# GPGS

# A Device-independent General Purpose Graphic System for Stand-alone and Satellite Graphics

L.C. Caruthers
J. van den Bos

Informatica/Ccmputer Graphics Group
Faculty of Science
University of Nijmegen
Nijmegen
The Netherlands

A. van Dam

Program in Computer Science
Brown University
Providence, Rhode Island
U.S.A.

## ABSTRACT

GPGS is a subroutine package offering powerful and versatile support for passive and interactive vector graphics, for time-sharing, batch, and stand-alone minicomputer systems. The package is computer, language, and operating system, as well as display device independent. Its key purpose is to allow for transportabilit of programs and programmers by providing easy to learn, high level features. The applications programmer writes his program once and then executes it on any supported graphics equipment without recompiling or relinking it. Device-independence was implemented by dividing GPGS into a device-independent part invoked by the applications programmer, and internal, "device drivers", one per display device. Like the GSPC "Core System" whose design it influenced, GPGS is a general purpose package. It has a subset of graphics facilities to handle output of line and character primitives with attributes such as line style and character size, and input from interaction tools such as lightpens, keyboards, valuators, and function keys. It also supports 2D and 3D viewin transformations for clipping and window to viewport mapping, and coordinate transformations.

Unlike the GSPC Core System, GPGS also includes a set of basic features for modelling objects which allows definition of device independent masters called seudo picture segment. These are distinguished from normal, device (DPU) dependent pictur segments into which primitives and their attribute-value settings are ordinarily compiled. These masters may be instanced subject to affine transformations (translate, rotate, and scale) to create a typical master-instance hierarchy. The hierarchy may be stored in a disk based library or compiled into a normal picture segment for output to a display device.

The images of objects stored in device dependent picture segments may be transformed on the display surface by v port (image) transformations. These typically allow use of hardware transformation capabilities for dragging or tumbling object images.

Host/satellite graphics is accommodated by having the device independent part of GPGS in the host and splitting the device drivers across host and satellite. At the source code level it therefore makes no difference on which.configuration a program will be executed.

Among the existing implementations are versions written in assembler for the IB 360/370 and the PDP 11, in both stand-alone and satellite mode, and under a variety of operating systems. They support plotters, storage tubes, and high performance refresh displays. FORTRAN based implementations exist for the Univac 1108, the PDP 10, and a Harris minicomputer.

Keywords and Phrases: interactive graphics, device independent graphics, graphics subroutine package, satellite graphics

CR Categories: 8.2, 4.29

## 1 INTRODUCTION

GPGS offers high level graphics support easily accessible to high level language programs. The subroutine call mechanism has been employed in preference to new language primitives as the easiest extension mechanism. Thus any (mini) computer with FORTRAN is a possible candidate for a GPGS implementation. GPGS interfaces to the operating system and handles all communications and data conversion problems for passive and interactive physical devices. The resulting environment and device independent graphics application programs may be transported without change (given identical FORTRAN's).

The design started as a joint effort of the Universities of Nijmegen and Delft, with consulting provided by Cambridge University. It was meant to supersede such device dependent packages as IBM's GSP [1] for 2250's and Calcomp's well known plotting subroutines. At Cambridge and several other locations in the U.K., experience with machine and device independent graphics had already proved successful, with the Cambridge GINO-3 [2] system. Rather than reimplement GINO for new hardware being acquired by all three Universities, it was decided to provide more extensive facilities and improve GINO's design.

Parallel implementations were begun in 1971 on an IBM 370 with a PDP-11/45 graphics satellite at Nijmegen and on a PDP-11/45 in standalone mode at Delft. In 1974-75 the graphics group at the computing center at the University of Trondheim in Norway, made an ANSI FORTRAN implementation for the Univac 1108 of a large subset of GPGS. Their GPGS-F is based on the Delft PDP implementation. Additional versions of GPGS exist in countries as far apart as Germany and India (the latter on a PDP 10); the official version is now being licensed at nominal cost. Altogether the system runs in production in several dozen installations.

## 2 GLOBAL DESIGN CONSIDERATIONS

### 2.1 THE SUBROUTINE PACKAGE AND AN OVERVIEW OF ITS FACILITIES

A primary GPGS design decision was to create a subroutine package instead of a new graphics language or graphics extensions to an existing language. A subroutine package is easier to design and implement than language extensions, simpler for programmers to learn, and easily extended by adding more subroutines. The ease of implementation also allowed for more efficient assembler language implementations on different computer systems. The obvious disadvantage of a subroutine package is its limited, awkward syntax.

The subroutines included in GPGS were chosen to be just far enough removed from the hardware to provide device independence and still allow the applications programmer to control the hardware of an advanced CRT display reasonably efficiently. An additional guideline for choosing which features to include in GPGS was to make the package general purpose and rich [3, 4]. At the same time, the design would be modular, to minimize the cost of learning and using a limited subset of the full system.

The features included are those required or generally useful for implementing a powerful graphics "Core System" [5]. Unlike the designers of the GSPC Standard, the GPGS designers felt a basic modelling component to be generally useful as well; both subsystems are overviewed in Sections 4 and 5.

The graphics subsystem includes line and text output primitives and their attributes (line style, intensity, character size, etc.), 2D and 3D windowing and clipping, and perspective and axonometric projections. The modelling subsystem allows specification of an object as a hierarchy of previously defined picture segments, with inclusion of picture segments controlled by placement (instancing) transformations (translation, rotation, and scale) and a transformation stack. The hierarchy is always compiled to a single device dependent picture segment for output purposes.

To aid viewing and/or modelling, (clipped) images of objects may be post-transformed on the display surface with hardware facilities for dragging or tumbling, using viewport transformations.

Provisions for more specialized facilities such as hidden-line removal, data structure support beyond n-level segment hierarchy, and animation were not included. Note that GPGS is a rich, stand-alone package with over a hundred subroutines, which may have special purpose packages built on top. GPGS-F, for example, supports a high level plotting package [6].

When designing the subroutines themselves, the key concept was simplicity. The mnemonic name of a subroutine indicates what its function is, and what type of entity it operates on; for example, SELDEV(I) means select device I whereas SELLIB(I) means select library I. Each subroutine has as few arguments as possible. GPGS supplies reasonable default values to allow the unsophisticated user to have to learn only a few calls.

## 2.2 OUTPUT DEVICE INDEPENDENCE

To avoid having to write and load a complete device dependent package for each output device on which a picture is to be displayed, GPGS as a device independent package has a common, shared device independent part and as many device dependent device drivers (to be loaded at execution time) as there are devices for that implementation (see figure 1). To the applications programmer this means portability in that he can write his graphics program once and use it with different graphics devices without changing the source code or relinking his program.

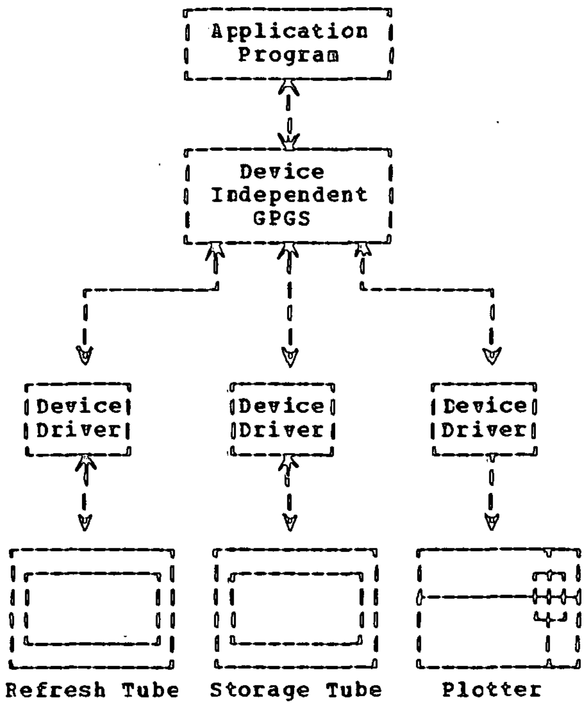

Fig. 1 GPGS System diagram

The applications programmer defines objects in user coordinates as picture segments composed of primitives (picture elements) These include lines and text and previously defined sub-objects pseudo picture segments). He then specifies views of these objects (images) to be plotted on an idealized output device. In reality, the device independent part of GPGS uses the setting of the viewing transformation parameters (including type of projection if 3D, window, clipping mode, viewport) to compile a (clipped) device independent image of the object. This intermediate form is then further mapped by a specific device driver onto a viewport of an actual output surface. These include plotters, microfilm recorders, storage tube displays and steered beam CRT refresh displays. By specifying viewports in "normalized device coordinates" which are fractions of actual physical dimensions of the display surfaces, the applications programmer need not be concerned about real device dimensions and can draw device independent pictures in user coordinates.

## 2.3 INPUT DEVICE INDEPENDENCE

The idealized device concept is also appropriate for describing the GPGS scheme for handling input from the console operator. Unlike most other high level packages which are primarily oriented towards plotters and storage tubes with limited input capability, GPGS was designed to handle powerful refreshed CRT's with a wide variety of input tools such as lightpens, joysticks, tablets, and keyboards. Upward compatibility from simple displays to powerful ones is achieved automatically since GPGS handles the superset of facilities of the most powerful commercially available vector displays. Downward compatibility from more complex to simpler graphics devices is provided through simulation of higher level (hardware) facilities carried out by device drivers. For example, a lightpen may be simulated with cursor crosshairs, a dial with a keyboarded value, etc. In this way, an applications program may still run essentially unchanged on a simple device, but with an altered (probably slower) operator interaction.

The GPGS idealized device includes all the basic input devices proposed by Wallace [7]. It supports the following single input tools: refresh clock, alphanumeric keyboard, lightpen for picking, and audible alarm. It also has at least one tool in each of the following classes of tools: function switches, dials (1 dimension), tracking cross and data tablet (2 dimensions), and joystick (3 dimensions). Each single tool or class of tools (including the simulated ones) returns appropriate information to the applications program in a specific format; function switches return their identifiers, dials return fractions between 0 and 1, the lightpen returns a name stack, etc.

## 3 RESOURCE MANAGEMENT

As part of solving the problem of interfacing graphics programs to the operating **environment**, GPGS performs applications **program** requested manipulations of the following graphics resources: graphics devices with their associated drivers, picture buffers, and picture libraries. The **manipulations** include initialize, select, clear, release, and status inquiry. Output buffers and devices **may** be sequentially selected as the current one; multiple input tools **may** be simultaneously enabled.

Pictures may be stored off-line in picture libraries which are controlled by the applications programmer much in the same way as the device and buffer resources. Indeed libraries can be thought **of** as extensions of buffers. Thus one can build a picture on disk rather than using a core buffer, and subsequently either send the picture directly from the library to a storage tube, or overlay a piece of the applications program to create a refresh buffer. Libraries are therefore particularly useful for making large pictures on small computers, or saving standard menus or standard drawing symbols (picture parts).

To allow the applications program to find the properties and status of its currently allocated resources, and to retrieve previously established attribute settings, GPGS has inquiry facilities for returning execution environment information to the applications program. Inquiry can be used by an optionally specified applications program subroutine which receives control on the occurrence of an error condition.

## 4 OUTPUT FACILITIES

### 4.1 PICTURE ELEMENTS, PICTURE SEGMENTS, AND THEIR ATTRIBUTES

Using GPGS, an applications programmer specifies an object as a **collection** of output primitives called **picture elements**, defined by manipulating an idealized drawing stylus in a 2D or 3D user coordinate system. Among the picture elements are individual lines and polyline sequences, character **strings**, and markers (special characters used for point plotting). **Picture element attributes** like line style and width, character size and spacing, color, intensity, or blinking are used to modify the output characteristics of these basic picture elements.

All picture elements and their attribute-value specifications are collected in one or more named picture **segment(s)**. Individual elements in the picture segment **may** not be altered after the segment has been defined. The segment as a whole is the GPGS unit of manipulation, for purposes of deleting or extending its set of elements as a logical unit. To alter the contents of a segment the programmer has to regenerate it.

The only other manipulation of a picture segment allowed is to change its associated segment attributes (visibility, lightpen sensitivity (pickability), and viewport transformations). These attributes are global for the picture segment and may be changed any time after the creation of the picture segment has begun and before it has been deleted.

Objects specified to GPGS as picture segments are compiled by device independent GPGS and the appropriate device driver into display device dependent code. This display file (for refreshed displays) will typically be a chain of buffers, each linearly segmented into picture segments. Each segment contains a header where the segment attributes are stored for subsequent modification.

### 4.2 VIEWING TRANSFORMATIONS

To produce one or more "snapshots" or views of an object on an output **device(s)**, the applications programmer first sets the proper viewing transformation **conditions**[1] and then defines the object as one or more picture segments. For 3D, he specifies the type of projection to be used (perspective or **axonometric)**, a window and a viewport. The window is a rectangle in 2D, or a parallelepiped or rectangular pyramid in 3D, determining the limits of the user coordinate space in which the object is defined that will be displayed (if clipping is enabled). The **viewport** is the portion of the display surface on which the contents of the window are going to be mapped on. Viewport boundaries are specified device independently in fractional normalized device coordinates.

As explained in Section 2.2, the picture elements in each picture segment are successively passed through the GPGS viewing transformation pipeline where they are (optionally) clipped to the boundaries of the window, and then mapped to the viewport by a device driver which produces actual device dependent coordinates.

### 4.3 IMAGE MANIPULATION WITH VIEWPORT TRANSFORMATIONS

Viewport Transformations are a facility introduced by GPGS primarily for refresh displays to take advantage of hardware facilities for translation (most displays

**[1]As for most GPGS features, suitable defaults are supplied by the system.**

support relative **vectors**), or even 2D or 3D rotation (via hybrid cr digital transformation hardware)

Since typical transformation hardware affects only **DPU primitives**, it can be used only after the entire device independent **viewing** transformation pipeline described above has been applied to produce **DPU** code consisting of clipped picture elements mapped to a **viewport**. The picture segment contains in its header instructions to load transformation registers. The **viewport** transformation changes only these instructions in the header. As an **example**, in the simple case of 2D **translation** (for dragging) using relative coordinates, the picture segment of relative **DPU** primitives **would** have an initial absolute move **in** its header. It should be noted that even for **DPU's** lacking transformation hardware, viewport transformations are a limited but efficient facility because they affect a clipped image of an **object**, with typically many fewer picture elements than the original object **had**. **Thus** the console operator can select a piece of his object using the viewing transformation pipeline **once**, and thereafter manipulate it on the screen with **viewport** transformations, rather than with the more expensive viewing transformation pipeline.

## 4.4 A BASIC MODELLING SYSTEM

If the user has a hierarchical object data **structure**, he **may** mirror this application oriented hierarchy in device independent GPGS definitions of pseudo **picture segments**. These may be inserted as many times as desired in a normal picture segment. **Furthermore**, they may be arbitrarily **nested**, ith a classical master/instance reference scheme, including the ability to use translation, **rotation**, and scale transformations to properly place a subpicture instance in a higher level **one**. High level facilities for stacking, saving, and restoring 4 **x 4** homogeneous coordinate transformations exist, as **well** as for pre- and post-oultiplying for matrix composition and matrix vector **multiplication**. Pseudo picture segments, ouch like **macros**, **may** be copied directly (subject to the instancing transformation) as they are specified inside the higher level (pseudo) picture **segment**, or this inclusion **may** be postponed until an entire hierarchy is built up and then is instanced in a real picture **segment**. Thus an entire device independent hierarchy, with all (pointer) references and transformations **may** be stored as a standard symbol in a library.

## 4.5 SINGLE LEVEL SEGMENTATION, MULTI LEVEL NAMING AND CORRELATION

While GPGS allots the definition of a hierarchy of objects as a tree of pseudo picture segments, they must ultimately be compiled to a linearly segmented display **file**. In **order** to allow mapping (correlation) from picture elements on the display surface to the original application data structure from which they were derived, **n-level** naming for picking/correlation is supported. within named picture segments, picture elements may be given unique names as part of their **specification**, and may additionally be grouped with another unique name using **BGNNAM**, **ENDNA "brackets"**. These group names may be nested to reflect the original hierarchy, and will be returned as a name hierarchy (stack) by the correlation mechanism. Note that GPGS only supports a single level of device dependent picture segmentation for manipulation purposes, but at least allows hierarchical naming **[8]**. If the hierarchy must be preserved for manipulation of individual subobjects, each should be compiled to its **own** picture segment, to be individually highlighted, deleted, or **viewport** transformed.

## 5 INTERACTION FACILITIES

The approach to provide the programmer with hardware or simulated interaction tools such as lightpens, keyboards, joysticks and function keys **was** explained in Section 2.3. To support these **tools**, the device driver will sample or be interrupted by each tool to see if it has been used by the operator. It does this independent of **(possibly** asynchronous to) the device independent part of GPGS and the applications program. To communicate this asynchronous activity **(i.e.**, to simulate tool interrupts) to the applications program GPGS maintains a **FIFO** interrupt queue. The device driver fills the queue with event reports, which the application program can interrogate using the **INWAIT** function, at its **convenience**.

The applications program calls the **INWAIT** function with a list of identifiers of the tools that it wishes to accept information **from**. GPGS then looks at the interrupt queue to see if the console operator has used any of the specified tools. If he has, the tool identifier and the event report for that tool is returned to the applications **program**.

The interrupt queue can be polled for any past tool activity or the applications program can go into a wait state to be **"interrupted"** by tool **activity**, as a function of the time parameter. If the time parameter is positive, GPGS will return to the applications program either when the time expires or the console user uses a requested **tool**. If the time parameter is zero, information is returned from a tool only if the console user had used the tool prior to the call to **INWAIT**, otherwise GPGS returns immediately without providing any tccl information. If the time parameter is negative, **INWAIT** returns to the applications program only after the console user uses one of the tools in the list. Though not all **implementations** currently support **it,** **INWAIT** is designed to wait for **information** frcm more than one display at the sane time.

The **queueing** discipline used by GPGS is to allow each tool of each initialized device to make at most one entry at a time in the interrupt queue common to all GPGS devices. Thus the first interrupt from a tool stays in the queue until it is either passed to the applications **program** or flushed by **INWAIT** because it was not requested by the applications program. The entire interrupt queue **may** also be cleared (flushed) by the applications program.

when **INWAIT** returns to the applications program it gives tack the information from only one tool. **To** allow the applications program to sample tcol values without using the more expensive **INWAIT** interrupt queue mechanism, GPGS has the **REATOL** subroutine which has a tool identifier as parameter and returns information in the same format as **INWAIT.**

## 6 SATELLI PL EENTATION QUESTIONS

Standard implementations of GPGS were visualized either fcr a host **(multi-programmed** or dedicated) suffi- cientIy powerful to run the entire applications **program** and its environment, or for a host/satellite system where the application would run on the host and the satellite would need to be only sufficientlv powerful tc implement an **"intelligent"** terminal capable of supporting local graphics housekeeping. **No genuine "cooperative distributed** processing" between co-egual host and satellite proessors was **visualized**.[2]

The satellite processor in GPGS therefore only holds the part of a device driver that deals directly with the physical device **(e.g., data ccnversion,** tool **sampling,** interrupt handling, and viewport **transformaticns).** The communication between the host and the satellite is in the **fcrm** of messages between two parts of a driver. The primary consideration in designing the satellite support **was** to get the best response tine **pcssible** for the console user. **Typically,** the determining factor on response time is the opportunity for **executicn** (dispatching) of the applications program on the multi-programmed host processor. with this in mind it is clear that the response time (and link traffic) is optimized by minimizinq the total number of messages **between the** host and the **satellite,** i.e., having as small a number of large messages as possible.

Each picture segment that is created must be sent to the satellite for display. To minimize the number of messages it is better to send the whole picture segment after it has been closed rather than sending each picture element in a separate message to the satellite. The situation for interaction is less advantageous than for sending picture segments because each **INWAIT** and REATOL request must be sent to the satellite as a separate message, and the reply from the satellite is, of course, another message. Each message requires operating system intervention for I/O, and potential loss of execution control, with the need for subsequent redispatching. Interaction is therefore likely to be slow on a busy host.

Another question in designing satellite support is why the division between host and satellite code was put inside the driver. This leaves the whole work of the device independent picture processing pipeline to be done by the host processor. Although it would be possible to put some part of the pipeline (say, clipping) on he satellite, it would probably violate the goal of minimizing message traffic. For example, if a small window would result in very few lines to be displayed, it would be very inefficient to send the entire (set of) segment(s) to the satellite for local clipping. Splitting the pipeline between any of the other stages of processing simply adds additional overhead **to the total processing by gathering the** half-processed picture elements into a message on the host which must be split apart again for further processing on the satellite. Given our assumptions about the limited power of the satellite, the best strategy is thus to create complete device dependent (clipped) picture segments on the host and send them to the satellite for display and local manipulation (e.g., changes in segment attributes or **viewport** transformations).

## 7 CONCLUSIONS

### 7.1 DEVICE INDEPENDENT VERSIONS

Transportability of highly interactive programs through device independence has been achieved with **GPGS.** Plotter programs can make the sane picture on a storage tube and on a refresh CRT, with only minor variation due to character **font,** etc.

---

**[2]The full power of a PDP 11/45 is therefore not utilized in satellite mode; the 11/45 runs stand-alone GPGS except for the largest applications better done on the 370 mainframe.**

Programs which use all the interaction tools of a 3D Vector General display can be debugged (with somewhat painful simulation) on a simple storage tube with only cursor crosshairs and a keyboard. But in order to allow this, the plotter or storage tube programs have been forced to abide by the same picture creation and manipulation rules as a refresh CRT program.

In our experience in writing device drivers we have seen that a driver for a simple output device like a plotter or line printer is very easy to write, while the driver for an interactive device, though much more work, is certainly simpler than creating a whole new package and conversion interfaces for other devices. Simulating interaction tools and viewport transformations for devices lacking adequate hardware is the hardest job. Making a new driver is usually a matter of modifying the lowest level of some existing driver.

Full assembly language implementations exist for PDP 11 (under DOS, RSX, RT-11, and Unix operating systems) and for IBM 360/370 - PDP 11 (RT-11 and Unix) satellite systems. The GPGS-F ANSI FORTRAN subset implementation supports Tektronix 4010-4015 and Kingmatic plotters [9]. Currently the following graphics devices are supported by the assembly language versions:

stand-alone:

**IBM: Tektronix 4010/4012/4014/4015 (batch and TSO),**
**Calcomp plotter**
**PDP: Vector General,**
**Tektronix 4010/4012/4014/4015,**
**Tektronix plotter,**
**DEC line printer,**
**Versatec plotter printer**

**satellite system:**

Vector General on PDP 11

GPGS was designed without general data structure support and as a highly modular package in which sophisticated features such as input tool simulation, viewport transformations, and device independent picture hierarchies are optional code segments, loaded only when needed. This has resulted in basic packages which take little memory space and run quickly. With a basic driver (no tool simulation) for the Tektronix 4014(4015), the RT-11 version of device independent GPGS and device driver takes a total of only 5.5k words of PDP 11 storage, less than the RT-11 batch system nucleus. A 370 system with PDP 11 satellite and full 3D Vector General display takes 32k bytes for device independent GPGS, the 370 portion of the device driver, and a picture segment buffer. As far as performance is concerned, for highly interactive (not much computation) applications programs on the IBM implementation, the CPU utilization and response time are comparable to that of text editing programs.

## 7.2 SUITABILITY FOR APPLICATIONS

The following comments pertain to how "general purpose" GPGS has proven to be, that is, how easy it is to write applications programs. For computer-aided design programs, where a fairly low level interface is needed along with multiple devices (interactive CRT and plotter), GPGS has proven to be very effective. GPGS has the primitives needed for making static data plots but it does not have any utilities to draw graphs. Therefore, a set of graphing routines to go on top of GPGS has been designed [6].

Applications that have proven to be unreasonable to attempt with GPGS have had to do with a picture which must be structurally changed in real time in response to console operator input. Due to the requirement that a picture segment must be completely rebuilt each time it is changed, even if the building of the next version of the picture is overlapped with the displaying of the previous version, it is difficult to achieve real time changes with anything but the simplest of pictures. Where a device has transformation hardware, however, a program accessing this hardware through GPGS can produce real time motion of arbitrarily complex picture parts.

## 7.3 USING GPGS FOR SATELLITE GRAPHICS

From our experience to date we can conclude that for satellite use with GPGS, a fairly simple satellite processor is sufficient. A PDP-11/10 is probably sufficient for all satellite work required by the current design. The amount of memory required by the satellite is determined by the amount of picture that can be refreshed flicker-free by the graphics device. A very high speed communications link (600K baud) is certainly nice for GPGS because it means that picture segment (message) length really isn't much of a factor in the response time. Even with a slower communications link, say 9600 baud, GPGS satellite support would still be useable. Because all decision making must be done by the applications program as to how to respond to the interactive tools, the limiting factors for the response of GPGS satellite graphics are the speed with which the applications program can be dispatched on the multiprogrammed host computer, and the speed of message transmission.

## 7.4 SUMMARY

GPGS has largely achieved its original design goals of being a device independent, easy to use subroutine package to allcw program portability. GPGS provides applications programs with access to multiple, diverse graphics devices through the same subroutine calls. The GPGS design has been shcwn to be implementable on small and large computers alike, with an implementation effort required from one to four or five man-years as a function of the language coded in, operating system support, number and sophistication of device drivers, etc. The FORTRAN implementation took approximately one man-year for device independent GPGS and simple device drivers.

GPGS is the only generally available graphics system that prcvides high level support for the whole range of devices from plotters to high perfcrmance vector displays, with device independence for both output and input facilities. Because of this, GPGS has had a significant influence on the design of the GSPC Core System.

ACKNOWLEDGEMENTS

Many people have contributed their efforts to the design, implementation, and critique of GPGS. We list them here with their present and/or former affiliations: Ed Anson, Ian Bergercn, Jan van den Bos, Steve Carmcdy, Larry Caruthers, Marty Michel, Martin Mueller, John Patberg, Dave Rice, Julie Schwartz of Nijmegen University; Andy van Dam of Nijmegen and Brown Universities (U.S.A.); Dik Groot, Lex Hensels, Edwin Hermans, Kees Thijs, Guus van der Wal, Peter Veenman of Delft University; Charles Lang and Peter Wcodsford of Cambridge University; Dick Newell, Martin Newell, and Tom Sancha of the Cambridge Computer Aided Design Center. The helpful reviews of this paper by Ingrid Carlbom, Dan Bergeron, and Jim Foley are also much appreciated.

REFERENCES

1. IBM - Graphics Subroutine Package (GSP) for FORTRAN IV, COBOL, and PL/I; form GC27-6932.
2. Woodsford, P.A., The Design and Implementation of the GINO 3-D Graphics Software Package, Software - Practice and Experience, Vol. 1 (October 1971), p. 335.
3. Caruthers, L.C., and van Dam, A., GPGS User's Tutorial, Informatica, Faculty of Science, University of Nijmegen, The Netherlands, October 1975.
4. Groot, D., Hermans, E., Caruthers, L.C., and Schwartz, J., GPGS Reference Manual, Rekencentrum, T.H. Delft and Informatica, Faculty of Science, University of Nijmegen, The Netherlands, May 1977.
5. SIGGRAPH GSPC, First Report on Graphics Standards, Proceedings of SIGGRAPH 77, San Jose, July 1977.
6. Skaland, M., Zachrisen, M., High-Level Graph-Plotting Routines for GPGS-F, Preliminary Specifications, RUNIT Report, University of Trondheim, Norway, 1976.
7. Wallace, V.L., The Semantics of Graphic Input Devices, Proceedings ACM Symposium on Graphic Languages, 26-27 April 1976, Miami Beach, Florida, pp. 61-65.
8. Foley, J.D., Picture Naming and Modification: An Overview, Proceedings ACM Symposium on Graphics Languages, 26-27 April 1976, Miami Beach, Florida, pp. 49-53.
9. GPGS-F User's Guide, RUNIT Computer Centre, University of Trondheim, Norway, September, 1975.