cases consists of the ASCII characters representing the decimal digits 0 to 9. The whole portion of a real number is separated from the fractional portion by the 2/14 character.

  If the parameter string starts with the bit combination 3/11, an empty parameter sub-string is assumed preceding the separator; if the parameter string terminates with 3/11, an empty parameter sub-string is assumed following the separator; if the parameter string contains successive bit combinations 3/11, empty parameter sub-strings are assumed between the separators. An empty parameter sub-string or a parameter sub-string that consists of bit combination 3/0 only, represents a default value that depends on the control function;

- I is a bit combination 2/1, which, togther with the final bit combination F, identify the control function;
- F is a bit combination chosen from Columns 4, 5, 6, or 7 which terminates the control sequence.

The following table describes the allocation of final bit combinations of elements that use MCSI with 2/1 as a single intermediate.

Element	Final Bit Combination
VDC DIMENSIONALITY	4/1
VDC TYPE	4/2
VDC EXTENT	4/3
CLIP RECTANGLE	4/4
TEXT ASPECT SOURCE FLAGS	4/5
POLYLINE ASPECT SOURCE FLAGS	4/6
POLYMARKER ASPECT SOURCE FLAGS	4/7
FILL AREA ASPECT SOURCE FLAGS	4/8
INTEGER PRECISION	4/9
REAL PRECISION	4/10
COLOR PRECISION	4/11
COLOR INDEX PRECISION	4/12
INDEX PRECISION	4/13
ENUMERATED PRECISION	4/14
COORDINATE PRECISION FOR INTEGERS	4/15
COORDINATE PRECISION FOR REALS	4/0

POLYLINE BUNDLE INDEX	5/0
POLYMARKER BUNDLE INDEX	5/1
FILL AREA BUNDLE INDEX	5/2
TEXT BUNDLE INDEX	5/3

The following table specifies how the parameters of the commands using MCSI should be interpreted.

Element	Parameter Interpretation
VDC DIMENSIONALITY	P1=0 if 2D; P1=1 if 3D
VDC TYPE	P1=0 if integer; 1 if real
VDC EXTENT	P1=coordinate data type; P2=coordinate data type
CLIP RECTANGLE	P1=coordinate data type; P2=coordinate data type
All ASPECT SOURCE FLAGS	P1=0 if individual; P1=1 if bundled
INTEGER PRECISION	P1=number of bits
REAL PRECISION	P1=number of bits for integer portion; P2=number of bits for fractional portion
COLOR PRECISION	P1=number of bits for the red, green, and blue components
COLOR INDEX PRECISION	P1=number of bits for color

	index data type
INDEX PRECISION	Pl=number of bits for index data type
ENUMERATED PRECISION	Pl=number of bits for enumerated data types
COORDINATE PRECISION FOR INTEGERS	Pl=number of bits for VDC
COORDINATE PRECISION FOR REALS	Pl=number of bits for integer portion; P2=number of bits for fractional portion
All BUNDLE INDEXes	Pl=bundle index

6.4.2 Graphical and Attribute Elements

A Metafile graphical or attribute element is encoded in the Gl set and is comprised of an opcode followed by one or more bytes of numeric data. If bit 7 is 0, an opcode is indicated. If bit 7 is 1, numeric data is defined.

Coordinate operand can be either real or integer. There are two precision elements associated with it. Coordinate operands are used in conjunction with graphical elements. The format, when VDC type is integer, is shown in Figure 16. The operands are interpreted as signed two's complement numbers.

The color index, index, and enumeration operands are all interpreted as unsigned integers composed of the sequence of concatenated

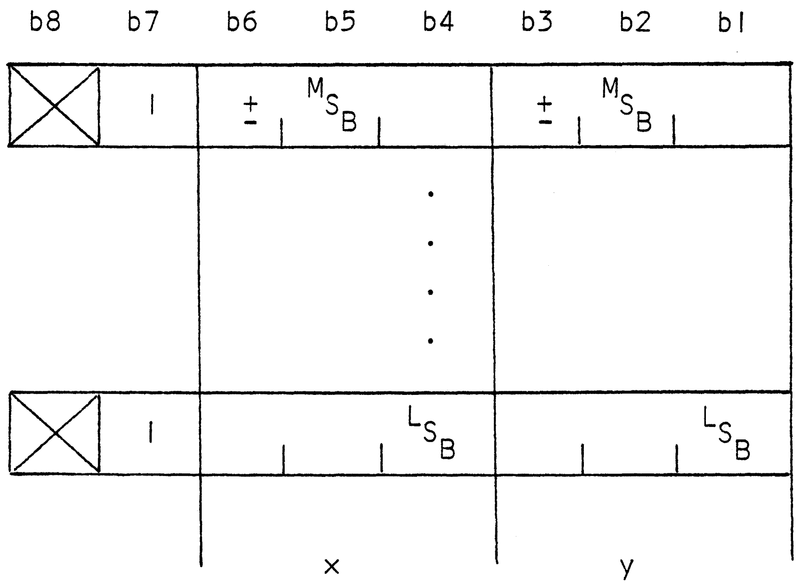

Figure 16 <u>The format for integer coordinate operands</u>

bits taken consecutively (high order bits to low order bits) from the numeric data bytes.

Integer operands are interpreted as signed, two's complement numbers. Real operands are interpreted as signed, two's complement numbers when the binary coordinate is determined from the COORDINATE PRECISION FOR REALS element.

The precision for VDC operands is determined from the COORDINATE PRECISION FOR REALS or COORDINATE PRECISION FOR INTEGERS element. The format for the VDC operand (VDC type is integer) is that of the integer operand. The format for the VDC operand (VDC type is real) is that of the real operand.

The following table describes the default G1 set.

Element	Code Position	Operand Type
POLYMARKER	2/0	nC
MARKER SIZE	2/1	VDC
MARKER TYPE	2/2	E
MARKER COLOR	2/3	CI or 3R
POLYLINE	2/4	nC
LINE WIDTH	2/5	VDC
LINE TYPE	2/6	E
LINE COLOR	2/7	CI or 3R
ARC	2/8	3C
PIXELS	2/9	4C,2I,mnCI
CIRCLE	2/10	C,VDC
ARC CLOSE	2/11	3C,E
POLYGON	2/12	nC
INTERIOR STYLE	2/13	C,2E
HATCH INDEX	2/14	IX
PATTERN INDEX	2/15	IX
PATTERN TABLE	3/0	IX,I,mnIX,mn3R
PATTERN SIZE	3/1	2VDC
FILL COLOR	3/2	CI or 3R
TEXT	3/3	C,S
CHARACTER HEIGHT	3/4	VDC
CHARACTER EXPANSION FACTOR	3/5	R

CHARACTER PATH	3/6	E
CHARACTER UP VECTOR	3/7	2VDC
CHARACTER SPACING	3/8	R
TEXT COLOR	3/9	CI or 3R
TEXT FONT INDEX	3/10	IX
TEXT ALIGNMENT	3/11	2E,2R
TEXT PRECISION	3/12	E
COLOR TABLE	3/13	IX,n3R
BACKGROUND COLOR	3/14	CI or 3R
Reserved for future standardization	3/15	

The data types used in the table have the following meanings:

data types		Meaning
C	Coordinate	Coordinate pair or triple in VDC space.
CI	Color index	Pointer into a table of color values.
E	Enumerated	Set of standardized values. The set is defined by enumerating the identifiers that denote the values.
I	Integer	Number with integer portion.
ID	Identifier	String or integer.
IX	Index	Pointer into a table of values other than color values.
R	Real	Number with integer and fractional portion, only one of which need exist.

S	String	Sequence of characters.
VDC	VDC values	Single real or integer values (as determined by VDC type) in VDC space.

Bit specification of the enumerated data values is as follows:

1. INTERIOR STYLE

b6b5 - interior style	b1 - perimeter visibility
b6b5 = 00 hollow	b1 = 0 invisible
= 01 solid	= 1 visible
= 10 hatch	
= 11 pattern	

2. TEXT ALIGNMENT

b6b5 - horizontal alignment	b3b2b1 - vertical alignment
b6b5 = 00 left	b3b2b1 = 000 top
= 01 center	= 001 cap
= 10 right	= 010 half
= 11 continuous horizontal	= 011 base
	= 100 bottom
	= 101 continuous vertical

3. TEXT PRECISION

b6b5 = 00 string
= 01 character
= 10 stroke

4. CHARACTER PATH

b6b5 = 00 right
= 01 left

= 10 up

= 11 down

5. MARKER TYPE

b6b5b4b3	= 0000	dot
	= 0001	plus
	= 0010	asterisk
	= 0011	circle
	= 0100	x

6. LINE TYPE

b6b5b4	= 000	solid
	= 001	dashed
	= 010	dotted
	= 011	dashed-dotted

CHAPTER 7

CONCLUSIONS

The primary objective of this project was to develop a graphics subroutine package, based on the capabilities described in the GKS specification, that could be used by an application programmer to produce a metafile for storing graphical information and for off-line, passive plotting.

Because GKS is by design device-independent and because a metafile is a device-independent representation of a picture, any application programmer who understands the conceptual model underlying GKS can use this GKSLIB library to support a wide range of 2-D passive graphics applications with little knowledge of the capabilities and characteristics of the physical graphics devices, to which the stored graphical information will eventually be sent.

The device independence is achieved in GKS through the concept of workstation. The current implementation of the GKSLIB library fully supports this fundamental concept and two other important features, two-stage transformation and two-stage attribute handling. Because of this, it should be much easier to add a device driver to this library and to perform on-line plotting in the future.

GKSLIB currently contains about 85% of functions provided in "0a" GKS. Obviously, it would be desirable to include precisely all of the functional capabilities defined in level 0a, and thus to complete a valid implementation of GKS.

APPENDIX I

CONCEPTUAL DIFFERENCES BETWEEN GKS AND THE GSPC-CORE

As mentioned in earlier chapter, whereas GSPC-Core is a proposal for a 3D graphical core system, designed by ACM-SIGGRAPH's GSPC in US, GKS is a proposal for a 2D graphical system and is designed by the sub-committee "Computer Graphics" of German standards-making body. GSPC-Core is much richer than GKS in functionality. Two other important conceptual differences between these two proposals lie in the transformation processing and attribute handling.

In GSPC-Core, the transformation from WC to DC is also performed by a two-stage process. But, it embodies only a single normalization transformation, and the application must laboriously re-create the correct normalization transformation for the part of the display it wishes to modify (Figure 17).

There are almost no graphical applications in which the only pictures generated consist of a single view of a single object occupying the whole view surface. Thus, pictures will really be generated using several "world" coordinate systems. Further, a single transformation system places the burden of retransforming locator input coordinates from NDC space to the space used by the application entirely on the application. These requirements led GKS to include multiple normalization transformations (Figure 18).

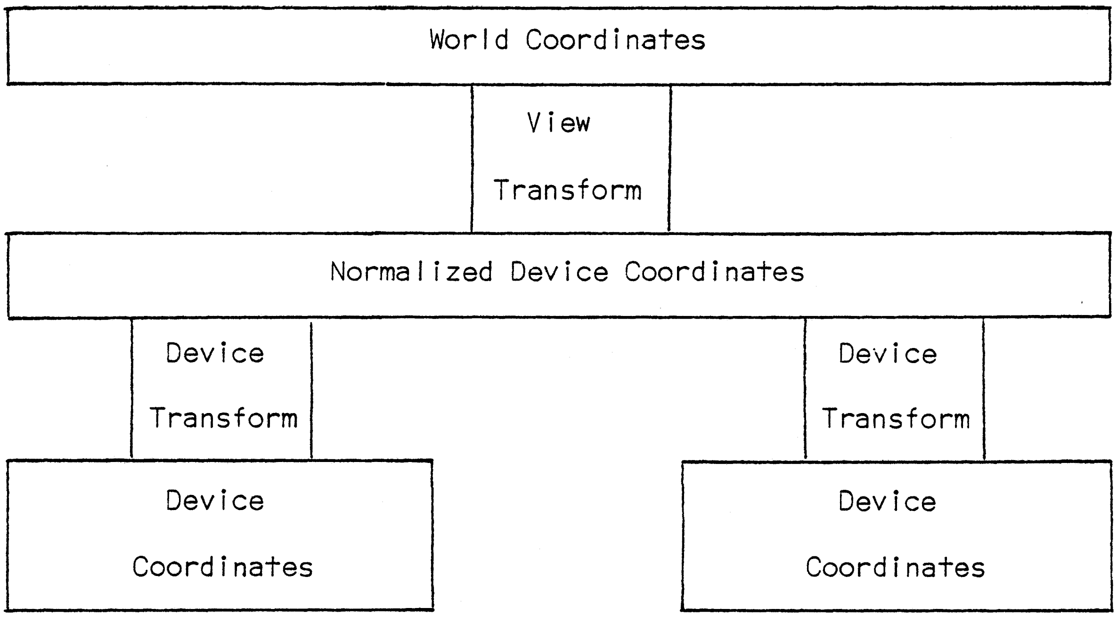

Figure 17 <u>CORE Transformation</u>

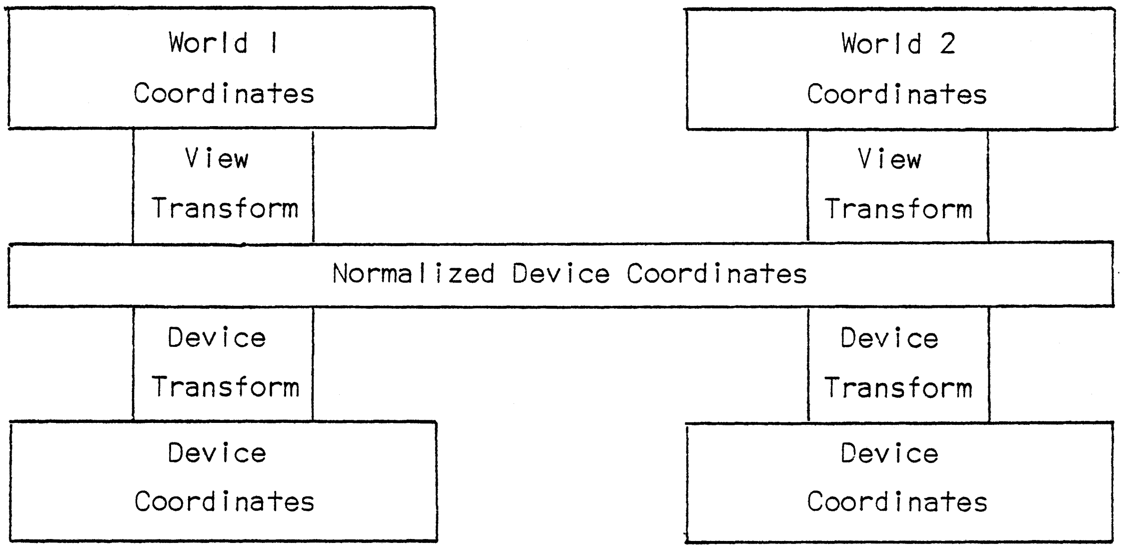

Figure 18 <u>GKS Transformation</u>

In GSPC-Core, the appearance of a primitive is controlled by a set of modal attributes and is associated with the primitive itself. In GKS, the appearance of a primitive is defined by two-stages (in a similar way to the transformation). In the first stage a symbolic attribute is associated with the primitive, while in the second stage the symbolic attribute is mapped on the capabilities of the workstation, thus determining the usual appearance on the graphical terminal.

APPENDIX II

0a LEVEL GKS FUNCTIONS

* not implemented in GKSLIB yet

FORTRAN names	Functions
Control Functions	
OPNGKS	OPEN GKS
CLSGKS	CLOSE GKS
OPENW	OPEN WORKSTATION
CLOSEW	CLOSE WORKSTATION
ACTWS	ACTIVATE WORKSTATION
DEAWS	DEACTIVATE WORKSTATION
CLEAR	CLEAR WORKSTATION
*	UPDATE WORKSTATION
*	ESCAPE
Output Functions	
PLLINE	POLYLINE
PLMARK	POLYMARKER
TEXT	TEXT
FAREA	FILL AREA
*	CELL ARRAY

* GENERALIZED DRAWING PRIMITIVE (GDP)

Output Attributes

SETPLX	SET POLYLINE INDEX
SETLNT	SET LINETYPE
SETLNW	SET LINEWIDTH SCALE FACTOR
SETPLC	SET POLYLINE COLOUR INDEX
SETPMX	SET POLYMARKER INDEX
SETMKT	SET MARKER TYPE
SETMKS	SET MARKER SIZE SCALE FACTOR
SETPMC	SET POLYMARKER COLOUR INDEX
SETTXX	SET TEXT INDEX
SETTFP	SET TEXT FONT AND PRECISION
SETCHE	SET CHARACTER EXPANSION FACTOR
SETCHS	SET CHARACTER SPACING
SETTXC	SET TEXT COLOUR INDEX
SETCHH	SET CHARACTER HEIGHT
SETCHU	SET CHARACTER UP VECTOR
SETTXP	SET TEXT PATH
SETTXA	SET TEXT ALIGNMENT
SETFAX	SET FILL AREA INDEX
SETFAI	SET FILL AREA INTERIOR STYLE
SETFAS	SET FILL AREA STYLE INDEX
SETFAC	SET FILE AREA COLOUR INDEX
SETPTS	SET PATTERN SIZE
SETPTR	SET PATTERN REFERENCE POINT

SETASF	SET ASPECT SOURCE FLAGS
SETCLR	SET COLOUR REPRESENTATION

Transformation Functions

SETW	SET WINDOW
SETV	SET VIEWPORT
SELNT	SELECT NORMALIZATION TRANSFORMATION
SETCLI	SET CLIPPING INDICATOR
SETWW	SET WORKSTATION WINDOW
SETWV	SET WORKSTATION VIEWPORT

Metafile Functions

* WRITE ITEM TO GKSM
* GET ITEM TYPE FROM GKSM
* READ ITEM FROM GKSM
* INTERPRET ITEM

Inquiry Functions

IQOPST	INQUIRE OPERATING STATE VALUE
IQLVL	INQUIRE LEVEL OF GKS
IQWLST	INQUIRE LIST OF AVAILABLE WORKSTATION TYPES
IQMNT	INQUIRE MAXIMUN NORMALIZATION TRANSFORMATION NUMBER
IQWKOP	INQUIRE SET OF OPEN WORKSTATION
IQPATT	INQUIRE CURRENT PRIMITIVE ATTRIBUTE VALUES
IQATT	INQUIRE CURRENT INDIVIDUAL ATTRIBUTE VALUES
IQNTNO	INQUIRE CURRENT NORMALIZATION TRANSFORMATION NUMBER

IQNTLT	INQUIRE LIST OF NORMALIZATION TRANSFORMATION NUMBERS
IQNT	INQUIRE NORMALIZATION TRANSFORMATION
IQCLIP	INQUIRE CLIPPING INDICATOR
IQCNTY	INQUIRE WORKSTATION CONNECTION AND TYPE
IQWKST	INQUIRE WORKSTATION STATE
IQDFUP	INQUIRE WORKSTATION DEFERRAL AND UPDATE STATES
*	INQUIRE TEXT EXTENT
IQCLST	INQUIRE LIST OF COLOUR INDICES
IQCLR	INQUIRE COLOUR REPRESENTATION
IQWT	INQUIRE WORKSTATION TRANSFORMATION
IQCATE	INQUIRE WORKSTATION CATEGORY
IQCLAS	INQUIRE WORKSTATION CLASSIFICATION
IQDSIZ	INQUIRE MAXIMUM DISPLAY SURFACE SIZE
IQPLFC	INQUIRE POLYLINE FACILITIES
IQPPL	INQUIRE PREDEFINED POLYLINE REPRESENTATION
IQPMFC	INQUIRE POLYMARKER FACILITIES
IQPPM	INQUIRE PREDEFINED POLYMARKER REPRESENTATION
IQTXFC	INQUIRE TEXT FACILITIES
IQPTX	INQUIRE PREDEFINED TEXT REPRESENTATION
IQFAFC	INQUIRE FILL AREA FACILITIES
IQPFA	INQUIRE PREDEFINED FILL AREA REPRESENTATION
IQPTFC	INQUIRE PATTERN FACILITIES
IQPPT	INQUIRE PREDEFINED PATTERN REPRESENTATION
IQCLFC	INQUIRE COLOUR FACILITIES
IQPCLR	INQUIRE PREDEFINED COLOUR REPRESENTATION
IQGLST	INQUIRE LIST OF AVAILABLE GENERALIZED DRAWING PRIMITIVES

IQGDP		INQUIRE GENERALIZED DRAWING PRIMITIVE
	*	INQUIRE PIXEL ARRAY DIMENSIONS
	*	INQUIRE PIXEL ARRAY
	*	INQUIRE PIXEL

Error Handling

	*	EMERGENCY CLOSE GKS
ERRORH		ERROR HANDLING
ERRORL		ERROR LOGGING

APPENDIX III

0a LEVEL GKS DATA STRUCTURES

FORTRAN names	Data structure

Operating State

OPSTAT	operating state value

GKS Description Table

GLEVEL	level of GKS
WKNUM	number of available workstation types
WKLIST	list of available workstation types
OPNNUM	maximum number of simultaneously open workstations
ACTNUM	maximum number of simultaneously active workstations
MNTNUM	maximum normalization transformation number

GKS State List

OPNWKS	set of open workstations
ACTWKS	set of active workstations
PLNIND	current polyline index
LNTYPE	current linetype
LNWIDT	current linewidth scale factor
LCOLOR	current polyline colour index
LTASF	current linetype ASF

LWASF	current linewidth scale factor ASF
LCASF	current polyline colour index ASF
PMKIND	current polymarker index
MKTYPE	current marker type
MKSIZE	current marker size scale factor
MCOLOR	current polymarker colour index
MTASF	current marker type ASF
MSASF	current marker size scale factor ASF
MCASF	current polymarker colour index ASF
TXTIND	current text index
TFONT	current text font
TPRECI	current text precision
CEXPAN	current character expansion factor
CSPACE	current character spacing
TCOLOR	current text colour index
FPASF	current text font and precision ASF
CEASF	current character expansion factot ASF
CSASF	current character spacing ASF
TCASF	current text colour index ASF
CHHIGH	current character height
CHARUP	current character up vector
TPATH	current text path
TALIGN	current text alignment (horizontal and vertical)
FAIND	current fill area index
FINTER	current fill area interior style
FSTYLE	current fill area style index

FCOLOR	current fill area colour index
FIASF	current fill area interior style ASF
FSASF	current fill area style index ASF
FCASF	current fill area colour index ASF
PTSIZE	current pattern size
PTPNT	current pattern reference point
CNTNUM	current normalization transformation number
NTNUMS	list of normalization transformation numbers
NTLIST	list of normalization transformations (window & viewport)
CLIPIN	clipping indicator

Workstation Description Table

WKTYPE	workstation type
WKCATE	workstation category

Workstation State List

WKID	workstation identifier
CONNID	connection identifier
WTYPE	workstation type
WSTATE	workstation state
DFMODE	deferral mode
IRMODE	implicit regeneration mode
DEMPTY	display surface empty
NFRAME	new frame action necessary at update
DPLNUM	number of polyline bundle table entries

table of defined polyline bundles

DPLINX	polyline index
DLTTBL	linetype
DLWTBL	linewidth scale factor
DLCTBL	polyline colour index
DPMNUM	number of polymarker bundle table entries

table of defined polymarker bundles

DPMINX	polymarker index
DMTTBL	marker type
DMSTBL	marker size scale factor
DMCTBL	polymarker colour index
DTXNUM	number of text bundle table entries

table of defined text bundles

DTXINX	text index
DFNTBL	text font
DPRTBL	text precision
DCETBL	character expansion factor
DCSTBL	character spacing
DTCTBL	text colour index
DFANUM	number of fill area bundle table entries

table of defined fill area bundles

DFAINX	fill area index
DFITBL	fill area interior style

DFSTBL	fill area style index
DFCTBL	fill area colour index
DPTNUM	number of pattern table entries

table of pattern representations

DPTINX	pattern index
DPARRD	pattern array dimensions
DPARR	pattern array
DCLNUM	number of colour table entries

table of colour representations

DCLINX	colour index
DCLTBL	colour (red, green, blue intensities)
WTUPDT	workstation transformation update state
RWWIND	requested workstation window
CWWIND	current workstation window
RWVIEW	requested workstation viewport
CWVIEW	current workstation viewport

GKS Error State List

ERSTAT	error state
ERFILE	error file

APPENDIX IV

0a LEVEL GKS ERROR LIST

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
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

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
29	Specified workstation is active
30	Specified workstation is not active
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

Transformations

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

Output Attributes

60	Polyline index is invalid
62	Linetype is less than or equal to zero
64	Polymarker index is invalid
66	Marker type is less than or equal to zero
68	Text index is invalid
70	Text font is less than or equal to zero

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
78	Style (pattern or hatch) index is less than or equal to zero
81	Pattern size value is not positive
84	Dimensions of colour array are invalid
85	Colour index is less than zero
86	Colour index is invalid
88	Colour is outside range [0,1]

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

Metafile

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

Escape

180 Specified function is not supported

181 Contents of escape data record are invalid

REFERENCES

[ANS82] "Draft Proposed American National Standard for the Virtual Device Metafile", X3H33 82-15 R5, X3H3 82-33 R5, (December 1982).

[BON82] Bono P.R. _et al_, "GKS - The First Graphics Standard", IEEE Computer Graphics & Applications, (July 1982), pp. 9-23.

[ENC80] Encarnacao J. _et al_, "The Workstation Concept of GKS and the Resulting Conceptual Differences to the GSPC Core System", Computer Graphics, Proc. Siggraph 80, Vol. 14, No. 3, (1980), pp. 226-230.

[ENC81] Encarnacao J., "Graphical Kernel System", IBM SYST J, Vol. 20, No. 4, (1981), pp. 438-440.

[END83] Enderle G., "Core Systems", Computers & Graphics, Vol. 7, No. 1, (1983), pp. 87-90.

[FOL82] Foley J.D. and van Dam A., _Fundamentals of Interactive Computer Graphics_, Addison Wesley, (1982).

[GSP79] "Status Report of the Graphics Standards Planning Committee", published as Computer Graphics, Vol. 13, No. 3, (August 1979).

[ISO82] ISO. "Graphical Kernel System (GKS) - Functional Description", Draft International Standard ISO/DIS 7942, (November 1982).

[NEW78] Newman W.M. and van Dam A., "Recent Efforts Towards Graphics

Standardization", Computing Surveys, Vol. 10, No. 4, (December 1978), pp. 365-380.

[NEW79] Newman W.M. and Sproull R.F., Principles of Interactive Computer Graphics, 2nd Edition, McGraw-Hill, (1979).

[REE82] Reed T.N., "A Metafile for Efficient Sequential and Random Display of Graphics", Computer Graphics, Vol. 16, No. 3, (July 1982), pp. 39-43.

[ROS82] Rosenthal D.S.H., "GKS in C", Eurographics 82, Greenaway D.S. and Warman E.A. (eds), North-Holland, (1982), pp. 359-369.

[SIM83] Simons R.W., "Minimal GKS", Computer Graphics, Vol. 17, No. 3, (July 1983), pp. 183-189.

[STR83] Straayer D.H., "Adapting Applications to the Graphical Kernel System", Computer Design, (July 1983), pp. 167-172.

[SUT82] Sutcliffe D.C., "Attribute Handling in GKS", Eurographics 82, Greenaway D.S. and Warman E.A. (eds), North-Holland, (1982), pp. 103-110.

[TEN82] ten Hagen P.J.W., "Graphics Standardization", Computer Graphics, Vol. 16, No. 3, (July 1982), pp. 45-46.

[WAG80] Wagener J.L., FORTRAN 77 Principles of Programming, Wiley, (1980).