C /

A HIGH-LEVEL GRAPHICS LANGUAGE BASED ON THE GRAPHICAL KERNEL SYSTEM

by

HANQIU SUN

B.A.Sc., Huazhong University, 1981

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Electrical Engineering

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 1986

78

Department of Electrical Engineering

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date March 20, 1986

Abstract

Being an application area of programming languages, graphics languages should keep pace with the development of today's programming languages. Data types, structural operations and free layout of statements provide a more effective means of picture generation, i.e., modelling, rendering and viewing. The Graphical Kernel System (GKS), an international standard graphics language since 1984, is specified on a subroutine basis, therefore suffering from the lack of such high-level language features. This thesis investigates and implements the FORTRAN language binding of GKS into a high-level programming language (HL/GKS) by a generated precompiler. The weaknesses and restrictions of GKS and its FORTRAN binding are discussed. The advanced features and functions of HL/GKS are addressed. The graphical syntax and semantics rules of the extended portion of HL/GKS are introduced. It is expected that HL/GKS will have more attractive features and effective productivity for GKS applications compared to the procedure-level GKS system. The input statements of HL/GKS have the capability of picture communication by interactive devices and hence enable the implementation of sophisticated graphical application programs.

Table of Contents

List of Figures

Acknowledgement

I am most grateful to my parents, members of my family and all my dear friends for their contant encouragement.

I would like to express my sincere appreciation to Dr. G.F. Schrack, the supervisor of this project, for his continued interest, encouragement and guidance during the research work and writing of the thesis. This research would not have been possible were it not for the financial support of the People's Republic of China.

I. INTRODUCTION

As a communications carrier between computers and human beings, a computer language takes the responsibility of conferring a user's intention expressed in the language to the computer, and of reporting related results processed by the computer back to the user. In order to communicate naturally and freely, the implementation of high-level programming languages has been investigated through many years. The language FORTRAN is characterized by its natural arithmetic expressions, the language PASCAL originated from a need to define data structures flexibly, and LISP is based on its attractive features of logical relationships. Basically, all high-level programming languages emphasize the differences of actual and abstract, realistic and virtual models during the communication process.

The interrelationship between two models is shown in Figure 1.

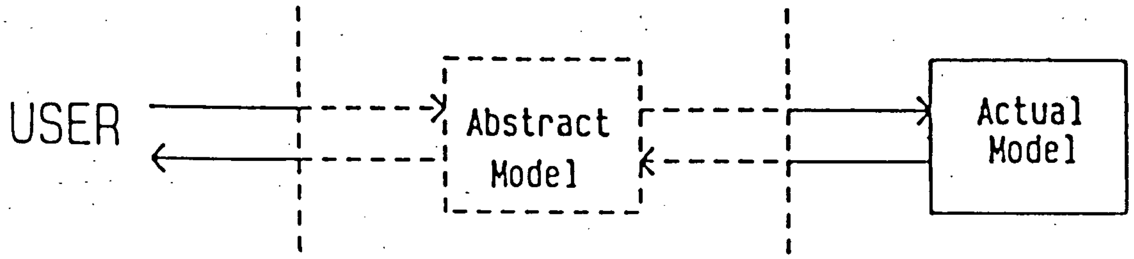


Figure 1 - Two Models in Language Communications

When following the development of programming languages, beginning from machine codes, to assembly languages, and to today's programming languages, high-level languages tend to offer more freedom to users so that a programmer is able to concentrate fully on research problems rather than on system details. To meet this requirement, data types, variables, block structures, and control facilities have formed a necessary part of high-level programming languages. In fact, transparency to the real world and closeness to a natural language became the major characteristics of a high-level programming language.

Graphics languages can be considered as an application area of programming languages. Three aspects of computer graphics need to be dealt with in languages, namely,

. picture modelling,

. picture rendering,

. picture viewing.

A picture is modelled by picture elements, rendered through data processing, and viewed on different graphical devices. In a graphics language, picture modelling and viewing are affected by programming functions and interactive facilities; picture rendering is mainly processed on the basis of a built-in database and data structures. Similar to high-level programming languages, graphical types and hierarchical picture structures should be included in a high-level graphics language.

Regarding the hierarchical structure of graphical objects, two kinds of element references are distinguished:

. single level reference,

. multiple level reference.

Single level reference means that only a single graphical object, composed of a collection of one or more graphics primitives, can be referenced; multiple level reference means that subelements, at various levels of an object structured tree, can be accessed. Superposition and deletion are the two graphical operations to deal with hierarchical structures. A graphical object can be constructed by superposing its subparts level by level and by deleting some of its subparts at certain levels. Other processing functions, such as replacment, zooming, etc., can be simulated with those two functions. Therefore, an evaluation of a graphics language usually focuses on the data structures and basic functions of the language provided.

During the research process of high-level graphics languages, two main approaches have emerged:

. entirely new graphics language,

. graphics extension on a standard language.

A new graphics language offers full freedom in providing graphical tools for programmers. A picture grammar, comprehensive modelling, and versatile facilities are provided via the language definition. However, this approach has not been popularly adopted except for some special cases. The reason is that a more or less extensive compiler needs to be developed and that providing the necessary non-graphical computing support is difficult. The language GLIDE(77)[East77] is designed within this class. An example of its code is demonstrated in Figure 2.

The second class, language extensions, has further diverged into two subbranches:

```
ATTRIB TEXT title, notes;
ATTRIB REAL shelflength, deskft, filespace;
ATTRIB BOOL phone;
FORM   WORKSPACE =
       BFORM  title <-- 'secretary';
              shelflength <-- 15; deskft <-- 5;
              phone <-- true; notes <-- 'none'
       EFORM;
COPY WORKSPACE[10] = {; phone <-- false};
COPY WORKSPACE = {; title<--'chairman';
                    deskft<--10;
                    notes<--'likes red wallpaper'};
```

Figure 2 - A Piece of Code of the Language GLIDE(77)

- subroutine packages,
- extended languages.

Both of these incorporate graphics constructions into a host language but by different approaches. In subroutine packages, some graphical subroutines are simply added to the host's runtime library, thus no additional compilation is required for its implementation. The system is easily extendable and portable to serve different needs and installations. Therefore, subroutine packages are adopted as a common design approach in practice. On the other hand, subroutine packages are not the proper candidates for high–level graphics languages because of the shortage of data types and structures. The functions of picture modelling, rendering and viewing are processed only by subroutine calls via parameter lists. For a large system with many subroutines and even more parameters, programming becomes difficult. Also, most programming errors can only be detected at execution time, thus leading to a loss of its appeal as a language. A comparison between subroutine packages and high–level graphics languages is presented in Figure 3.

Subroutine Packages	High-Level Languages
numerical coordinates	graphical variables
arrays of coordinates	structured graphical variables
input and output calls	interactive statements
functions calls	graphical keywords and symbols
attributes calls	name assignments
controls calls	syntax parsing

Figure 3 - Comparison of Subroutine Packages and High-Level Languages

Consequently, no distinction is made between graphical processing and conventional processing in the form of subroutine calls, resulting in a lack of explicitness of the syntax and the semantics of the language.

Extended languages, in contrast, include graphical data types and operators in their extended portion. Frequently referred-to picture elements can be defined formally as standard data types of a language; commonly used picture operators can be expressed symbolically as reserved words in its syntax. As an example, the graphical data type ellipse in the language MIRA[Magn81] is presented in Figure 4.

```
TYPE ellipse = FIGURE (centre: vector; a,b: real);
  VAR theta, step: real; x1,x2: vector;
  BEGIN  theta := 0.0;
         x1 := <<a,0.0>> + centre; step:= 0.1;
       WHILE theta <= 2.0*pi DO
          BEGIN theta := theta + step;
                x2 := <<a*cos(theta),b*sin(theta)>>
                     + centre;
                CONNECT(x1,x2); x1 := x2
          END
  END;
```

Figure 4 - Data Type Ellipse in the Language MIRA

Another example is the union operator U, appearing in the language GRAPHEX68[Dene75] as:

```
          menu <-- and SCAL 0.25 POS (0.8,0.8)
                   U or SCAL 0.25 POS (0.8,0.6)
```

Obviously, an extended language shows many attractive points due to its graphical types, picture structures, and symbolic expressions. These are implemented at the expense of efforts in precompiler construction or compiler extension. From the analysis above, subroutine packages are appropriate for generative (of vertices and patterns) graphics in picture viewing; and an extended language is more suitable for inferential (relational description) graphics in picture modelling and rendering.

The Graphical Kernel System (GKS)[Ende84], recommended as an international graphics standard since 1984, is specified on the subroutine basis. Obviously, all the shortcomings of subroutine ` packages will influence the utilization and popularization of the GKS system. To further better programming using GKS, this thesis investigates the design and implementation the high-level GKS language (HL/GKS) by a built-in preprocessor. Besides removing the restrictions of subroutine calls, HL/GKS has hidden many details of the GKS specification, simplified the performance of GKS functions and introduced new graphical types and operators while staying compatible with the concepts of GKS.

HL/GKS is implemented as a FORTRAN extension for the FORTRAN language binding of the GKS standard. The built-in preprocessor, which translates HL/GKS programs into GKS subroutine invocations, is synthesized by means of the Top-Down

Compiler Writing System (CWS)[Schr82]. Details of the preprocessor design and generation will be presented in Chapter 4.

Chapter 2 introduces several ancestors of high-level graphics languages in the extended class. Advanced graphics features and individual design styles are explored in some depth.

A brief overview and analysis of GKS and its FORTRAN binding is presented in Chapter 3. The system organization and specification are discussed. Some of the restrictions and weaknesses of GKS and its FORTRAN binding are pointed out.

Chapter 4 describes the design principles and implementation steps of the HL/GKS language. The design objectives, the new graphical types and structures are addressed. The internal data structures and related issues of the preprocessor are illustrated.

Four examples of HL/GKS with each output picture are provided in Chapter 5. Example 1 and Example 2 are written in both HL/GKS and GKS to offer a direct comparison of the two languages for identical picture generation. Example 3 and Example 4 are HL/GKS programs that provide additional programming styles and application instances of the HL/GKS language. More facilitate features and practical implementation of HL/GKS can be inferred from those examples.

II. A SURVEY OF HIGH-LEVEL GRAPHICAL LANGUAGES

Coming from different application areas and practical installation environments, a number of high-level graphical languages have been proposed and implemented in the past with different styles. For example, the language IMAGE[Brie75], designed in 1975, concatenates on an OBJECT/ACTION control structure, the display picture description syntax and the hardware independent handling of input devices. An application program written in IMAGE is functionally constructed by four independent blocks, as follows, (Figure 5).

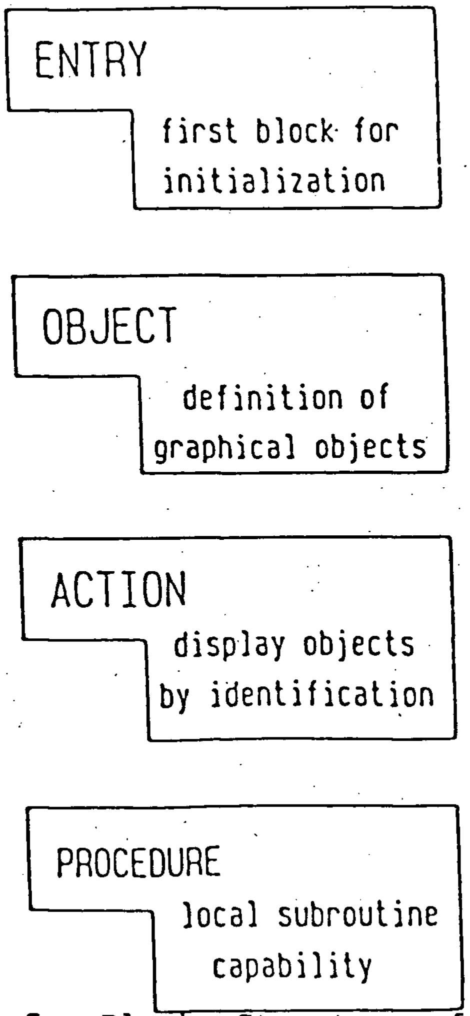

ENTRY
first block for initialization
OBJECT
definition of graphical objects
ACTION
display objects by identification
PROCEDURE
local subroutine capability

Figure 5 - Blocks Structure of the Language IMAGE

Departing from conventional serial programming, the language IMAGE shows up as an action-oriented language.

ESP^3 (Extended SNOBOL Picture Pattern Processor)[Shap75], also designed in 1976, has a different orientation. The language is based on the premise that structural descriptions are an essential part of both picture construction and pattern recognition. Basic picture expressions in ESP^3 appear in statements as illustrated in Figure 6.

```
t = draw('line','start=(1,2); end=(1.5,3);')
t1 = transpic(T,'point(T,"TC") ==> (1.5,1)')
t2 = turnpic(T,'about=point(T,"TC"); deg=180;')
```

Figure 6 - Basic Expressions in the Language ESP^3

Originating from a linguistic approach, ESP^3 has been successfully implemented for the field of pattern recognition with most of its features being of the nature of a high-level graphics language.

GLIDE (Graphical Language for Interactive Design)[East77], proposed in 1977, exhibits some new design characteristics. It attempts to organize commonly-needed database features and operations for the design of physical systems in a high-level computing environment. The hierarchical definition of an graphical object in GLIDE is shown in Figure 7.

Based on a three-dimensional object database, GLIDE provides an effective means of spatial modelling and manipulations for application programs in CAD.

MUMBLE[Guib82], built in 1982, is another graphics language appropriate for describing bitmaps and digital computations for

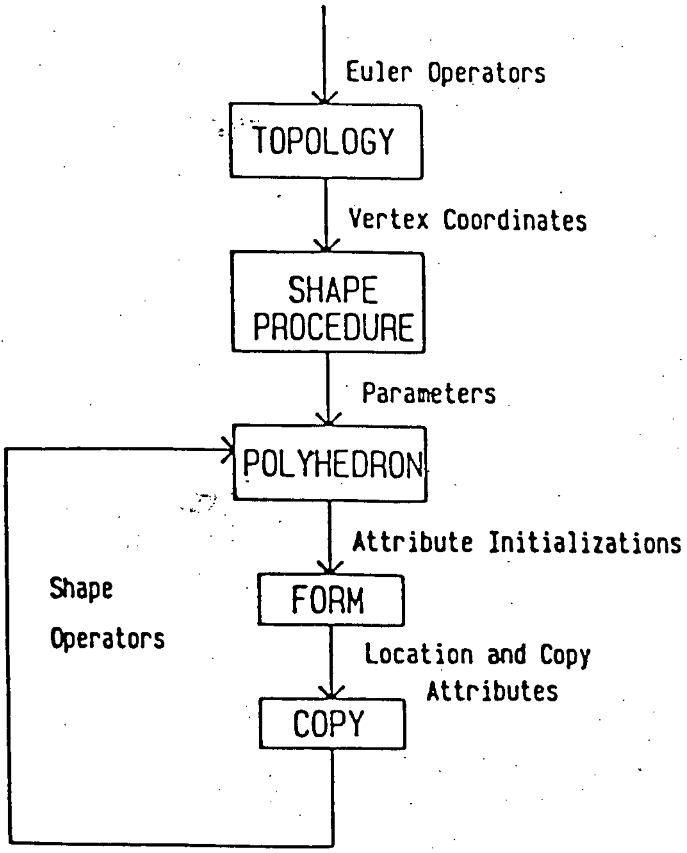


Figure 7 - Hierarchical Definition of the Language GLIDE

raster displays. Boolean operations like AND, OR and XOR are performed for each pixel of the image plane on a bit-by-bit basis. An example of the computations in MUMBLE is shown in Figure 8.

```
A <- B OR (((B SHIFT[0,1]) + (B SHIFT[0,-1])
         + (B SHIFT[1,0]) + (B SHIFT[-1,0]))> ALL 2 )
```

Figure 8 - Computation Expressions in the Language MUMBLE

Moreover, primitives and geometric operations are precisely specified in the language MUMBLE. As raster devices are becoming increasingly important, the interesting and useful features of MUMBLE will become apparent due to its specific design.

From the above brief overview, it becomes obvious that, unlike other uniform programming languages, graphics languages have been created of highly individual character and versatile styles stemming from different motivations, facilities and practical requirements. However, what are the common criteria for every high-level graphics language to follow? What are the basic principles and crucial factors to be kept in languages design? In practice, as mentioned before, the three main research approaches existing in graphics language design are:

. subroutine packages,

. entirely new graphics languages,

. extended languages.

Subroutine packages are easily portable and extendable but structureless; entirely new languages have the freedom of a totally new graphics syntax and semantics but at the cost of an entirely new compiler involved. Therefore, the approach of extended languages has attracted most research attention due to economics considerations. The host language can provide all non-graphic computation and control facilities, thus the cost of a language implementation is restricted to its extended graphics part.

Using the same principles of a high-level programming language design, a graphics language should contain its graphical types, variables, and operators via name reference and hierarchical structuring towards a higher level. Three kinds of structures are generally considered in the design of graphical languages:

. data structures,

. picture structures,

. language structures.

Data structures deal with the internal representation of graphical objects in the data base; picture structures focus on the interrelationships between graphical objects in a picture space; and language structures consider both internal procedures of system functions and external procedures of statement types. In some sense, the picture structures will somehow be reflected by the internal data structures. Data structures and language structures are mainly concerned with the structures of database and language layout.

In the following section, some typical examples of extended languages are investigated in some depth, concentrating on the aspects of graphics syntax, semantics, and the creation of high-level language features such as graphical data types, structures, and picture operations.

2.1 CADEP (1971)

(Computer Assisted Description Patterns)

CADEP[Brac71] is a problem-oriented graphics language extended in FORTRAN for positioning geometric patterns in a two-dimensional space. The graphical types defined in CADEP are:

- geometric,
- graphic,
  - unilevel graphic,
  - multilevel graphic.

A geometric type is employed to build up geometric entities viewed as auxiliary patterns. It can be assigned by geometric primitives and processed by the following operators and functions.

- / intersection,
- + union,
- - difference,
- * concatenation,
- copy(g,s),
- trans(g,x1,y1,s),
- rot(g,x1,y1,alfa,s).

The unilevel graphic type represents physical two-dimensional entities lying on a particular graphical plane. An operation and function used for geometric types can be used as a unilevel type as well. The multilevel graphic type is composed of several unilevel patterns lying on different planes and belonging to the same physical object (for instance, in integrated circuit masks, the same gate lying on different levels). The operations on multilevel types can be performed on each of the unilevel types separately and then link all of them by the concatenation operator *.

In addition, predicate functions to acquire the state and relationship between graphic objects as well as display commands are supplied in CADEP.

Without interactive activities involved, CADEP proves to be a passive language applicable to two-dimensional pattern layout possessing a simple syntax and natural semantics, which lead to a well-defined language structure in CADEP. The introduction of multilevel types provides a clear concept for each level modelling and three-dimensional modelling where the same object can overlap different levels. Being an extended FORTRAN language, CADEP keeps a consistent form in its declaration and assignment statements with those of host language FORTRAN.

2.2 GPL/I (1971)

(PL/I Extension for Computer Graphics)

GPL/I[Smit71] is an interactive graphics language intended to handle graphical situations such as three-dimensional modelling, hardcopying, display, picture storage, colour attributes, and picture animation.

The graphical types in GPL/I are:

. vector,

. image,

. attr,

. text,

. graphic.

The operations on vectors are

. + addition,

. - deletion,

. *.* dot product,

. *** cross product;

on images are

. +> inclusion,

. ||| connection,

. @ position,

. <> scaling,

- *> rotation.

Some operations and attributes, such as perspective projection, intensity, and thickness, are implemented by functions. Graphical types are processed into graphical files with sets of attributes which can be displayed as an image. Functionally, they are classified as

- design files,
- display files,
  - hardcopy files,
  - softcopy files,
- storage files.

In GPL/I, graphical attributes can be assigned directly to an image or indirectly to an ATTR variable for global reference. Interactive programming is achieved by extending the standard PL/I interrupt management statements. An interesting feature of the GPL/I system is its animation function. Both the IMINTR function and the ANIMATE statement can dynamically process images. With all the predefined display procedures, picture viewing in the GPL/I language is well performed.

The use of character sets instead of names for graphical operators does not convey readily their meaning. Also, as most attributes of variables are set by function calls, the GPL/I system requires a larger runtime library than other systems.

2.3 GRAPHEX68 (1975)

(Graphical Language Features in ALGOL 68)

Departing from other languages, GRAPHEX68[Dene75] uses the facilities of ALGOL 68 for defining new graphical data types and operators. Much flexibility is gained thereby which is valuable during the experimental definition and evaluation phase of the system.

The two graphical types in GRAPHEX68 are

. point,

. pict.

Correspondingly, the operators are

. - connection (of points),

. <- assignment,

. pos, rot, scal,

. U union (of pictures),

. extended by or reduced by.

In an application, graphical types and two versions of the operator connection '-' are declared in GRAPHEX68 as shown in Figure 9.

```
mode point = struct( real x,y )
mode string = [1:0 flex] char

op - = ( point p1, point p2 ) ref poly:
         heap poly := ((0,p1),(1,p2));
op - = ( ref poly p0, point q ) ref poly:
(int n= upb p0;
heap[1:n+1 flex] struct ( int beammode,
                          point p) pon;
               pon[1:n]:= p0;
               pon[n+1]:= (1,q);
               pon ) ;
```

Figure 9 - Data Types and Operators of Language GRAPHEX68

Several procedures are constructed for handling various interrupt situations. Being problem oriented, GRAPHEX68 defines its graphical syntax and semantics in a concise and natural way. Graphical variables are declared and processed by graphical operators in a rather obvious manner. Moreover, interactive features in GRAPHEX68 provide a facile approach on picture modelling and viewing. Since no specific specialized compiler activities are involved, GRAPHEX68 is completely portable to any system which has an ALGOL 68 compiler installed.

2.4 MIRA (1981)

(A Graphical Extention of PASCAL)

MIRA[Magn81] is a graphical-oriented language. It provides users the means of defining graphical types much like other types of the host language PASCAL. Nested type definitions allow simulating a hierarchical structure of graphical objects. Graphical variables under each data type are semantically checked to avoid any mismatches.

The two graphical types defined in MIRA are:

. vector,

. figure.

The operations on vector are:

. + addition,

. * scalar product,

. read and write.

The operations on figures are

. connect,

. include,

. create,

. delete,

. draw.

The create statement dynamically creates new figures in memory and the delete statement deletes them. The draw statement permits a created figure to be drawn on a graphical terminal.

Five standard figure types are predefined in the system:

. segment,

. line,

. circle,

. square,

. triangle.

Picture operations such as rotation, scaling and translation are performed by standard procedures in MIRA. With user-defined types and a number of predefined procedures, the syntax and semantics of MIRA appear quite straightforward for a naive user who wishes to learn the language. MIRA emphasises picture modelling, and pays less attention to picture rendering and viewing. The superposition operation is simulated by nested type specification, but the facility of partial deletion is missing. Consequently no updating is possible after an object is built. The graphical data types and operations of MIRA are implemented like a procedure and possess an argument list, an unusual characteristic, and hence somewhat "immaterial".

2.5 LIG (1982)

(Language for Interactive Graphics, Version 6)

LIG[Ross82] is a three-dimensional, interactive, and object-oriented language extension of FORTRAN. In the system, versatile functions and virtual models are provided by its extensive preprocessor and internal database.

The two graphical types defined in LIG are:

- vector,
- graphical.

The operations on vectors are:

- + addition,
- # cross product,
- || magnitude,
- / scalar division,
- - subtraction,
- . dot product,
- * scalar product,
- .EQ. .NE. vector comparison.

The operations on graphical type are:

- + superposition,
- delete statement.

Formally, four standard graphical primitives are provided by the runtime library:

- blank,
- polyline,
- polygon,
- text.

In addition, LIG includes a programmable primitive model definition. Picture transformations are defined similarly to picture attributes. Additional control structures like the case and repeat statements are made available. A camera model is built into LIG to allow viewing three-dimensional objects from different viewpoints. In particular, as the preprocessor of LIG parses all statements, not only graphics statements, full communication between graphical types and non-graphical types is provided.

Being a self-sufficient graphics system, some incompatible language constructs appear in LIG. For example, a reference point is set separately from transformations, the deletion operation is expressed in statement form rather than the symbolical one like the superposition operator. In contrast, the data structure in LIG is well defined and is fully accessible.

2.6 PASCAL/GRAPH (1982)

(Extension of PASCAL)

PASCAL/GRAPH[Bart82] is another language extending PASCAL which offers device-independent programming. A device-independent language allows a programmer to focus his attention on the creation of abstract models, the use of intelligible algorithms, and the clarity of the results.

The data types in PASCAL/GRAPH are:

. picture,

. outunit,

. inunit.

Operations on picture are:

. over,

. conn,

. app,

. encl.

The functions on each graphics device are distinct from each individual call, such as:

. readcoord (digitizer, x, y),

. readpicture (digitizer, p),

. readreal (valuator, x),

. readchar (keyboard, c).

Graphical primitives and operations are declared as standard functions in PASCAL/GRAPH, providing a powerful means for model building.

In PASCAL/GRAPH, the function over corresponds to the superposition operation; the deletion operation is partly realized by the substitute function. The main feature of PASCAL/GRAPH appears to be its machine independence, accomplished by adapting the system to the available graphical hardware using description tables. With hardware devices mapped to logic devices, it is possible to write interactive programs for arbitrary graphical device installations in PASCAL/GRAPH.

As introduced above, language extensions for graphics allow the creation of graphical types and structures, and the processing of graphical objects based on the predefined graphical syntax and semantics. Procedures and statements are the two alternative syntactical constructs for expressing graphics. Statements require more processing effort by the preprocessor but have a natural appearance; procedures increase the size of the runtime library with restricted clarity. Both symbolic operators and identifiers representing graphical operators are adopted in language design as both are suitable for different situations. Interactive facilities are considered to be an important means in picture modelling especially for engineering applications.

In summary, data structures and language structures are the most important research areas for the implementation of high-level graphics languages.

III. THE GRAPHICAL KERNEL SYSTEM

3.1 Overview

The Graphical Kernel System, GKS, has been developed by the International Standards Organization in a long process since 1976 and finally evolved as the internationally recognized standard in computer graphics in 1984.

GKS belongs to the class of subroutine packages. With GKS, a programmer can incorporate graphical tasks within an application program to produce and manipulate pictures. The most important feature of GKS system is its independence from any particular programming language and from any graphical device, which makes GKS totally transportable and independent of hardware updations and configurations.

To realize the idea, the concepts of logic input devices and functional workstations were introduced. There are six logic input devices defined in GKS:

- locator,
- valuator,
- choice,
- pick,
- string,
- stroke.

The concept of workstation ties a flexible virtual system to a variety of real-world devices. Each workstation consists of a single display surface with zero or more input devices.

For instance, a plotter is considered to be the simplest workstation.

The data types of GKS are specified independently of any specific programming language. For instance, the data type enumeration comprises an ordered set of values which can be mapped, e.g., to the scalar type in the language PASCAL or to the integer type in FORTRAN.

Three sets of coordinate spaces are available in GKS to facilitate independent transformations of graphical data. They are

- world coordinate spaces (WC),
- normalized device coordinate space (NDC),
- device coordinate spaces (DC).

WC spaces are the application spaces to be used by a programmer independently of graphical output devices. A DC space is the coordinate space specific to a display device. The NDC space is chosen as the uniform coordinate space of a virtual device, conceived to lie between the WC spaces and DC spaces. To map the data from one coordinate space to another, two types of system transformations are specified:

- normalization transformations (NT),
- workstation transformations (WT).

The relationships of the coordinate spaces via the transformation mappings are illustrated in Figure 10.

Another main concept in the GKS system besides workstations is segments.

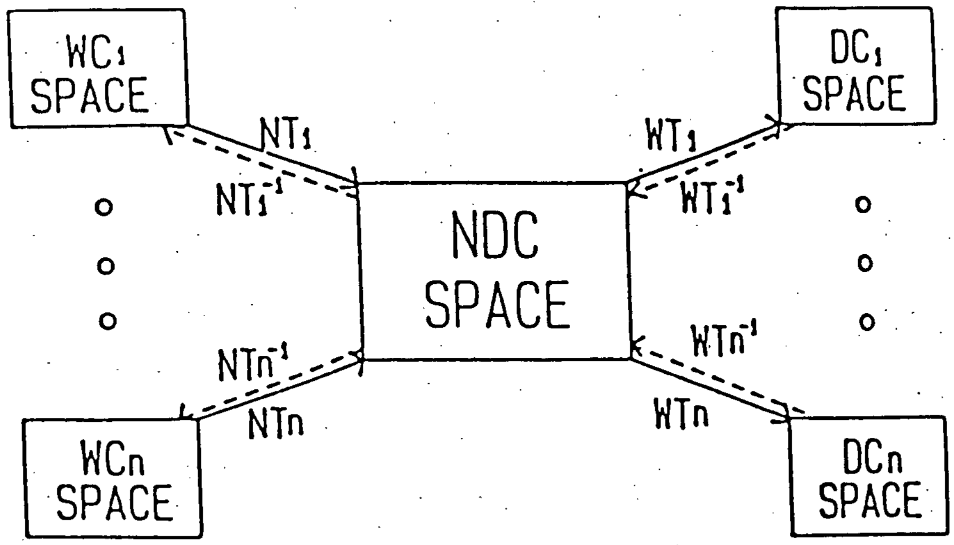


Figure 10 - Relationship of the GKS Spaces

GKS system = segments +
workstations + ...

The workstations correspond to the interface between a GKS program and graphical terminals while segments are the basic objects which are easily referred to for picture processing.

Generally, the syntax of GKS is represented by the following two aspects.

(1) The Protocol of Subroutine Calls

There are 201 subroutines specified in the FORTRAN language binding of GKS. For each subroutine call, the parameter order, parameter data types, and the subroutine name have to observe strict reference rules. For example, to evaluate a transformation matrix in the FORTRAN binding of GKS, the routine needing to be invoked are shown in Figure 11.
Any mismatch in number, order, or data type will cause a failure of the performance.

```
GEVTM(x0,y0,dx,dy,phi,fx,fy,sw,mout)
where  x0,y0(real) --- reference point to the transformations
       dx,dy(real) --- shift vector,
       phi(real)   --- the rotation angle,
       fx,fy(real) --- scale factors for x and y coordinates,
       sw(integer) --- a parameter indicating the coordinate
                       space,
                        . 0 world coordinates,
                        . 1 normalized device coordinates,
       mout(real) --- the name of a 2 by 3 array which will
                      contain the resulting transformation
                      matrix.
```

Figure 11 - The Routine for Evaluating Transformations of GKS

(2) GKS States

There are five system states defined in GKS:

- GKS closed,
- GKS open,
- at least one workstation open,
- at least one workstation active,
- at most one segment open.

The legal transitions of the states is presented in Figure 12. Under each state, only certain subroutine calls are allowed to be issued. For example, the statement "deactivate workstation" is not legal in the state "workstation open"; the command "open GKS" is not acceptable in the state "workstation active". The legal set of subroutine invocations under each state shows the proper syntax order of GKS.

The semantics of GKS are implicitly passed to each subroutine invocation. The subroutine of the primitive POLYLINE(n,px,py) implies that N points are to be connected by line segments as a polyline. The call of the primitive

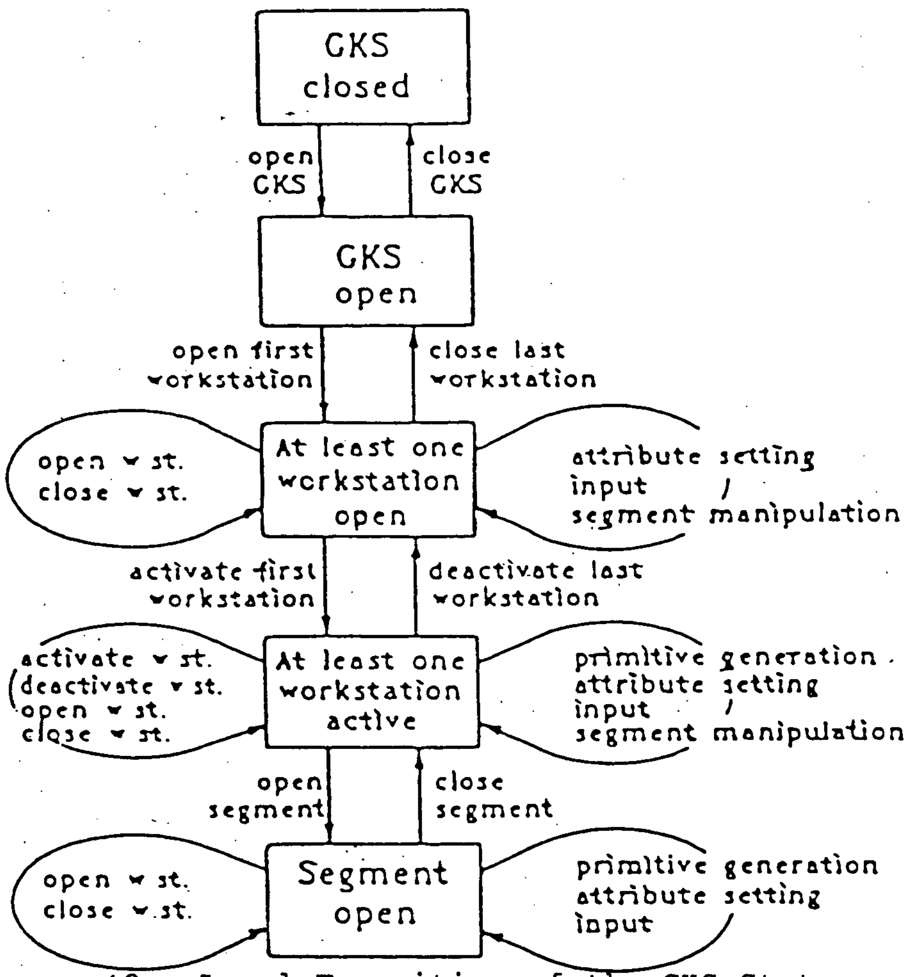


Figure 12 - Legal Transition of the GKS States

TEXT(px,py,chars) confers that a string of characters is output, starting at point (px,py).

3.2 Evaluation of GKS

A review of GKS will reveal some restrictions and inconveniences of the system performance, stemming from its own definition.

3.2.1 Poor Syntax Definition

To use GKS, a programmer needs not only to understand all the details of the system specification but also to take the responsibility of correct invocation of subroutines, and similar details. When can a segment be opened? Which subroutines can be called in the present state? What is the proper subroutine name and which are the proper parameters which need to be referred to? Which workstation is open and active? Taking care of such restrictions undoubtly becomes an unnecessary burden.

3.2.2 Restricted Semantics Performance

In GKS, segments represent picture units, thus segment operations refer to picture operations. The segment functions defined in GKS are given in Figure 13.

create segment
close segment
rename segment
delete segment (from workstation)
associate segment with workstation
copy segment to workstation
insert segment

Figure 13 - The Segment Functions in GKS

The pair "create segment" and "close segment" deals with the state flow of GKS. "Rename segment" reassigns a new segment name to a created segment. "Delete", "associate" and "copy segment" perform the specified functions on display workstations. Among these segment operations, "insert segment" is the only one which simulates the superposition function for picture modelling. As only a single level reference of segments is provided, the deletion operation is not supported by GKS.

The absence of this function may cause trouble when it is required in an application. For example, it is not possible to have a situation like:

house:- house - window(1) + window(2)

For GKS, a programmer needs to rebuild a new segment which contains all the original objects in the segment house with the exception that segment window(1) is replaced by window(2), then delete segment house and rename the new segment as house. Indirect processing leads to a rather confusing and troublesome situation. Even for the superposition operation, done by each insert call, the interrelationship of picture models is rather ambiguous. Consider a picture model:

house :- door + window + roof

The equivalent processing steps in GKS is the sequence of subroutine invocations as shown in Figure 14.

```
CALL GEVTM(0.,0.,0.,0.,0.,          {evaluate trans matrix}
           1.,1.,0,tran1)
CALL GCRSG(4)                       {create segment}
CALL GINSG(1,tran1)                 {insert segment from WISS}
CALL GINSG(2,tran1)
CALL GINSG(3,tran1)
CALL GCLSG                          {close segment}
```

Figure 14 - Superposition Operation in GKS

where segments 1, 2, 3 and 4 represent the objects door, window, roof and house.

The parameter settings in the GKS subroutines are not uniquely specified. As a simple example, the range in the call of cell array appears as:

```
CALL GCA(px,py,qx,qy,dx,dy,dimx,colla)
```

Here, px, py and qx, qy correspond to the lower-left and upper-right hand corners of the cell array, which are arranged in a vertex order. Yet the range in the call of normalization transformation is:

CALL GSWN(tnr,xmin, xmax, ymin, ymax)

in which the range of the window is arranged in an axis order of x and y instead of a vertex order. The irregularity and incompleteness of the subroutine invocation will deter the efficiency and utilization of the system functions.

3.2.3 Ambiguous Attibute Settings

By definition, besides processing functions, the abundant attribute settings and inquiry facilities of GKS are also bothersome for programmers. A topological view of them is given in Figure 15.

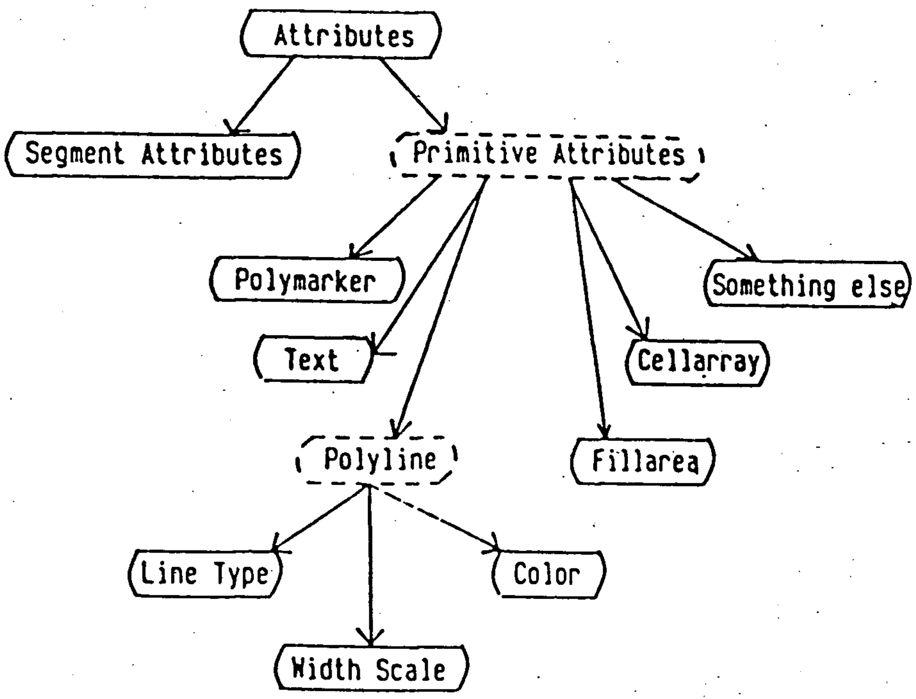


Figure 15 - Topological View of GKS Attributes

Each attribute in its family is set by an integer number rather than an individual name. Consider the colour attribute, it is set in GKS as in Figure 16.

colour	GKS setting
blank	0
green	1
blue	2
orange	3
yellow	4
red	5
purple	6
white	7

Figure 16 - The Colour Attributes of GKS

The attribute of text alignment is set in GKS as in Figure 17.

text alignment	GKS setting
(horizontal)	
normal	0
left	1
center	2
right	3
(vertical)	
normal	0
top	1
cap line	2
half	3
base line	4
bottom	5

Figure 17 - The Text Alignments of GKS

Basically, numbers do not provide any clue for a programmer as to its meaning. The overlapping of numeric values for the setting of primitives, workstations, segments and transformation attributes results in a rather misleading and confusing situation. For example, the number 1 represents green as a colour attribute, dot as a polymarker type attribute, segment 1 as a segment name attribute, etc. The repetition of numbers strongly block programmers to choose the proper candidate during the assignment process.

In addition, being built on a subroutine basis, the GKS system carries all the weaknesses and shortcomings of a subroutine package and presents its users the real model of GKS specifications rather than the virtual model at a higher level.

From above analysis, the high-level GKS language (HL/GKS), designed on a level above the GKS system, is studied and implemented in this thesis.

IV. STUDIES ON THE HIGH-LEVEL GKS LANGUAGE (HL/GKS)

4.1 Objectives

Being one important application area, graphics languages should keep pace with the development of today's programming languages with regard to abstract data structures and advanced programming features. However, the development of high-level graphics languages is still far from where it could be. Either lower-level drawing commands are performed or a set of subroutine patterns is issued. What special characteristics of graphics languages distinguishes today's programming languages? Stimulated by Wirth's definition on programming languages: "Programs = Algorithms + Datastructures", a similar formula defining computer graphics is: "Computer Graphics = Datastructures + Computer Algorithms + Languages". All structures and advanced features of programming languages are also relevant for the design of graphics languages.

From the software supporting point of view, graphics languages are classfied into different levels as shown in Figure 18.

Consider two classes, subroutine packages and extended languages. Extended languages maintain more graphical structures and process functions, which are very appealing for learning and utilizing a language. GKS, built on a subroutine basis, suffers from the shortage of high-level programming features and structures. As an international standard, GKS

Level 0	no support
Level 1	subroutine or macro libraries
	a) low level
	b) high level
Level 2	language extension
Level 3	special purpose graphic languages
	a) graphical approaches
	b) command languages

Figure 18 - The Levels of Software Support in Graphical Languages

provides all the basic graphics functions, terminal facilities and multiple processing for general applications. On the other hand, being a subroutine oriented system, GKS carries its control states, transformations, and function invocations into a poor background of frequent parameter assignments and subroutine calls. It is the programmer's responsibility to maintain the proper states, to set attributes correctly and invoke a subroutine for each function of GKS. As a simple example, a created segment can be displayed on an active workstation from the open workstation WISS (workstation independent segment storage). Therefore, to display a segment in such way, a series of actions need to be performed, as presented in Figure 19.

```
CALL GOPWK(1,8,100)       {open WISS}
CALL GOPWK(2,9,100000)    {open a workstation}
CALL GACWK(2)             {activate the workstation}
CALL GCSGWK(2,3)          {copy segment 2}
CALL GDAWK(2)             {deactivate the workstation}
CALL GCLWK(1)             {close WISS}
```

Figure 19 - A Segment Display in GKS

It is preferable to use a single display statement in a higher working environment, which replaces all control details, e.g.,

```
display house on screen
```

An abundant set of attributes, which are grouped as to the primitive they apply to, provide versatile choices for the display of pictures. Consider the text attribute, for which a toplogic view of its components is given in Figure 20.

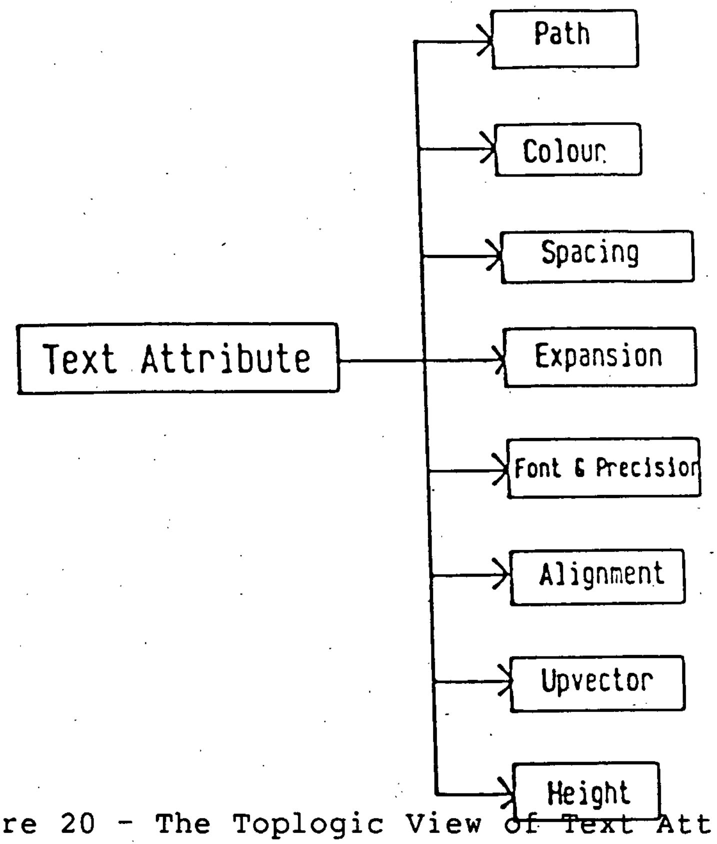


Figure 20 - The Toplogic View of Text Attribute

As pointed out before, the overlap of integer values representing the GKS attributes can cause difficulties to a programmer when choosing the proper value of an attribute for assignment. Using a name reference instead of a number for the attribute assignment will be more acceptable and reasonable, being more natural and providing clarity. A comparison of name and number assignments of the POLYLINE attribute is shown in Figure 21.

Another major motivation for the HL/GKS implementation is to supplement graphical types, data structures, and programming

In HL/GKS:

```
POLYLINE T(1:5)    < TYPE: 'dash',
                     COLOUR: 'yellow',
                     WIDTH: 0.5 >
```

In GKS:

```
CALL GSLN(2)         {line type assignment}
CALL GSPLCI(4)       {colour assignment}
CALL GSLWSC(0.5)     {line width assignment}
CALL GPL(5,tx,ty)    {POLYLINE output}
```

Figure 21 - Name and Number Setting of Polyline Attributes

format in terms of the graphics syntax and semantics specified in the extended portion of HL/GKS, elevating the poor background of the GKS system to a better GKS programming style at a higher level.

4.2 Graphical Types and Structures

Derived from GKS concepts, four graphical types are implemented in the HL/GKS language:

- vector,
- segment,
- trans,
- workstation.

Each of them can be declared as a single variable or as an array variable. Single vectors can replace any coordinate pair in GKS assignments or the output of primitive polymarker; an array vector then constructs a single output object. Array segments supply the alternatives to some elements of a structured object while a single segment remains an individual entity. An array trans simulates the multiple mapping of GKS coordinate space transformations. An array variable of type workstation produces multiple output by the hardcopy terminal. With the array type supplied, graphical variables of types vector, segment, trans, and workstation can be easily processed together with the programming structures of the host language FORTRAN 77. The graphical declaration statements are kept consistent with the other declaration statements of FORTRAN 77. Some instances of graphical expressions of HL/GKS are provided in Figure 22.

```
      VECTOR v1(5), v2(5), v3
      SEGMENT window, house(3), door(3)
      TRANS n1(2)
      WORKSTATION plotter(3), screen

      DO 10 i=1,3
         house(i) =  window + door(i)
10    CONTINUE
      DO 20 i=1,5
         IF s(i)>10 THEN
            v1(i) =  v2(i) - v3
20    ENDIF
```

Figure 22 - Some Graphical Expressions of HL/GKS

The type vector of HL/GKS is expressed in the form of (vx,vy) for the two-dimensional GKS situation. Obviously, it is easily extended to the three-dimensional space by the form of (vx,vy,vz). A vector variable can be assigned by a DATA statement of FORTRAN 77, a pair of real numbers or real expressions, or another vector variable name. A vector initialization in a DATA statement can be freely interspersed with other types of FORTRAN 77. No continuation character is required for the cross-line initialization of a vector in the DATA statement. The different forms of the vector assignment are presented in Figure 23.

```
VECTOR v1(2), vs(6), v2, v3(3)
DATA v1/1.0,2.3; 3.8,4.5/, r1/2.0, 0.8/
DATA vs/0.0,0.0; 7.8,9.0; 5.4,2.8; 7.0,5.0;
     8.4,9.0; 0.0,0.0/
v2 = (2.9, 1.8)
v3(1) = v2
v3(2) = (r1,r2)
v3(3) = (3*sin(i)+5.2, cos(i+2)/3.4)
```

Figure 23 - Vector Assignments of HL/GKS

For convenience, the subelements of a vector can be selected with the post fix .xcoord or .ycoord appended to the vector identifier, e.g.,

```
v3(1) = (v2.xcoord+1.9, v2.ycoord)
```

The vector operators applicable to vectors are listed in Figure 24.

```
+ addition
- subtraction
* scalar product
. dot product
/ scalar division
```

Figure 24 - The Vector Operators

With the introduction of the data type vector, graphical objects are easily modelled and processed since they are based on vertices, which form the necessary foundation for the applications of computational algorithms. Any coordinate pair can be assigned by a vector name; primitive objects are specified by the indicated range of vector arrays. Figure 25 shows some common occurrences of the type vector in HL/GKS. Alternatively, a subpart of a graphical object can be selected by the related subrange of its vector such as:

```
VECTOR atv,sclv, tree(15), table(10)
TEXT 'welcome to GKS' AT (1.5,0.8)
TEXT 'this is a text primitive' AT atv

star < AT: (1.0,0.9),
     SCALE: (0.5,2.0) >
star < AT: atv,
     SCALE: sclv >

POLYLINE tree(1:15)
FILLAREA table(1:10)
```

Figure 25 - Vector Placements in HL/GKS

```
POLYLINE tree(3:12)
```

The type segment of HL/GKS directly reflects segments of GKS. As from GKS, two working environments for segments are provided by the keywords OPEN and CLOSE. In the open environment, segment can be superimposed by predefined primitives and previously created segments within its specified coordinate space. In the closed environment, a segment can be modelled and processed by previously created segments through the superposition and deletion functions. A simple example of segment operations under both environments is given in Figure 26.

```
OPEN house
     house =  POLYLINE <TYPE: 'dash'>
                       square(1:5)
            + POLYMARKER <TYPE: 'circle',
                          CLOUR: 'red' >
                         handle
            + door + window(1)
CLOSE
flag =  star + stars + frame
house =  house - window(1) + window(2)
```

Figure 26 - Segment Operations in HL/GKS Environments

The data type trans of HL/GKS implements the transformations of the GKS coordinate spaces. The mappings are specified by rectangles. The rectangle to be mapped from is called the window of a transformation; the rectangle to be mapped to is called the viewport of a transformation. The keywords FROM and TO in a transformation pattern signal the window and viewport assignment of the transformation, respectively, such as:

```
TRANS nt
nt = < FROM: (3.8,2.9); (6.8,12.0)
       TO: (0.0,0.0); (0.5,1.0) >
```

A window or viewport assignment which is not specified in a transformation pattern will be set to the default assignment of GKS, viz. (0.,0.) to (1.,1.), or the one assigned previously. Normalization transformations and workstation transformations are distinguished by the assignment context. A transformation assigned to an opened segment as the segment open attribute is a normalization transformation; a transformation assigned to a viewing workstation as the workstation viewing attribute is a workstation transformation. A transformation can be specified either by a trans variable or by a transformation pattern. Examples of transformation assignments are presented in Figure 27.

The input priority of normalization transformation, to select the transformation order in the normalized device coordinate space, are indicated by the keywords of ABOVE and BELOW for each transformation pair.

```
OPEN house WITH nt(1) BELOW nt(2)
```

The data type workstation is designed to minimize state

```
TRANS nt(1)
TRANS wt
wt = <TO: (0.,0.); (3.6,5.8)>
nt(1) = <FROM: (3.5,2.0); (4.8,7.0)>

OPEN house WITH nt(1)
DISPLAY house ON screen WITH wt

OPEN house WITH < FROM: (3.5,2.0); (4.8,7.0) >
DISPLAY house ON plotter WITH < TO: (0.0,0.0);(3.6,5.8)>
```

Figure 27 - The Alternate Settings of Two Transformations

assignments of workstations such as "workstation open", "workstation active", etc. Workstations are opened when declared as a workstation type and activated when referred to in a display statement. Some long-term running workstation types are systematically defined in the precompiler, which are easily changed for updating a configuration. The workstation types used in HL/GKS are:

- screen,
- printer,
- plotter,
- t4027,
- t4025,
- t4105,
- t4112,
- t4113,
- t4115.

The display functions of GKS "associate segment with workstation" and "copy segment to workstation" are expressed by the keywords SEND and DISPLAY in a display statement of HL/GKS.

```
        SEND house TO plotter
        DISPLAY house ON screen
```

Using natural names, workstations are much more distinctly controlled and processed in HL/GKS in contrast to the use of numeric references of GKS. This is especially true in a multi-workstation system.

With the introduction of graphical data types, more high-level programming features and freedom from supplying system function details can be provided. Type declarations, type checking, and type parsing of HL/GKS aid in detecting mismatched graphical types and avoid passing incorrect parameters to the GKS system. Using types, the functions of the GKS system are simply performed as units by HL/GKS. Display statements in both GKS and HL/GKS are shown in Figure 28.

```
CALL GOPWK(1,8,100) {open works1}
CALL GOPWK(2,9,100000) {open works2}
CALL GACWK(2) {activate works2}
CALL GSVP(1,0.,0.5,0.,1.0) {set viewport1}
CALL GSELNT(1) {select norm trans1}
CALL GCSGWK(2,3) {copy seg3 to works2}
CALL GDAWK(2) {deactivate works2}
CALL GCLWK(2) {close works2}
CALL GCLWK(1) {close works1}

WORKSTATION plotter, screen
TRANS n1,n2
n1 = <TO: (0.,0.); (0.5,1.0)>
DISPLAY house ON plotter WITH n1
```

Figure 28 - A Display Fragment in GKS and HL/GKS Language

4.3 HL/GKS Implementation

HL/GKS is implemented as a high-level graphics language on the basis of the GKS system by a precompiler. The precompiler is specified by two separate but interrelated files and generated with the Top-Down Compiler Writing System [Schr82]. The files of the precompiler are:

- integrated BNF description file,
- global file in standard PASCAL.

The source file contains the syntax and semantics rules of the language; the global file is designated for additional semantics actions. The interface of the precompiler of HL/GKS to the CWS-TD system is shown in Figure 29.

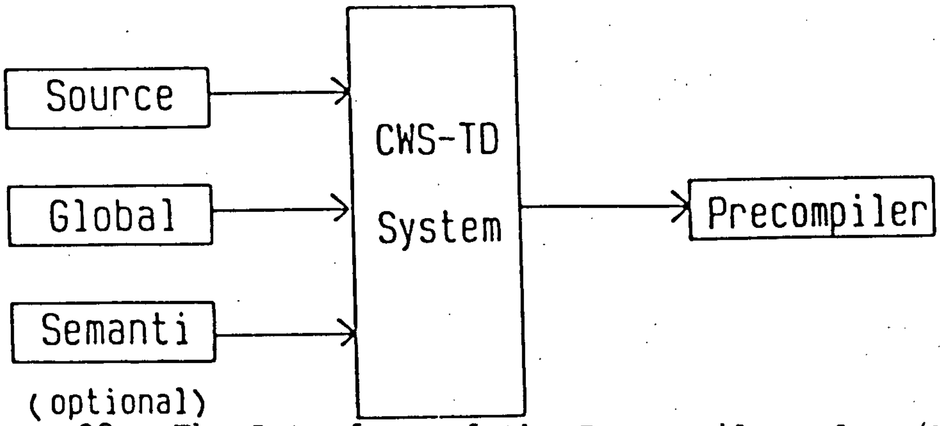


Figure 29 - The Interface of the Precompiler of HL/GKS to the CWS-TD System

The precompiler translates graphical statements of an HL/GKS program into GKS subroutines invocations which are then compiled by the FORTRAN 77 compiler and executed with the GKS runtime library. The sequence of transformation steps of a HL/GKS program is illustrated in Figure 30.

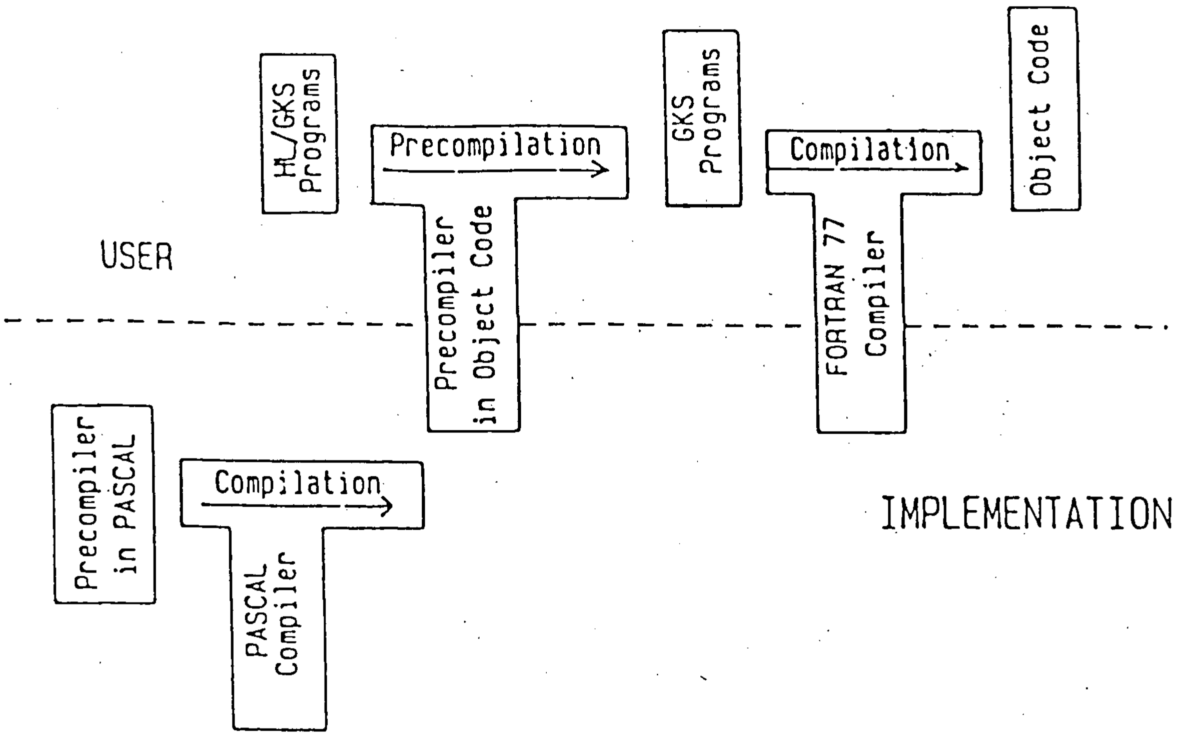


Figure 30 - The Transformation Steps of the HL/GKS Language

To provide a better working environment and on a higher level, most system control and specification details of the GKS system are hidden or are transparent to the HL/GKS programmer. Thus, a HL/GKS programmer specifies the GKS functions and facilities in his programs unconsciously and implicitly.

Four system models are internally built up during the precompiling phase by means of the graphical data types:

. vector model,
. segment model,
. transformation model,
. workstation model.

The concrete structure of each of the models is depicted in Figure 31.

During precompilation, type checking, state changes, and function mappings are carried out based on the information

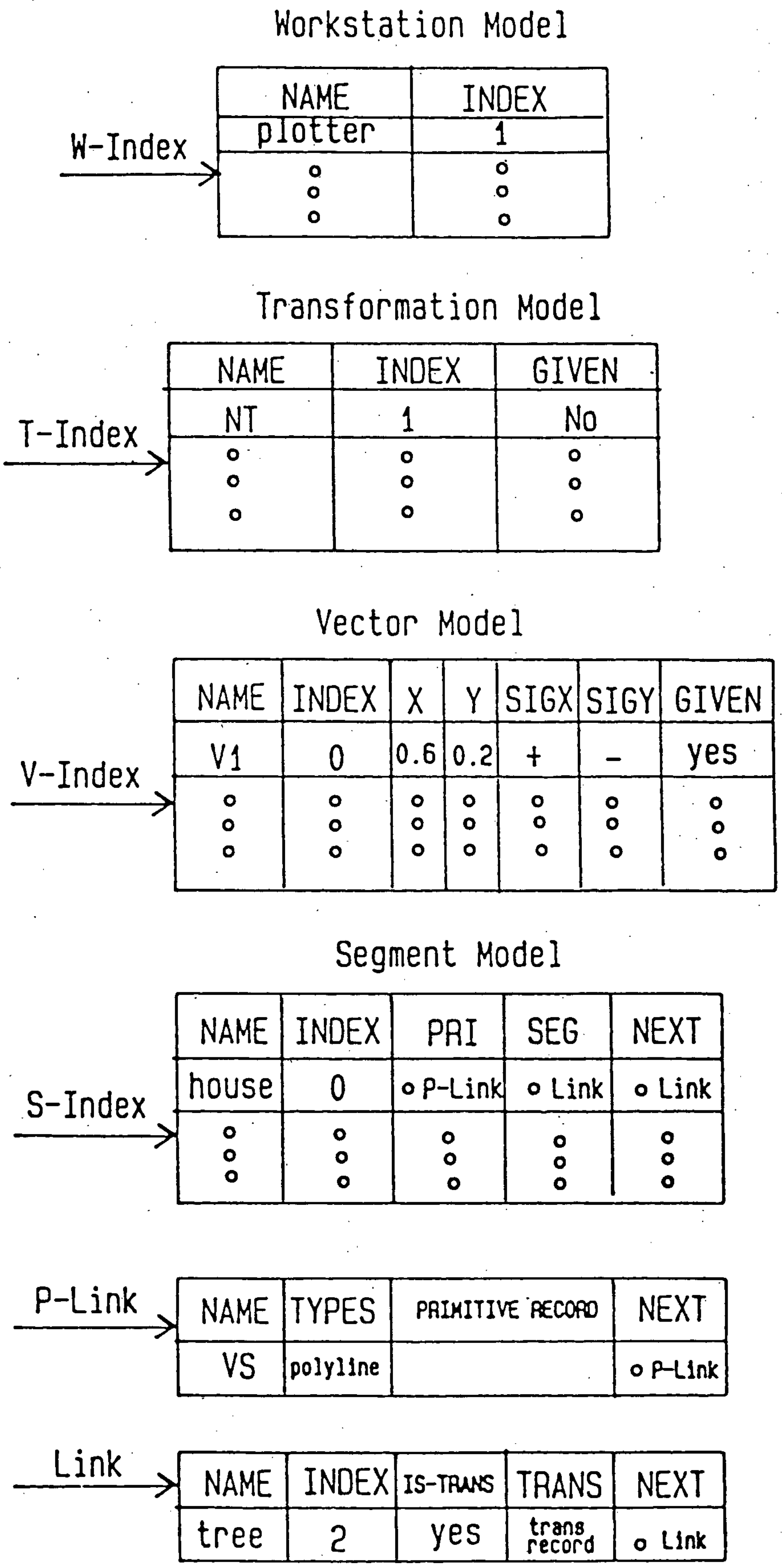


Figure 31 - The Structures of the Data Models

provided from above models. Consider the segment mode. Each entry of the segment table contains three pointers --- seg, pri, and next. The pointer seg links all the present segment components of the entry segment which were constructed previously, the pointer pri stores all the information of the output primitives of the entry segment in its opening state, and the pointer next collects all the superimposed segments which are referred to in the present operation. In addition, M-link is a globally defined pointer chain which keeps all the segments which are to be deleted in the present operation. The transformations to be associated with each segment are also recorded for each instance of a model when needed. With models so constructed, the operations of superposition and deletion of segments are executed based on the stored models; the system updates them both internally and externally.

In HL/GKS, the two kinds of statements, namely host language and graphics extension statements, can be interspersed freely in a program. The graphical statements of HL/GKS are identified and translated by the following steps: lexical scan, syntactic parsing, semantic analysis, and GKS code generation. A language L(G), produced by a grammar G, is formulated as:

$$L(G) = \{ S \epsilon T^* \mid S \Rightarrow s \}$$

Here, T --- a finite set of terminal symbols,

S --- non-terminal start symbol,

s --- legal sentence in language.

Accordingly, the graphical extension of a language must follow the same definition. What syntax and semantics rules need to be supplied for the HL/GKS language? What lexical layout is adopted for the graphical part of HL/GKS? To answer these questions, the implementation of HL/GKS is explored from the following aspects:

. lexical usage,

. syntax rules,

. semantics rules.

4.3.1 Lexical Usage

In HL/GKS, all identifiers and keywords can be entered using either lower case or upper case characters. Intermixing of characters is acceptable and not distinguished. For instance, the keywords VECTOR, vector or VecTor are equivalent. No continuation mark in column 6 is required to flag the continuation of a line of a graphical statement as long as the proper syntax rules for the statement are followed. An example of a graphical layout of HL/GKS is presented in Figure 32.

```
curve =  POLYLINE <TYPE: 'dash'> xsin(1:51)
       + TEXT     <HEIGHT: 0.03,
                   SPACE: 0.8>
            'sine curves' AT (0.05,0.1)
```

Figure 32 - The Graphical Layout of HL/GKS

Other card restrictions of FORTRAN 77, such as characters "C" or "*" in column 1 for a comment line, columns 2 to 5 for statement labels, and columns 7 to 72 for language statements are not effective for the layout of graphical statements of HL/GKS. Labels can be set in columns 7 to 72, some programming portion can be expressed in columns 2 to 5. Blank lines may be used in HL/GKS programs for improved readability. Comments included between the delimiters "{" and "}" can be freely placed in HL/GKS, and will be ignored during the precompiling process.

Two loop structures are supplied by HL/GKS to have more control on the language flow. Figure 33 shows the syntactical expression of the structures.

```
WHILE  <boolean expression>  DO
       <statements>
ENDWHILE

REPEAT
    <statements>
UNTIL <boolean expression>
```

Figure 33 - Syntactical Expression of New Structures

Block compiling is possible in the precompiling phase of HL/GKS to give freedom of model programming and local debugging. The main block and subroutine blocks of a program may be compiled separately or in an arbitary sequence, which is represented in the syntax rule of:

```
<HL/GKS blocks> ::= { <main block>
                    |<sub block> }1
```

All graphical data types can be passed to a subroutine when declared in its parameter list. The declaration of the graphical types in subroutine blocks follows the same declaration rule as for the main block.

Identifiers for all graphical data types except vector can be up to 15 characters long, providing a more reasonable reference of a full name. Vector identifiers are presently restricted to 5 characters as each vector variable will be translated into a pair of real variables in the compiled output.

If the programmer refrains from introducing identifiers beginning with the characters G or g, no conflicts will arise with the identifiers of the GKS system.

4.3.2 Syntax Rules

As pointed out earlier, the syntax of the GKS system is mainly represented by the following aspects:

- state flows,
- subroutine protocols.

GKS should always be in one of the following states:

- GKS closed (a),
- GKS open (b),
- at least one workstation open (c),
- at least one workstation active (d),
- at most one segment open (e).

State changes are restricted to a sequential order of flow in both directions. Within each flagged state, only some subroutines can be legally invoked. As a result, in GKS it is the programmer's responsibility to be in the proper state for legal subroutine invocations. In HL/GKS, only one state restriction is imposed on the programmer:

- at most one segment open

Outlines of the states in both languages are presented in Figure 34.

The other states are internally controlled by the precompiler according the syntax and semantics definition of HL/GKS. The BEGIN statement in the main block sets the state of "at least one workstation active" through the states of "GKS open" and "at

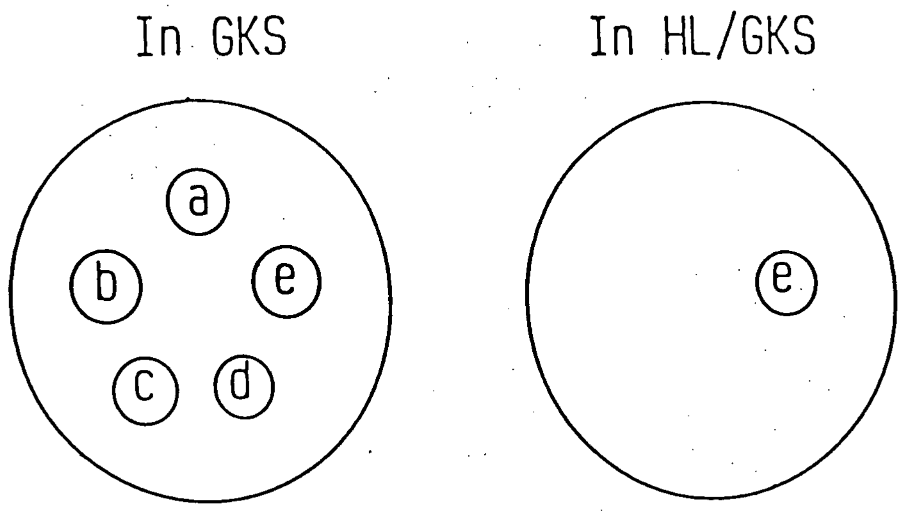


Figure 34 - The Outlines of the States in Both Languages

least one workstation open"; the END statement in the main block returns the system to the state of "GKS closed". The state of "at most one segment open" is optionally switched on or off by the appropriate statements of HL/GKS. The segment open statement sets the state of "at most one segment open" on; the segment close statement turns it off. With only one state outlined, the graphical statements of HL/GKS are simply partitioned into two groups, "segment open" and "segment not open". A display statement (to copy segments to the specified workstations) may not be issued in the state "segment open", the clear statement (to clear all segments from the specified workstations) is also not accepted when a segment is open. The "segment closed" environment is assumed to be the state in each subroutine block, unless it has been changed with the keyword OPEN appended to a segment parameter.

```
        SUBROUTINE find( ts:OPEN, r)
        SEGMENT ts
```

Few restrictions on state changes for proper function execution are levied on a HL/GKS programmer.

Using the language extension approach, HL/GKS removes the restrictions of the subroutines protocol of GKS. The GKS control and system specifications are abstracted and incorporated into the HL/GKS definition. The major components of the GKS system are presented in Figure 35.

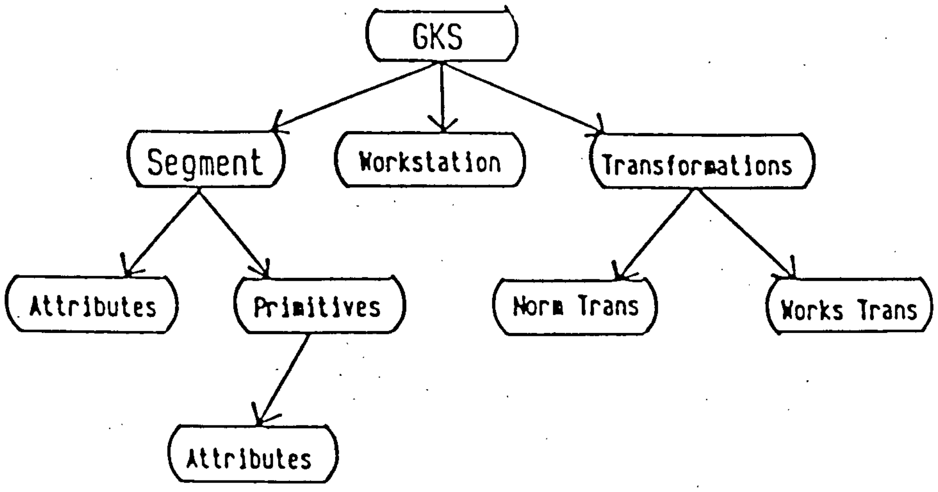


Figure 35 - The Major Components of the GKS System

Five standard output primitives are defined in the GKS document:

- polyline,
- polymarker,
- fillarea,
- cellarray,
- text.

Instead of invoking a subroutine for each primitive, keywords are combined with vectors to represent primitives. The keywords via the primitive names indicate the output types; a modelled vector array resembles the structured object through the feature points in the vector. Some examples of primitive generation in HL/GKS are presented in Figure 36.

For convenience, the primitive cellarray is specified based on a

```
POLYLINE pg(1:10)
POLYMARKER mg(1:10)
FILLAREA fg(3:10)
CELLARRAY <RANGE: (px,py); (qx,qy)
           MAXROW: dimx>
          colia(dx,dy)
TEXT 'GOODBYE' AT (ax,ay)
```

Figure 36 - Output Examples of HL/GKS

two-dimensional real array.

Appropriate attributes may be applied to an output primitive which are subsequently assigned to each primitive keyword to form a whole primitive concept. Two assignment approaches of bundled and individual attributes of GKS are implemented in HL/GKS. The keyword TABLE set in the attribute pattern flags the bundled function of GKS; the assignment of an attribute places each attribute to individual one at a time. An assignment of type individual is specified by a related keyword, directly inferred from the GKS document, such as TYPE, COLOUR, and FONT. Common attributes of primitives are distinguished by the assignment context. For example, the keyword TYPE refers to either a type of polyline or a type of polymarker. Parameters of two data types, CHARACTER and REAL, are used in attribute assignments and the examples are the attribute COLOUR, set by the name of the colour, and the attribute SPACE, set by a number of type real. Constants, variables, as well as expressions are acceptable in assignment statements. Several alternative attributes assignments are shown in Figure 37.

Multiple primitive specification of the same primitive type is supported in HL/GKS, but still allowing individual attribute assignment for each primitive. An instance of a multiple

```
POLYLINE <TABLE: 2>

POLYLINE <TYPE: 'dash',
          WIDTH: 0.6 >

POLYLINE <WIDTH: 0.1+0.5>

CHARACTER*5 ptype
ptype = 'dash'
r1 = 0.6
POLYLINE <TYPE: ptype,
          WIDTH:r1 >
```

Figure 37 - Alternative Settings of Attributes

assignment of the primitive polyline is given in Figure 38.

```
house = POLYLINE <TYPE: 'dot'>
                 round(1:16); block(2:20);
                 <TYPE: 'solid'>
                 side(1:5)
```

Figure 38 - Multiple Output of the Primitive Polyline

The workstation independent segment storage (WISS) is opened and activated at the time of system opening to provide a global processing environment for segments. Other workstations are opened when declared in the declaration statement and activated when referred in the display statement. This use of workstations combined with the global modelling of segments simplified most system restrictions.

The utilization of input functions allows an interactive approach to picture modelling. Pictures can be defined by their vertices read on-line, avoiding input by meaningless data patterns. The graphical input statements of HL/GKS are implemented in a form consistent with the input statements of FORTRAN 77. They are:

```
READ (wkid, devicetype) graphical variable
```

{ , graphical variable }

In comparison to FORTRAN 77, wkid resembles the logic unit name from which data is to be read while devicetype imitates the read format specification to be applied. Five logic input devices of GKS are implemented in the HL/GKS language as:

- locator,
- string,
- stroke,
- valuator,
- choice.

Each logic device is distinguished by the type of graphical variables which is specified in the input statement. The corresponding data types for the logic devices are listed in Figure 39.

Logic Device	Data Type
locator	vector
string	character
stroke	array vector
valuator	real
choice	integer

Figure 39 - The Data Types for the Logic Devices

Two transformations, normalization transformation and workstation transformation, defined in the GKS document to aid in device independence, are combined in one transformation type (type trans) and are distinguished by different syntactical assignment rules. When a trans variable is assigned to an open segment statement, it refers to a normalization transformation, otherwise to a workstation transformation assigned to a viewing workstation.

The clipping function of the GKS transformation, to remove the overlapping part of the clipping rectangle (normalization transformation viewport) with a workstation window on NDC space, is flagged on or off by the statements CLIPPING ON or CLIPPING OFF, respectively.

In GKS, one or several segments can be displayed on one or several workstations by means of the display statement. The symbols of "+" and "," serve as separators on the multiple assignment of segments and workstations in an output statement. As for GKS, output primitives issued during state are "segment closed" included in the display streams of HL/GKS. The two segment viewing attributes visibility and highlighting can be assigned to a segment name in the display statement. Figure 40 shows several statements in HL/GKS which cause multiple output.

```
VECTOR vstar
  ...
DISPLAY tree + house<highlighting> ON screen
DISPLAY flag, polymarker<type:star> vstar
              ON screen, plotter
REDRAW 4027, 4014
SEND curve, title + axis TO plotter
```

Figure 40 - Multiple Output Display Statements of HL/GKS

A comparison of multiple output in HL/GKS and GKS is given in Figure 41.

The detailed syntax rules of the HL/GKS extension are presented in BNF form in Appendix A.

4.3.3 Semantics Rules

In HL/GKS:

```
DISPLAY flag, flags ON screen, plotter, printer
```

In GKS:

```
CALL GOPWK(2,9,100000)
CALL GOPWK(3,10,100000)
CALL GOPWK(4,11,100000)
CALL GACWK(2)
CALL GACWK(3)
CALL GACWK(4)
CALL GCSGWK(2,1)
CALL GCSGWK(2,2)
CALL GCSGWK(3,1)
CALL GCSGWK(3,2)
CALL GCSGWK(4,1)
CALL GCSGWK(4,2)
CALL GDAWK(4)
CALL GDAWK(3)
CALL GDAWK(2)
CALL GCLWK(2)
CALL GCLWK(3)
CALL GCLWK(4)
```

Figure 41 - Multiple Output in HL/GKS and GKS

Implemented on the basis of GKS, the semantical definitions of HL/GKS have been kept as close as possible to the GKS concepts. The keyword BEGIN in a main block opens the GKS system and the declared workstations; the keyword END in the main block closes the system and all workstations. The assignment rules of GKS are applied in HL/GKS. Attributes and transformations which are not assigned a specific value are assigned default values. As examples, the default value for the attribute polyline type is solid; the default viewport of a normalization transformation is the entire NDC space. A value once assigned will be effective until a new value is assigned. Some instances of assignments are given in Figure 42.

```
OPEN trees WITH nt
     ...
CLOSE
OPEN house
     house:- POLYLINE outs(1:5);
             <TYPE:'dash'> left(1:5);
                           right(1:5)
CLOSE
```

Figure 42 - Some Assignment Instances of HL/GKS

In Figure 42, the normalization transformation nt is assigned to the segments trees and house; the attribute polyline type dash is assigned to the polylines left and right while the default value solid is used for the attribute of polyline outs.

A grammar for a language to be parsed by the Compiler Writing System is required to be of type LL(1); it determines the language parts by only one character look ahead from its left-most derivation, and in the order of left to right. Therefore, certain graphical symbols and operators are specified in the extended portion of HL/GKS to allow proper language parsing. The symbols "<" and ">" act as delimiters of attribute assignments; the symbol ":" precedes each parameter in an individual assignment. The operators "+" and "-" are overloaded and have different meanings for each process type. They are referred to as vector addition and subtraction for the vector operands or as segment superposition and deletion for the segment operands. The semantics of the graphical symbols and operators in the HL/GKS definition is presented in Figure 43.

No special symbol is applied for the graphical assignment of HL/GKS, it is parsed by means of the system semantics checking on each graphical type, thus unifying the concept of

<	left delimiter of attribute assignment
>	right delimiter of attribute assignment
:	individual attribute assignment
;	delimiter of vectors
,	delimiter of multiple attributes, segments and workstations
+	vector addition or segment superposition
-	vector subtraction or segment deletion
*	vector product with a scalar
/	vector division by a scalar
.	vector dot product

Figure 43 - The Semantics of Symbols and Operators of HL/GKS

graphical and nongraphical assignment. For segment modelling, a segment object may be modelled level up with reference to itself or redefined directly by other segment components.

```
house = house + tree

house = tree
```

Each declared workstation is matched with the standard workstation types of HL/GKS to check any misuse of workstation types.

The graphical input statements of HL/GKS are distinguished from the read statements of FORTRAN 77 by the identifiers wkid and devicetype. The five logic devices of GKS are parsed by the type of the graphical variable in the read statement. For example, the data read from the device locator should be a pair of real numbers, which are represented by the type vector in HL/GKS. No distinction is made between the format of graphical input statements, FORTRAN input statements, or logic device input statements. Two kinds of device types of GKS (keyboard and graphical tablet) are referred to by literals in the read statement of HL/GKS. Figure 44 demonstrates read statements of HL/GKS of each logic device.

```
CHARACTER*10 title, axis
VECTOR  vp1,vp2,vps(10)
REAL    rs1,rs2
INTEGER ci1,ci2

READ (4027, keyboard)vp1      {locator}
READ (screen,keyboard) title  {string}
READ (4027, tablet) vps(1:10) {stroke}
READ (4012, keyboard) rs1     {valuator}
READ (screen,keyboard)ci1     {choice}
```

Figure 44 - Read Statements of HL/GKS

More than one variable (but of same type) may appear in the list of an input statement. A subrange can be specified by an upper and lower bound of vector array for input of the stroke device.

```
READ (4027, keyboad) vp1, vp2
READ (4027, tablet) vps(2:8)
```

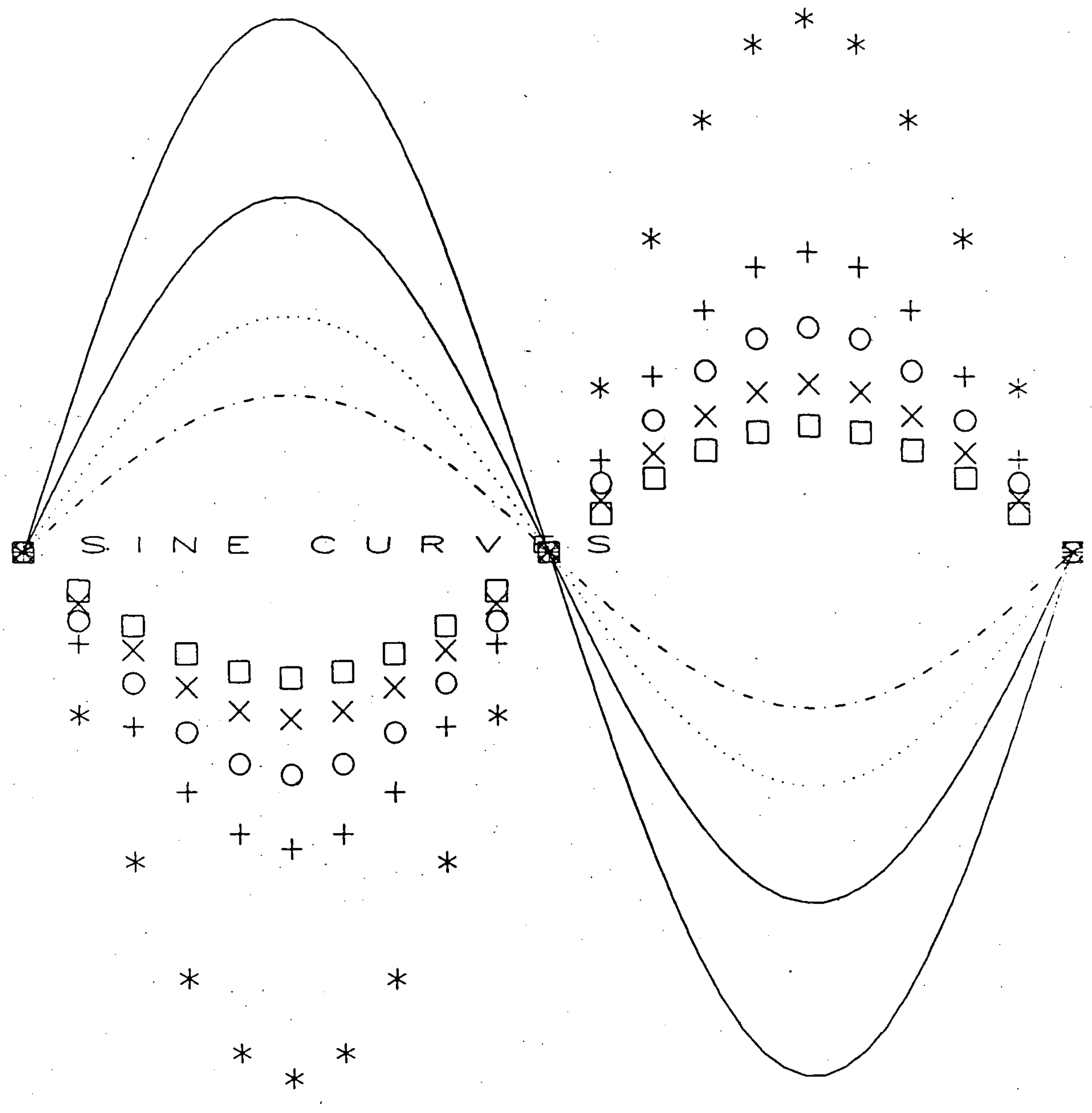
V. EXAMPLES OF GKS AND HL/GKS LANGUAGE PROGRAMS

There are four examples presented in the following section. Example 1 and Example 2 are given both in HL/GKS and GKS, the programs produce the identical picture output, and provide a direct comparison for an evaluation of the HL/GKS implementation. Example 3 and Example 4 in HL/GKS show further characteristics and application examples of the HL/GKS language. Each output produced follows the program listing to present the visual effects. Among the examples, Example 1 is taken from the original demonstration example of the GKS document.

```
    {                                   }
    {      EXAMPLE 1 IN HL/GKS          }

    SEGMENT CURVE
    CHARACTER*9 SETS
    VECTOR XSIN(51)
    WORKSTATION PLOTTER, SCREEN

    BEGIN  {GKS}

    OPEN CURVE WITH <FROM: (0.0,-1.0); (1.0,1.0)>
         MESSAGE 'GKS CAN GENERATE TEXT' ON SCREEN
         CURVE= TEXT <HEIGHT:0.03,
                      SPACE:0.8> 'SINE CURVES' AT (0.05,0.0)
         MESSAGE 'GKS CAN GENERATE POLYLINES' ON SCREEN

         DO 10 K=1,51
            R1 = FLOAT(K-1)*0.02
            XSIN(K)= (R1, SIN(R1*6.283))
10       CONTINUE

         CURVE= CURVE + POLYLINE XSIN(1:51)

         DO 30 K=1,3
            DO 20 KK=1,51
20          XSIN.YCOORD(KK)=XSIN.YCOORD(KK)*0.667
            GO TO (1,2,3),K
1           SETS= 'DASH'
            GO TO 35
2           SETS= 'DOT'
            GO TO 35
3           SETS= 'DASHDOT'
35          CURVE= CURVE + POLYLINE <TYPE: SETS> XSIN(1:51)
30       CONTINUE

         MESSAGE 'GKS CAN GENERATE POLYMARKERS' ON SCREEN

         DO 40 K=1,21
            R1 = FLOAT(K-1)*0.05
            XSIN(K)= (R1, -SIN(R1*6.283))
40       CONTINUE

         CURVE= CURVE + POLYMARKER XSIN(1:21)

         DO 60 K=1,5
            DO 50 KK=1,21
50          XSIN.YCOORD(KK)=XSIN.YCOORD(KK)*0.75
            GO TO (5,6,7,8,9),K
5           SETS= 'DOT'
            GO TO 65
6           SETS= 'PLUS'
            GO TO 65
7           SETS= 'CIRCLE'
```

```
               GO TO 65
8              SETS= 'CROSS'
               GO TO 65
9              SETS= 'SQUARE'
65             CURVE= CURVE + POLYMARKER <TYPE: SETS>
                                            XSIN(1:21)
60          CONTINUE

     CLOSE

     DISPLAY CURVE ON PLOTTER
     STOP
     END
```

```
      PROGRAM DEMO1
*
*   This program demonstrates how GKS may be used
*
*   The following primitives will be used:
*      - POLYLINE
*      - POLYMARKER
*      - TEXT

      INTEGER K,L,M

      REAL SINX(51),SINY(51),TWOPI/6.28318/

      CHARACTER *7 FNAME, CFNAME

*   Open unit 21, the error message file.  The open statement
*      attaches the unit to the temporary file "-GKSERR".
*
      FNAME = '-GKSERR'
      OPEN (UNIT=21,FILE=FNAME,STATUS='UNKNOWN')
*
*   Specify the output metafile.  It is assigned to unit 20.
*
      CFNAME = '-METAFL'
      OPEN (UNIT=20,FILE=CFNAME,IOSTAT=IOS,STATUS='UNKNOWN')
      IF (IOS.NE.0) THEN
         WRITE(6,*) 'Cannot open output metafile'
         STOP
      END IF
*
*   Open the Graphical Kernel System
*
      CALL GOPKS(21)
*
*   Open and activate a workstation.
*   The workstation selected is "metafile out"
*      defined by the systems constant '100000'.
*
      CALL GOPWK(1,20,100000)
      CALL GACWK(1)
*
*   Define and select a transformation that directs output
*      to a window of (0.0,1.0) x (-1.0,1.0)
*
      CALL GSWN(1,0.0,1.0,-1.0,1.0)
      CALL GSELNT(1)
*
*   Begin output:
*   Start with a set of polylines, to be output with four
*      different linetypes starting with the default (1 = solid)
*
      CALL GMSG(1,'GKS can generate polylines')
      DO 10 K=1,51
         SINX(K) = FLOAT(K-1)*0.02
         SINY(K) = SIN(SINX(K)*TWOPI)
   10 CONTINUE

      CALL GPL(51,SINX,SINY)

      DO 30 K=2,4
         L = K
         CALL GSLN(L)
         DO 20 M=1,51
            SINY(M) = SINY(M)*0.667
   20    CONTINUE
         CALL GPL(51,SINX,SINY)
   30 CONTINUE
*
*  Next output a set of polymarkers starting with
*     the default marker (3 = asterisk)
*
      CALL GMSG(1,'GKS can generate polymarkers')

      DO 40 K=1,21
         SINX(K) = FLOAT(K-1)*0.05
         SINY(K) = -SIN(SINX(K)*TWOPI)
   40 CONTINUE

      CALL GPM(21,SINX,SINY)

      DO 60 K=1,5
         M = K
         CALL GSMK(M)
         DO 50 M=1,21
            SINY(M) = SINY(M)*0.75
   50    CONTINUE
         CALL GPM(21,SINX,SINY)
   60 CONTINUE
*
*  Finally, output a text string - default is font 1,
*     select a reasonable character height and spacing
*
      CALL GMSG(1,'GKS can generate text')
      CALL GSCHH(0.03)
      CALL GSCHSP(0.80)
      CALL GTX(0.05,0.00,'sine curves')
*
*  Deactivate and close the workstation and GKS
*
      CALL GDAWK(1)
      CALL GCLWK(1)
      CALL GCLKS
      STOP
      END
```


SINE CURVES

```
      {                                        }
      {       EXAMPLE 2 IN HL/GKS              }

      subroutine maker( radius, v)

      {  generate up the vectors of a circle   }

      real radius
      vector v(37)

      begin
      angle= 0.0

      do 10 i=1,37
         r= angle* 3.14159/180.
         v(i)= (radius*cos(r)+1.5, radius*sin(r)+1.5)
         angle= angle+10
10    continue

      return
      end

      SEGMENT CIRCLES, SUBCIR, TITLE(2),  CURVE(2),
              XAXIS, YAXIS, FRAME, PICTURE(2)
      VECTOR CUR(38), ROUND(37), POINT(2),
             CUR1(100), CUR2(100), SIDE(5)
      TRANS NT(2)
      WORKSTATION PLOTTER, T4014
      DATA SIDE/0.0,0.0; 3.0,0.0; 3.0,2.0; 0.0,2.0; 0.0,0.0/

      BEGIN  {GKS open}

      {  set two transformations  }

      NT(1)= <FROM: (-1.5,0.0); (4.5,3.0)
                TO: (0.0,0.5); (1.0,1.0)>
      NT(2)= <FROM: (-1.0,-1.0); (4.0,3.0)
                TO: (0.0,0.0); (1.0,0.5) >
      ANGLE= 0.0

      DO 10 I=1, 38
         R=ANGLE* 3.14159/180.
         CUR(I)=( 0.2*(COS(R)+R*SIN(R))+1.3,
                  0.2*(SIN(R)-R*COS(R))+1.5)
         ANGLE= ANGLE+10.
  10  CONTINUE

      {  create the images on the upper-half plane of output }

      OPEN CIRCLES WITH NT(1)
           R=0.2
           CALL MAKER(R,ROUND)
           CIRCLES= POLYLINE ROUND(1:37)
           DO 20 I=1,6
              R=R+0.1
              CALL MAKER(R,ROUND)
              CIRCLES= CIRCLES+POLYLINE ROUND(1:37)
 20        CONTINUE

      CLOSE

      OPEN SUBCIR
           CALL NUMBER(0.0,0.3,30.0,SUBCIR)
           CALL NUMBER(0.3,0.6,15.0,SUBCIR)
           CALL NUMBER(0.6,0.8,4.0,SUBCIR)
      CLOSE

      OPEN CURVE(1)
           CURVE(1)= POLYLINE CUR(1:38)

           DO 30 I=1,19
              CUR(I)=(CUR.XCOORD(2*I), CUR.YCOORD(2*I))
 30        CONTINUE

           CURVE(1)= CURVE(1)+POLYMARKER CUR(1:19)
      CLOSE

      OPEN TITLE(1)
           TITLE(1)= TEXT <HEIGHT: 0.09,
                                 SPACE: 0.15>
                  'A POLAR PLOT WITH GRID'
                         AT (0.6,2.6)
      CLOSE

      PICTURE(1)= TITLE(1)+ CURVE(1) < SCALE: (0.5,0.7)
                                       AT: (1.5,1.5) >  + CIRCLES

      {  create the images on the lower-half plane of output  }

      OPEN TITLE(2) WITH NT(2)
           TITLE(2)= TEXT <HEIGHT: 0.1,
                                  SPACE: 0.2 >
                   'XGRAF WITH A GRID' AT (0.5,2.5)
      CLOSE

      OPEN FRAME
           FRAME= POLYLINE SIDE(1:5)
           POINT(1)= (0.0,0.0)
           POINT(2)= (0.0,2.0)

           DO 50 I=1,10
              DO 60 J=1,2
60            POINT.XCOORD(J)=POINT.XCOORD(J)+0.3
              FRAME= FRAME+POLYLINE POINT(1:2)
50         CONTINUE
```

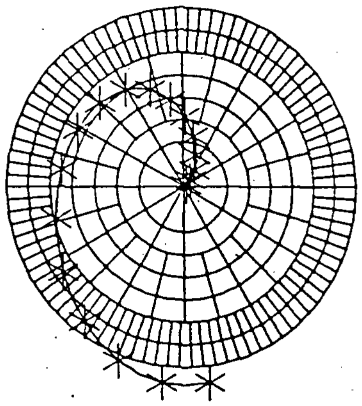
```
          POINT(1)= (0.0,0.0)
          POINT(2)= (3.0,0.0)

          DO 70 I=1,10
             DO 80 J=1,2
80           POINT.YCOORD(J)=POINT.YCOORD(J)+0.2
             FRAME= FRAME+ POLYLINE POINT(1:2)
70        CONTINUE
       CLOSE

       DO 90 I=1,100
          X= FLOAT(I)/ 100.
          CUR1(I)=(X*3.0,EXP(-3.*X)*SIN(X*8.*3.14159)+1.0)
          CUR2(I)=(X*3.0,EXP(-3.*X)*COS(X*8.*3.14159)+1.0)
90     CONTINUE

       OPEN CURVE(2)
            CURVE(2)=POLYLINE CUR1(1:100); CUR2(1:100)
       CLOSE

       OPEN XAXIS
            XAXIS= TEXT <HEIGHT: 0.05,
                               SPACE: 0.93>
                   '0.0 0.1 0.2 0.3 0.4 0.5 0.6'
                    AT (-0.1,-0.2);
                   '0.7 0.8 0.9 1.0' AT (1.96, -0.2);
                    < HEIGHT: 0.08, SPACE: 0.15>
                   'X AXIS WITH GRAF TYPE' AT (0.5,-0.5)
       CLOSE

       OPEN YAXIS
            YAXIS= TEXT <UPVEC:(-1.0,0.0)
                               HEIGHT: 0.05,
                               SPACE: 0.8 >
                   '-1.0 -0.6 -0.2 +0.2 +0.6 +1.0'
                              AT (-0.1,-0.1);
                   <SPACE: 0.12, HEIGHT: 0.07 >
                   'Y AXIS WITH GRAF TYPE' AT (-0.32,0.2)
       CLOSE

       PICTURE(2)= FRAME+ CURVE(2)+ TITLE(2)+ XAXIS+ YAXIS
       DISPLAY PICTURE(2) ON PLOTTER
       DISPLAY PICTURE(1), SUBCIR ON PLOTTER
       STOP
       END  {GKS close}

       {  number the radius lines between two circles  }

       SUBROUTINE NUMBER(r1,r2,degree,sub:OPEN)
       VECTOR p(2)
       SEGMENT sub
       REAL degree

       BEGIN
```

```
      angle= 0.0
      j=360./degree

      DO 10 I=1,j
         p(1)= (r1*COS(angle)+1.5, r1*SIN(angle)+1.5)
         p(2)= (r2*COS(angle)+1.5, r2*SIN(angle)+1.5)
         sub= sub+ POLYLINE p(1:2)
         angle=angle+degree*3.14159/180.
10    CONTINUE

      RETURN
      END
```

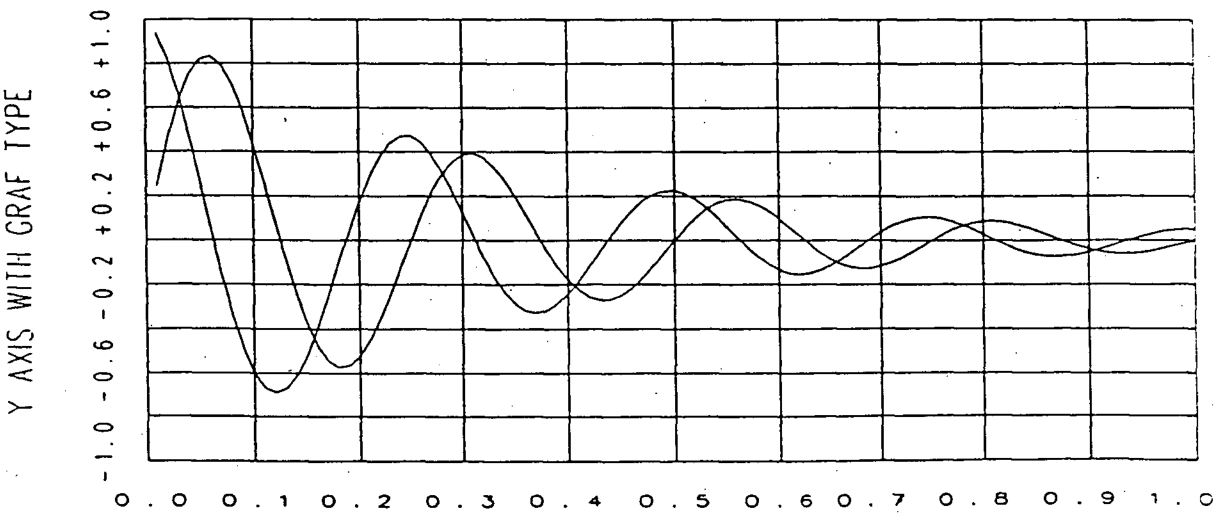
```
C
C       EXAMPLE 2 IN GKS
C
C       generate two circle arrays
        SUBROUTINE MAKER(RIDUS,VX,VY)
        REAL VX(37),VY(37)
        REAL RIDUS
C
        ANGLE= 0.0
        DO 10 I=1,37
        R= ANGLE* 3.14159/180.
        VX(I)=RIDUS*COS(R)+1.5
        VY(I)=RIDUS*SIN(R)+1.5
        ANGLE= ANGLE+10
 10     CONTINUE
        RETURN
        END
C
C
        REAL CURX(38),CURY(38)
        REAL ROUNDX(37),ROUNDY(37)
        REAL POINTX( 2),POINTY( 2)
        REAL CUR1X(100),CUR1Y(100)
        REAL CUR2X(100),CUR2Y(100)
        REAL SIDEX( 5),SIDEY( 5)
        CHARACTER*4 GWF1, GWF2, GF
        REAL GTRAN1(2,3), GTRAN2(2,3)
        DATA SIDEX/ 0    , 3.00, 3.00, 0    , 0    /
        DATA SIDEY/ 0    , 0    , 2.00, 2.00, 0    /
C
C       open GKS system
        GF='-ERR'
        OPEN(UNIT=20,FILE=GF,STATUS='UNKNOWN')
        CALL GOPKS(20)
        CALL GOPWK(1,8,100)
        CALL GACWK(1)
        GWF1='-PLOTTE'
        GWF2='-T4014 '
        OPEN(UNIT=14,IOSTAT=IOS,FILE=GWF1,STATUS='UNKNOWN')
        OPEN(UNIT=15,IOSTAT=IOS,FILE=GWF2,STATUS='UNKNOWN')
        CALL GOPWK( 2,14,100000)
        CALL GOPWK( 3,15,100000)
C
C       set two transformations of number 1 and number 2
        CALL GSWN( 1,-1.5,4.5,0.0,3.0)
        CALL GSVP( 1,0.0,1.0,0.5,1.0)
        CALL GSWN( 2,-1.0,4.0,-1.0,3.0)
        CALL GSVP( 2,0.0,1.0,0.0,0.5)
C
C       create the upper-half image
        ANGLE= 0.0
        DO 10 I=1, 38
        R=ANGLE* 3.14159/180.
```

```
      CURX(I)= 0.2*(COS(R)+R*SIN(R))+1.3
      CURY(I)=0.2*(SIN(R)-R*COS(R))+1.5
      ANGLE= ANGLE+10.
 10   CONTINUE
      CALL GSELNT( 1+1-1)
      CALL GCRSG( 1)
      R=0.2
      CALL MAKER(R,ROUNDX,ROUNDY)
      CALL GPL(37,ROUNDX, ROUNDY)
      DO 20 I=1,6
      R=R+0.1
      CALL MAKER(R,ROUNDX,ROUNDY)
      CALL GPL(37,ROUNDX, ROUNDY)
 20   CONTINUE
      CALL GCLSG
      CALL GCRSG( 2)
      CALL NUMBER(0.0,0.2999,30.0)
      CALL NUMBER(0.2999,0.5999,15.0)
      CALL NUMBER(0.5999,0.7999,4.0)
      CALL GCLSG
      CALL GCRSG( 5)
      CALL GPL(38,CURX, CURY)
      DO 30 I=1,19
      CURX(I)=CURX(2*I)
      CURY(I)=CURY(2*I)
 30   CONTINUE
      CALL GPM(19,CURX, CURY)
      CALL GCLSG
      CALL GCRSG( 3)
      CALL GSCHSP( 0.15)
      CALL GSCHH( 0.09)
      CALL GTX(0.6,2.6,'A POLAR PLOT WITH GRID')
      CALL GCLSG
      CALL GCRSG(12)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 3,GTRAN1)
      CALL GEVTM(1.5,1.5,0.0,0.0, 0    ,0.5,0.7,0,GTRAN1)
      CALL GINSG( 5,GTRAN1)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 1,GTRAN1)
      CALL GCLSG
      CALL GRENSG(12,10)
C
C     create lower-half image
      CALL GSELNT( 2)
      CALL GCRSG( 4)
      CALL GSCHSP( 0.2 )
      CALL GSCHH( 0.1)
      CALL GTX(0.5,2.5,'XGRAF WITH A GRID')
      CALL GCLSG
      CALL GCRSG( 9)
      CALL GPL( 5,SIDEX, SIDEY)
      POINTX(1)=0.0
      POINTY(1)=0.0
      POINTX(2)=0.0
      POINTY(2)=2.0
      DO 50 I=1,10
      DO 60 J=1,2
60    POINTX(J)=POINTX(J)+0.3
      CALL GPL( 2,POINTX, POINTY)
50    CONTINUE
      POINTX(1)=0.0
      POINTY(1)=0.0
      POINTX(2)=3.0
      POINTY(2)=0.0
      DO 70 I=1,10
      DO 80 J=1,2
80    POINTY(J)=POINTY(J)+0.2
      CALL GPL( 2,POINTX, POINTY)
70    CONTINUE
      CALL GCLSG
      DO 90 I=1,100
      X= FLOAT(I)/ 100.
      CUR1X(I)=X*3.0
      CUR1Y(I)=EXP(-3.*X)*SIN(X*8.*3.14159)+1.0
      CUR2X(I)=X*3.0
      CUR2Y(I)=EXP(-3.*X)*COS(X*8.*3.14159)+1.0
90    CONTINUE
      CALL GCRSG( 6)
      CALL GPL(100,CUR1X, CUR1Y)
      CALL GPL(100,CUR2X, CUR2Y)
      CALL GCLSG
      CALL GCRSG( 7)
      CALL GSCHSP( 0.93)
      CALL GSCHH( 0.05)
      CALL GTX(-0.1,-0.2,'0.0 0.1 0.2 0.3 0.4 0.5 0.6')
      CALL GTX(1.96, -0.2,'0.7 0.8 0.9 1.0')
      CALL GSCHSP( 0.15)
      CALL GSCHH( 0.08)
      CALL GTX(0.5,-0.5,'X AXIS WITH GRAF TYPE')
      CALL GCLSG
      CALL GCRSG( 8)
      CALL GSCHSP( 0.8 )
      CALL GSCHH( 0.05)
      CALL GSCHUP(-1.0,0.0)
      CALL GTX(-0.1,-0.1,'-1.0 -0.6 -0.2 +0.2 +0.6 +1.0')
      CALL GSCHSP( 0.12)
      CALL GSCHH( 0.07 )
      CALL GTX(-0.32,0.2,'Y AXIS WITH GRAF TYPE')
      CALL GCLSG
      CALL GCRSG(12)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 9,GTRAN1)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 6,GTRAN1)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 4,GTRAN1)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 7,GTRAN1)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 8,GTRAN1)
      CALL GCLSG
      CALL GRENSG(12,11)
C
C     display the created images onto the workstation 2
      CALL GACWK( 2)
      CALL GCSGWK( 2,11)
      CALL GCSGWK( 2,10)
      CALL GCSGWK( 2, 2)
      CALL GDAWK( 2)
C
      CALL GDAWK(1)
      CALL GCLWK(1)
      CALL GCLWK( 2)
      CALL GCLWK( 3)
      CALL GCLKS
      STOP
      END
C
C
      SUBROUTINE NUMBER(R1,R2,DEGREE)
C     number the radius lines between two circles
      REAL PX( 2),PY( 2)
      REAL DEGREE
C
      ANGLE= 0.0
      J=360./DEGREE
      DO 10 I=1,J
      PX(1)=R1*COS(ANGLE)+1.5
      PY(1)=R1*SIN(ANGLE)+1.5
      PX(2)=R2*COS(ANGLE)+1.5
      PY(2)=R2*SIN(ANGLE)+1.5
      CALL GPL( 2,PX, PY)
      ANGLE=ANGLE+DEGREE*3.14159/180.
 10   CONTINUE
      RETURN
      END
```

A POLAR PLOT WITH GRID



XGRAF WITH A GRID

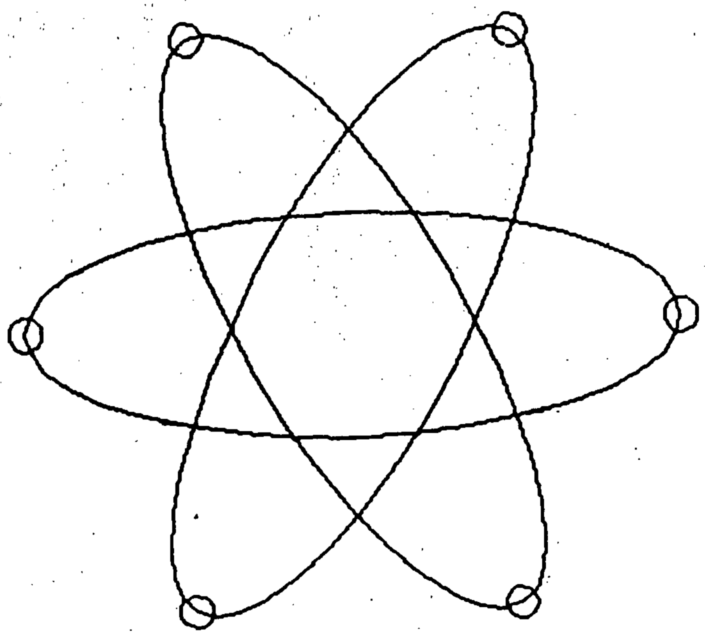


X AXIS WITH GRAF TYPE

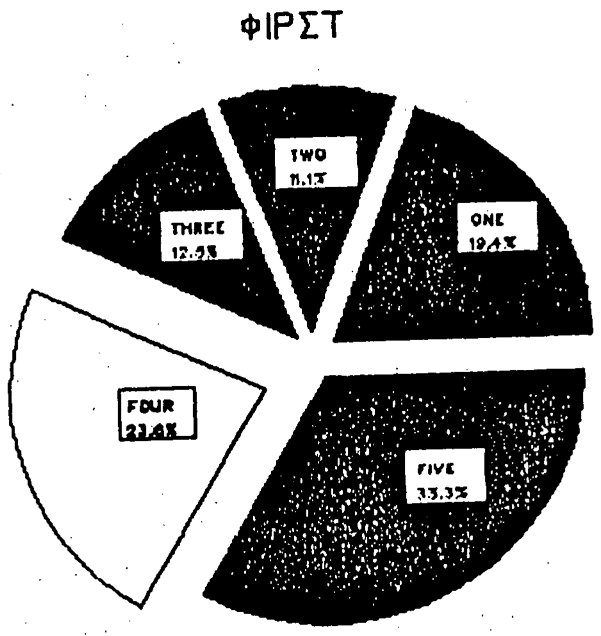
```
        {                                  }
        {     EXAMPLE 3 IN HL/GKS          }

        VECTOR ROUND(37), MARKS(2), MARK
        VECTOR PART1(18),PART2(18), PART3(18)
        VECTOR PART4(18), OUTP(18), SIDES(5), PSIDE(5),DT
        SEGMENT PICTURE, CENTER, PIE, PART
        SEGMENT TITLE, PATHS
        TRANS NT(3)
        WORKSTATION PLOTTER
        DATA MARKS/-1.0,0.0; 1.0,0.0/
        DATA SIDES/0.0,0.0; 0.3,0.0; 0.3,0.2; 0.0,0.2;
                   0.0,0.0/

        BEGIN {GKS}
        REST= 10.*3.14159/180.
        ANGLE= REST

        DO 10 I=1,37
           ROUND(I)= (COS(ANGLE), SIN(ANGLE))
10      ANGLE= ANGLE+REST

        NT(1)= <FROM:(-2.0,-5.5); (5.5,2.0)>
        NT(2)= <FROM:(0.0,0.0); (5.2,5.2)>
        NT(3)= <FROM:(-4.5,-3.0);(2.0,3.5)>

        { CREATE THE UPPER-LEFT PORTION OF THE IMAGE }
        {                                            }
        OPEN CENTER WITH NT(1)
             CENTER= POLYLINE ROUND( 1:37)
        CLOSE

        OPEN PICTURE
             PICTURE= POLYMARKER <TYPE:'CIRCLE'>
                                      MARKS(1:2)
                     + CENTER
        CLOSE

        PICTURE= PICTURE <SCALE: (1.5,0.5)>
                + PICTURE <SCALE: (1.5,0.5)
                           ROT: 60.0D >
                + PICTURE <SCALE: (1.5,0.5)
                           ROT: 120.0D >

        { CREATE THE LOWER-LEFT PORTION OF THE IMAGE }
        {                                            }
        OPEN PATHS WITH NT(2)
             MARK= (1.0,2.0)
             PATHS= TEXT <FONT: 'C_ITALIC',
                          HEIGHT: 0.1,
                          SPACE: 0.15>
                   'GKS' AT (1.1,2.0);
                         <PATH:'LEFT'>
```

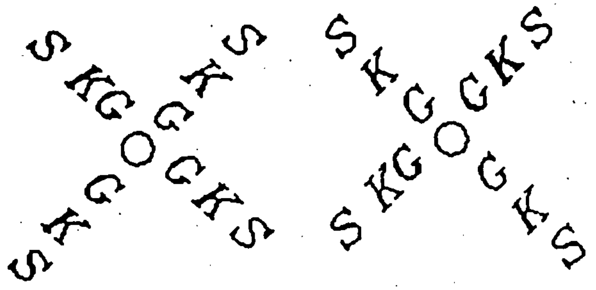
```
              'GKS' AT (0.9,2.0);
                    <PATH: 'UP'>
              'GKS' AT (1.0,2.1);
                    <PATH: 'DOWN'>
              'GKS' AT (1.0,1.9)
             + POLYMARKER <TYPE: 'CIRCLE'>
                    MARK
CLOSE

PATHS=PATHS <AT:(1.0,2.0)
              ROT: -45.0D>
      +PATHS <AT:(1.0,2.0)
              ROT: 45.0D
              SHIFT:(1.0,0.0) >

OPEN TITLE
     TITLE= TEXT <FONT: 'ENGLISH'
                  PATH: 'RIGHT',
                  HEIGHT: 0.15>
             'WELCOME TO' AT (0.65,1.02)
CLOSE

{ CREATE THE RIGHT PORTION OF THE IMAGE }
{                                       }
CALL GETP(-120.0,360.0,PART1)
CALL GETP(0.0,70.0,PART2)
CALL GETP(70.0,110.0,PART3)
CALL GETP(110.0,155.0,PART4)
CALL GETP(155.0,240.0,OUTP)

OPEN PART WITH NT(3)
     CALL BLOCK(198.0,SIDES,PSIDE,DT)
     PART= POLYLINE OUTP(1:18); PSIDE(1:5)
          + TEXT <FONT: 'S_ROMAN'
                  HEIGHT: 0.04,
                  SPACE: 0.15>
         'FOUR' AT(DT.XCOORD+0.04,DT.YCOORD+0.11);
         '23.6%' AT (DT.XCOORD+0.03,DT.YCOORD+0.01)
CLOSE

OPEN PIE
     PIE= FILLAREA <INSIDE:'SOLID'>
                   PART1(1:18);
                   PART2(1:18);
                   PART3(1:18);
                   PART4(1:18)
     CALL BLOCK(300.0,SIDES,PSIDE,DT)
     PIE= PIE + FILLAREA
               <COLOUR:'WHITE'> PSIDE(1:5)
          + TEXT 'FIVE' AT (DT.XCOORD+0.04,
                            DT.YCOORD+0.11);
               '33.3%' AT (DT.XCOORD+0.03,
                            DT.YCOORD+0.01)
     CALL BLOCK(35.0,SIDES,PSIDE,DT)
     PIE= PIE + FILLAREA PSIDE(1:5)
         + TEXT 'ONE' AT (DT.XCOORD+0.05,
                          DT.YCOORD+0.11);
                '19.4%' AT (DT.XCOORD+0.04,
                            DT.YCOORD+0.01)
     CALL BLOCK(100.0,SIDES,PSIDE,DT)
     PIE= PIE+ FILLAREA PSIDE(1:5)
         + TEXT 'TWO' AT (DT.XCOORD+0.05,
                          DT.YCOORD+0.11);
                '11.1%' AT (DT.XCOORD+0.04,
                            DT.YCOORD+0.01)
     CALL BLOCK(145.0,SIDES,PSIDE,DT)
     PIE= PIE+ FILLAREA PSIDE(1:5)
         + TEXT 'THREE' AT (DT.XCOORD+0.03,
                            DT.YCOORD+0.11);
                '12.5%' AT (DT.XCOORD+0.03,
                            DT.YCOORD+0.01);
                <HEIGHT: 0.15,
                 FONT: 'C_ITALIC',
                 SPACE: 0.2>
           'PIE CHARTS' AT (-0.6,1.7);
                <FONT: 'S_GREEK',
                 HEIGHT: 0.09,
                 SPACE: 0.20>
           'FIRST' AT (-0.25,1.3)
CLOSE
PIE= PIE+PART<SHIFT: (-0.1,-0.1)>

DISPLAY PIE ON PLOTTER
DISPLAY TITLE+PATHS ON PLOTTER
DISPLAY PICTURE ON PLOTTER
STOP
END

{ GET SOME PART OF A CIRCLE }
{                           }
SUBROUTINE GETP(A1,A2,POINT)
VECTOR POINT(18), DIST
REAL RADIUS

BEGIN
IF (A1.LT.0.0) THEN
A1= 360.+A1
ENDIF

IF (A2.LT.0.0) THEN
A2= 360.+A2
ENDIF

ANGLE= A1
R1=((A2-A1)/2.+A1)*3.14159/180.
DIST= (0.1*COS(R1), 0.1*SIN(R1))
POINT(1)= (DIST.XCOORD,DIST.YCOORD)
POINT(18)= POINT(1)
```

```
        ADDING= (A2-A1)/14.

        DO 10 I=1,15
           RADIUS= ANGLE*3.14159/180.
           POINT(I+1)= (COS(RADIUS)+DIST.XCOORD,
                        SIN(RADIUS)+DIST.YCOORD)
           ANGLE= ANGLE+ ADDING
10      CONTINUE

        RADIUS= A2*3.14159/180.
        POINT(17)= (COS(RADIUS)+DIST.XCOORD,
                    SIN(RADIUS)+DIST.YCOORD)
        RETURN
        END

        { GET A LABELLING BLOCK }
        {                       }
        SUBROUTINE BLOCK(A,VS,PS,VT)
        VECTOR VS(5), VT
        VECTOR PS(5)

        BEGIN
        A= A*3.14159/180.
        VT= (0.7*COS(A), 0.7*SIN(A))

        DO 10 I=1,5
           PS(I)= (VS.XCOORD(I)+VT.XCOORD,
                   VS.YCOORD(I)+VT.YCOORD)
 10     CONTINUE

        RETURN
        END
```



PIE CHARTS





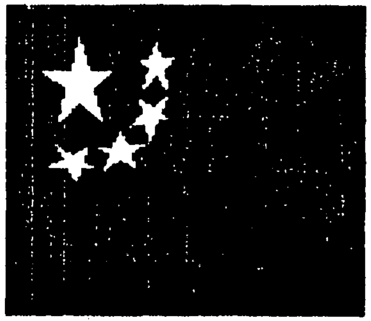
WELCOME TO

```
        {                                   }
        {       Example 4 in HL/GKS         }
        {                                   }

        SEGMENT STAR(2),STARS(2),FLAG(2),FLAGS(2),NAME(2),
                TITLE
        VECTOR POINT(11), SIDE1(5), DIST,OUT(5), IN(5),
               LEAF(26), SIDE2(5), PART1(5), PART2(5)
        TRANS NT(3)
        WORKSTATION PLOTTER, T4010
        DATA SIDE1/1.0,3.0; 4.0,3.0; 4.0,5.0; 1.0,5.0; 1.0,3.0/
        DATA OUT/1.0,2.0; 0.0,1.3; 0.41,0.2; 1.61,0.2; 1.9,1.3/,
             IN/0.8,1.35; 0.65,0.9; 1.0,0.65; 1.38,0.9;
                1.25,1.35/
        DATA SIDE2/0.0,0.0; 6.0,0.0; 6.0,3.0; 0.0,3.0; 0.0,0.0/
        DATA LEAF/3.0,2.5; 2.8,2.0; 2.6,2.2; 2.9,1.5; 2.25,1.75;
                  2.3,1.5; 1.9,1.4; 2.3,1.3; 2.15,1.05; 2.9,1.2;
                  2.5,0.8; 2.9,1.0; 2.9,0.2; 3.1,0.2; 3.1,1.0;
                  3.5,0.8; 3.1,1.2; 3.85,1.05; 3.7,1.3; 4.2,1.4;
                  3.7,1.5; 3.8,1.7; 3.1,1.5; 3.4,2.2; 3.2,2.0;
                  3.0,2.5/,
              PART1/0.0,0.0; 1.5,0.0; 1.5,3.0; 0.0,3.0; 0.0,0.0/

        BEGIN  { GKS initialisation }

        { set transformations }

        NT(1)= <FROM:(0.0,-4.0); (9.0,9.8)
                  TO:(0.5,0.0); (1.0,1.0)>
        NT(2)= <FROM:(0.0,-3.2); (18.0,17.8)
                  TO:(0.0,0.0); (0.5,1.0)>
        NT(3)= <FROM:(0.0,0.0); (1.0,0.2)
                  TO:(0.0,0.75); (1.0,0.95)>

        { create Chinese national flags onto the     }
        { right-half plane of the output image       }

        DO 10 I=1,5
           J=(2*I)-1
           K=2*I
           POINT(J)= OUT(I)
           POINT(K)= IN(I)
10      CONTINUE

        POINT(11)= POINT(1)

        { create the centre Chinese flag      }

        OPEN STAR(1) WITH NT(1)
             STAR(1)= FILLAREA <INSIDE: 'SOLID'
                                COLOUR: 'WHITE' >
                                POINT(1:11)
        CLOSE
```

```
STAR(1)= STAR(1) <AT:(1.0,1.0)
                  SCALE:(0.3,0.3)>
STAR(2)= STAR(1) <AT:(1.0,1.0)
                  SCALE:(0.54,0.54)>
STARS(1)= STAR(1) <AT:(1.0,1.0)
                   SCALE:(0.3,0.3)>
STARS(2)= STARS(1)<AT:(1.0,1.0)
                   SCALE:(0.5,0.5)>
STAR(1)= STAR(1) <SHIFT:(0.6,3.5)>

OPEN FLAG(1)
     FLAG(1)= FILLAREA <COLOUR:'BLACK'>
                          SIDE1(1:5)
             + TEXT <HEIGHT: 0.3,
                     FONT: 'C_ROMAN',
                     SPACE: 0.8>
              'CHINA' AT (1.4,2.2)
             + STAR(1)
CLOSE

FLAGS(1)= FLAG(1) <AT:(1.6,4.5)
                   SCALE:(0.3,0.3)>
R= 0.65
DIST= (R*COS(10.*3.14159/180.)+0.6,
       R*SIN(10.*3.14159/180.)+3.5)
STAR(2)= STAR(2) <SHIFT:(DIST.XCOORD,DIST.YCOORD)>
STAR(2)= STAR(2)+ STAR(2) <ROT:-36.0D
                           AT:(1.6,4.5)>
                + STAR(2) <AT:(1.6,4.5)
                           ROT:-72.0D>
                + STAR(2) <AT:(1.6,4.5)
                           ROT:-108.0D>
FLAG(1)= FLAG(1)+ STAR(2)

{ create the surrounding Chinese flags  }

R= 0.24
DIST= (R*COS(10.*3.14159/180.)+0.6,
       R*SIN(10.*3.14159/180.)+3.5)
STARS(2)= STARS(2) <SHIFT:(DIST.XCOORD,DIST.YCOORD)>
STARS(2)= STARS(2)+ STARS(2)<AT:(1.6,4.5)
                             ROT:-36.0D>
                  + STARS(2)<AT:(1.6,4.5)
                             ROT: -72.0D>
                  + STARS(2)<AT:(1.6,4.5)
                             ROT:-108.0D>
FLAGS(1)= FLAGS(1)+ STARS(2)
R=4.5
DIST=(R*COS(260.0*3.14159/180.)+1.5,
      R*SIN(260.0*3.14159/180.)+0.5)
FLAGS(1)= FLAGS(1) <SHIFT:(DIST.XCOORD,DIST.YCOORD)>
FLAGS(1)= FLAGS(1) <AT:(2.5,4.0)
                    ROT: 45.0D>
```

```
            +FLAGS(1) <AT:(2.5,4.0)
                       ROT: 80.0D>
            +FLAGS(1) <AT:(2.5,4.0)
                       ROT:10.0D>

       { create the title of the output image  }

       OPEN TITLE WITH NT(3)
            TITLE= TEXT <FONT: 'ENGLISH'
                          HEIGHT: 0.06,
                          SPACE: 0.04>
                    'FRIENDSHIP' AT (0.15,0.1);
                    'OF' AT (0.38,0.0)
       CLOSE

       { create Canadian national flags onto the   }
       { left-half plane of the output image       }

       DO 50 I=1,5
          PART2(I)= (PART1.XCOORD(I)+4.5, PART1.YCOORD(I))
50     CONTINUE

       { create the centre Canadian flag   }

       OPEN FLAG(2) WITH NT(2)
            FLAG(2)= POLYLINE SIDE2(1:5)
                    + FILLAREA <INSIDE:'HATCH'> LEAF(1:26);
                               <INSIDE:'SOLID'> PART1(1:5);
                      PART2(1:5)
       CLOSE

       OPEN NAME(1)
            NAME(1)= TEXT <FONT:'S_GREEK',
                           HEIGHT: 0.6,
                           SPACE: 0.3>
                     'CANADA' AT (0.0,0.0)
       CLOSE
       OPEN NAME(2)
            NAME(2)= TEXT <FONT:'C_4ITALIC',
                           HEIGHT: 0.10,
                           SPACE: 1.0 >
                  'CANADA' AT (0.0,0.0)
       CLOSE

       { create the surrounding Canadian flag  }

       FLAGS(2)= FLAG(2) <SCALE: (0.3,0.36)
                          SHIFT: (8.0,3.0)>
                + NAME(2) <SHIFT: (8.5,2.5)>
       FLAG(2)= FLAG(2) <SHIFT: (6.5,7.5)>
               + NAME(1) <SHIFT: (7.0,6.2)>
       FLAGS(2)= FLAGS(2) <ROT: -10.0D,
                            AT: (9.5,7.5) >
               + FLAGS(2) <ROT: -50.0D,
                    AT: (9.5,7.5) >
        + FLAGS(2) <ROT: -90.0D,
                    AT: (9.5,7.5) >

DISPLAY FLAG(2), FLAGS(2) ON PLOTTER
DISPLAY FLAG(1)+ FLAGS(1), TITLE ON PLOTTER
STOP
END
```

FRIENDSHIP OF



ΨANAΔA

CHINA

VI. CONCLUSION AND PROPOSALS

As an extended language, a number of syntax and semantics rules, based on the GKS system, are supplemented in the graphically oriented extension of HL/GKS. The two kinds of statements, the host language FORTRAN 77 statements and the graphical extension statements of HL/GKS, can be freely interspersed in an HL/GKS program, providing full graphical and nongraphical support. Some examples of interspersed statements are presented in Figure 45.

```
    REAL r1, r2
    VECTOR vg, vt(2)
    SEGMENT house
    DATA r1/1.3/, vt/0.0,0.0; 1.0,1.5/, r2/2.0/
    vg = (1.5, 2.8*sin(0.2))
    vg = (r1, r2)

    OPEN house
         {any FORTRAN statements}
         house = POLYLINE side(1:10)
         {any FORTRAN statemets}
         house = house +TEXT 'welcome to GKS' AT vg
         {any FORTRAN statements}
    CLOSE
    DO 10 i=1,15
       {any graphical statements}
       k= k + i
       vg = (vg.xcoord+i, vg.ycoord+k)
       house = house + house <SCALE: (r1,r2),
                                 AT: vg >
       {any FORTRAN statements}
10     CONTINUE
```

Figure 45 - Some Interposing Instances in HL/GKS

The precompiler is not case sensitive as to keywords and variables, i.e., upper and lower case characters may be used. The programming structures WHILE and REPEAT blocks and a free layout of statements have been introduced in HL/GKS. Moreover, blank lines and freely-placed comments within the braces "{" and

"}" are allowed. Modular programming is available with the possibility of precompiling parts of a program. More reasonable and understandable language patterns compared to the subroutine invocations of GKS render HL/GKS appealing for both learning and utilizing the system. The state restrictions of GKS and many system specifications are simplified in HL/GKS, they are basically controlled and generated by the precompiler.

In GKS, the segment transformation order is predefined and fixed to the order of scaling, rotation and translation. For any other order, it is the responsibility of the programmer to accumulate the transformations in the order desired one by one into an appropriate transformation matrix.

```
    CALL GEVTM(x0,y0,dx,dy,phi,fx,fy,sw,mout)
    CALL GACTM(mout,x0,y0,dx,dy,phi,fx,fy,sw,mall)
```

The declaration and setting of the evaluate and accumulate matrices for differently ordered transformations are also the programmer's responsibility. In contrast, a pattern of transformations is applied in HL/GKS instead of the matrix assignment of GKS. The keywords in a transformation pattern refer to the individual transformation steps and the order of the keywords is significant. The character D appended to the rotation angle means that it is measured by degrees instead of radians. A comparison of segment transformations in the order of translation, rotation, and scaling, in HL/GKS and GKS, is given in Figure 46.

The basic picture entity of GKS, the segment, is only single-level structured. No access is possible to the elements

```
house < AT: atv,
        SHIFT: shv,
        ROT: ra D,
        SCALE: scv >

 REAL tran1(2,3), tran2(2,3)
 r1 = ra *(3.14159/180.)
 CALL GEVTM(atvx,atvy,shvx,shvy,0.,1.,1.,0,tran1)
 CALL GACTM(tran1,0.,0.,0.,0.,r1,1.,1.,0,tran2)
 CALL GACTM(tran2,0.,0.,0.,0.,0.,scvx,scvy,0,tran1)
 CALL GSSGT(3,tran1)
```

Figure 46 - An Arbitrarily Ordered Segment Tansformation in Both Languages

of a compounded segment. A segment can not be updated after it has been created. In addition, most segment operations are restricted to the system state bounds. For example, the insertion segments into an open segment must occur in the state of "at most one segment open" with the additional condition that there are segments in WISS (workstation independent segment storage). In HL/GKS, with the data type SEGMENT supplied, a hierarchical modelling facility for segments has been provided. A segment can be superimposed or updated with any element after it has been created. Multiple modelling of a segment is available by allowing a reference to itself; segment redefinition is also possible by referring to other segments. Some instances of segment modelling are presented in Figure 47.

```
house =  house + door < AT:(4.0,5.2)>
               + window(1) <scale:(0.8,1.2)>
               + roof
house =  house - window(1) + window(2)
house =  building + tree
```

Figure 47 - Segment Modelling in HL/GKS

No state restrictions of GKS are imposed by the segment operations of HL/GKS except the state of "at most one segment open" which is provided by the statement pair of segment open and close.

The two attribute assignment modes of GKS, bundled and individual, are internally flagged and assigned to a bundle table only by the keyword TABLE or to individual by keywords in the attribute assignment. No restrictions have been placed on the assignment modes provided. The states of "GKS open", "GKS closed", "at least one workstation open", and "at least one workstation active" are all transparent to the HL/GKS programmer. The function restrictions on the states of GKS have been reduced by the clarified syntax and semantics of HL/GKS. Most specifications of GKS system are collected as attribute groups to be assigned to a related data type, thus providing the concept of functional performance instead of individual invocations of subroutines in GKS. The syntax design of HL/GKS keeps the form for its statements consistent with the host language FORTRAN 77, e.g., declaration, assignment, and input statements. The semantics of HL/GKS have been kept as close as possible to the definition of GKS. Most keywords used in HL/GKS are directly inferred from the corresponding definitions of the GKS document, providing a natural meaning for each. The workstation types of a configuration are predefined in the precompiler for proper reference of the workstations.

The viewing pipeline of GKS is reflected in definitions of HL/GKS. Multiple assignment of segments and workstations in a display statement is parsed to multiple output of segments onto several display workstations simultaneously. Output primitives are included in the output streams of HL/GKS, as for the GKS definition. The concept of multiple assignment is also applicable to primitive modelling, simplifying multiple primitive assignment with an identical primitive type.

The overloading of numerical parameter assignments of segments, transformations, and attributes of GKS has been removed by the use of name references of the parameters. Graphical parameters may appear in a subroutine formal parameter list, opening the graphical communications among main program and subroutines, which facilitates the graphical processing to each global and local function.

The main disadvantage of the HL/GKS implementation is due to the extra storage space required for the precompiler and the additional execution time required for the precompiling. The shortcoming will become less important with the development of a GKS chip and the popularity of GKS applications.

Stemming from the experience with the HL/GKS implementation, several suggestions and proposals for further development of HL/GKS are offered as follows:

. extending HL/GKS for three-dimensional graphics, either based on the functional mapping of the three-dimensional GKS ISO draft or by the development of new system routines to be included in its runtime library.

. supplementing the set of output primitives, e.g., by square and circle, using the same assignment rules as the primitives of HL/GKS. The parameters of a new primitive type can be assigned with the same syntax as for primitive attributes while the new primitive is named with an appropriate keyword. For example, the primitive circle may be specified as:

```
            CIRCLE left <CENTRE: (0.0,1.2),
                         RADIUS: 1.5       >
```

. adding more practical functions to the runtime library of HL/GKS, such as a comparison of two segments eqseg(f1,f2: segment), or test for presence of a vector in a segment member(v:vector; f:segment), etc., to facilitate implementation of graphical algorithms.

. extending the syntactic and semantic definitions of HL/GKS to other GKS levels, as for instance adding the input modes sample and event of GKS, to the GKS level 2C.

With the development of GKS chips and an increasing popularity of the GKS standard, the characteristics of the high-level graphics language HL/GKS will become attractive and significant, as it promises higher productivity of programmers and increased efficiency of GKS application programs.

REFERENCES

[Bart82] W. Barth, J. Dirnberger, and W. Purgathofer, "The High-Level Graphics Programming Language PASCAL/GRAPH," Computer and Graphics, Vol. 6, No. 3, 1982, pp. 109-119.

[Brac71] G. Bracchi and D. Ferrari, "A Language for Treating Geometric Patterns in a Two-Dimensional Space," Communications of the ACM, Vol. 14, No. 1, January 1971, pp. 26-32.

[Brie75] C.D. O'Brien and H.G. Bown, "IMAGE: A Language for the Interactive Manipulation of a Graphics Environment," Computer Graphics, ACM-Siggraph, June 1975, pp. 53-60.

[Dene75] E. Denert, G. Ernst, and H. Wetzel, "GRAPHEX68: Graphical Language Features in ALGOL 68," Computer and Graphics, Vol. 1, 1975, pp. 195-202.

[East77] C. Eastman and M. Henrion, "Glide: A Language for Design Information Systems," Computer and Graphics, 1977, pp. 24-33.

[Ende84] G. Enderle, K. Kansy and G. Pfaff, Computer Graphics Programming, GKS --- The Graphics Standard, Springer-Verlag, Berlin Heidelberg, 1984.

[Guib82] L.J. Guibas and J. Stolfi, "A Language for Bitmap Manipulation," ACM Transactions on Graphics, Vol. 1, No. 3, July 1982, pp. 191-214.

[Magn81] N. Magnenat-Thalmann and D. Thalmann, "A Graphical PASCAL Extension Based on Graphical Types," Software --- Practice and Experience, Vol. 11, 1981, pp.53-62.

[Mair84] S.G. Mair, UBC GKS, Computing Centre, The University of British Columbia, 1984.

[Ross82] R. Ross, LIG6 User's Manual, Department of Electrical Engineering, The University of British Columbia, 1982.

[Schr82] G.F. Schrack, G. Cheng, H. Leung and B. Yu, CWS-TD User's Manual, Department of Electrical Engineering, The University of British Columbia, April 1982.

[Shap75] L.G. Shapiro, "ESP^3: A High-level Graphics Language," Computer Graphics, Vol. 9, No. 1, 1975, pp. 70-77.

[Smit71] D.N. Smith, "GPL/I --- A PL/I Extension for Computer Graphics," Spring Joint Computer Conference, 1971, pp. 511-528.

BIBLIOGRAPHY

[Aho 77] A.V. Aho and J.H. Ullman, Principles of Compiler Design, Addison-Wesley, 1977.

[Aho 72] A.V. Aho and J.H. Ullman, The Theory of Parsing, Translation and Compiling, Prentice-Hall, 1972.

[Back79] R. Backhouse, Syntax of Programming Languages, Prentice-Hall, 1979.

[Berg76] S. Bergman and A. Kaufman, "BGRAF2: A Real-Time Graphics Language With Modular Objects and Implicit Dynamics," Computer Graphics ACM, Siggraph'76, Vol. 10, No. 2, 1976, pp. 133-138.

[Bono82] P.R. Bono, J.L. Encarnacao, F.R.A. Hopgood and P.J.W. ten Hagen, "GKS --- The First Graphics Standard," IEEE Computer Graphics and Applications, July 1982, pp. 9-22.

[Burk68] W.H. Burkhardt, "Metalanguage and Syntax Specification," Communications of the ACM, Vol. 8, No. 5, 1965, pp. 304-306.

[Clar81] D.R. Clark, Computers for Imagemaking, Pergamon Press, London, 1981.

[Enca81] J.L. Encarnacao, Ed., Eurographics '81, North-Holland, N.Y.,1981.

[Fole82] J.D. Foley and A. van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, 1982.

[Fole76] J.D. Foley, "Picture Naming and Modification: An Overview," Proceedings ACM Symposium on Graphic Languages, Computer Graphics, Vol. 10, NO. 1, 1976, pp. 49-53.

[Gilo74] W.K. Giloi and J. Encarnacao, "APLG - An APL based System for Interactive Computer Graphics," AFIPS Conference Proceedings, Vol. 43, 1974, pp. 521-528.

[Gilo78] W.K. Giloi, Interactive Computer Graphics, Prentice-Hall, Englewood Cliffs, N.J., 1978.

[Goos76] G. Goos and J. Hartmanis, Design and Implementation of Programming Languages, Lecture Notes in Computer Science 54, Ithaca, N.Y., 1976.

[Grif76] M. Griffiths, "LL(1) Grammars and Analysers, Compiler Construction - An Advanced Course," Lecture Notes in Computer Science 21, Springer-Verlag, 1977.

[Harr84] D. Harris, Computer Graphics and Applications, Chapman and Hall, N.Y., 1984.

[Hurw67] A. Hurwitz and J.P. Citron, "GRAF: Graphic Additions to FORTRAN," Spring Joint Computer Conference, 1967, pp. 553-557.

[Jone76] B. Jones, "An Extended ALGOL-60 for Shaded Computer Graphics," Proceedings ACM Symposium on Graphic Languages, Computer Graphics, Vol. 10, No. 1, 1976, pp. 18-23.

[Krig76] M.P. Kriger, "SUGER: A High-Level Programming Language for Geographical Analysis," Proceedings ACM Symposium on Graphic Languages, Computer Graphics, Vol. 10, No. 1, 1976, pp. 40-47.

[Lieb76] H. Lieberman, "The TV Turtle: A Logo Graphics System for Raster Displays," Proceedings ACM Symposium on Graphic Languages, Computer Graphics, Vol. 10, No. 1, 1976, pp. 66-72.

[Luci76] A.P. Lucido, "Software Systems For Computer Graphics," Computer, Vol. 9, No. 2, August 1976, pp. 23-32.

[Magn85] N. Magnenat-Thalmann, D. Thalmann and M.Fortin, "Miranim: An Extensible Director-Oriented System for the Animation of Realistic Images," IEEE Computer Graphics and Applications, March 1985, pp. 61-73.

[Mall82] W.R. Mallgren, "Formal Specification of Graphic Data Types," Transactions on Programming Languages and Systems, Vol. 4, No. 4, October 1982, pp. 687-710.

[Mell84] M.E. Mell and S.J. Noll, "Special Feature: A VLSI Support for GKS," IEEE Computer Graphics and Applications, August 1984, pp. 52-55.

[Newm71] W.M. Newman, "Display Procedures," Communications ACM, Vol. 14, No. 10, Oct. 1971, pp. 651-660.

[Newm73] W.M. Newman and R.F. Sproull, Principles of Interactive Computer Graphics, McGraw Hill, N.Y., 1973.

[Orga78] E.I. Organick, A.I. Forsythe and R.P. Plummer, Programming Language Structures, Academic Press, N.Y., 1978.

[Osla84] C.D. Osland, "Case study of GKS Development," Eurographics Tutorials'83, Springer-Verlag, 1984, pp. 264-289.

[Paga81] F.G. Pagan, Formal Specification of Programming Languages, Prentice-Hall, 1981.

[Pfis76] G.F. Pfister, "A High Level Language Extension for

Creating and Controlling Dynamic Pictures," Proceedings ACM Symposium on Graphic languages, Computer Graphics, Vol. 10, No. 1, 1976, pp. 1-9.

[Rubi84] R.V. Rubin and J.N. Pato, "MFE: A Syntax Directed Editor for Interaction Specification," Graphics Interface'84, 1984, pp. 265-269.

[Ruda84] M. Rudalics, "Dynamic Attributes Handling on a GKS Workstation," Graphics Interface'84, 1984, pp. 81-86.

[Scho84] J. Schonhut, "The Graphical Kernel System," Eurographics Tutorials'83, Springer-Verlag, 1984, pp. 233-257.

[Schr80] G.F. Schrack, "Modelling and Display Concepts in a High-Level Programming Language," Eurographics'80, North-Holland, 1980, pp. 225-236.

[Schr76] G.F. Schrack, "Design, Implementation and Experiences with a Higher-Level Graphics Language for Interactive Computer-Aided Design Purposes," Proceedings ACM Symposium on Graphic Languages, Computer Graphics, Vol. 10, No. 1, 1976, pp. 10-17.

[Shaw83] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch, "Descartes: A Programming-Language Approach to Interactive Display Interfaces," SIGPLAN Proceedings on Programming Language Issues in Software Systems, Vol. 18, No. 6, June 1983.

[Tekt84] Graphical Kernel System (GKS), TEK User Manual, Tektronix, Inc., May 1984.

[Tuck84] H.A. Tucker and K. Bo, Eds., Eurographics'84, North-Holland, N.Y., 1984.

[Vanw82] C.J. van Wyk, "A High-Level Language for Specifying Pictures," ACM Transactions on Graphics, Vol. 1, No. 2, April 1982, pp. 163-182.

[Wang76] P.S. Wang and W.I. Grosky, "A Language for Two-Dimensional Digital Picture Processing," Proceedings ACM Symposium on Graphic Languages, Computer Graphics, Vol. 10, No. 1, 1976, pp. 106-112.

[Wyvi77] B.L.M. Wyvill, "Pictures-68 MKI," Software --- Practice and Experience, Vol. 7, 1977, pp. 251-261.

APPENDIX A - SYNTAX IN BACKUS-NAUR FORM

```
<GKS Program>::= { <Main Block>
                  |<Sub Block> }_1

<Main Block>::= {<Declaration>}_0
                 BEGIN <GKSTMT>
                 STOP END

<Declaration>::= <Graph Types> <Identifier> <Array>
                        { , <Identifier> <Array> }_0
                 | DATA <Identifier>
                 / <Real List> { ; <Real List> }_0 /

<Graph Types>::=  VECTOR
                | SEGMENT
                | TRANS
                | WORKSTATION

<Identifier>::= <Letter> { <Letter> | <Digit> }^14

<Digit>::= 0 | <First Int>

<First Int>::= 1| 2| 3| 4| 5| 6| 7| 8| 9

<Letter>::= a| b| c| d| e| f| g| h| i| j| k| l| m| n|
            o| p| q| r| s| t| u| v| w| x| y| z|
            A| B| C| D| E| F| G| H| I| J| K| L| M| N|
            O| P| Q| R| S| T| U| V| W| X| Y| Z

<Array>::= ( <Int> ) | <Empty>

<Int>::= <Sign> <First Int> { <Digit> }_0

<Sign>::= +| -| <Empty>

<Real>::= <Sign> <Digit> . {<Digit>}_1

<Real List>::= <Real> , <Real>

<GKSTMT>::= { <Seg Create>
            | <Seg Operate1>
            | <Read STMT>
            | <Seg Operate2>
            | <Seg Operate>
            | <Message>
            | <Clear Works>
            | <Redraw Works> }_1

<Seg Create>::= OPEN <Identifier> <Array>
                     <Norm Trans>
               { { ABOVE
```

```
                   |BELOW }
                  <Identifier> <Array>
                 |<Empty>  }
                { <Seg Operate>
                | <Seg Operate1>
                | <Redraw Works>
                | <Message> }₁
                  CLOSE

<Norm Trans>::= WITH { <Identifier> <Array>
                     | <Norms>    }
                | <Empty>

<Norms>::= < { FROM : <Range> ; <Range>
             | TO : <Range> ; <Range>
             | ,    }₁  >

<Range>::= <Identifier> <Array>
         | ( <Expression> , <Expression> )

<Seg Operate>::= CALL <Sub Call>
               | WHILE ( <Boolean Exp> ) DO
                 <GKSTMT>
                 ENDWHILE
               | REPEAT   <GKSTMT>
                 UNTIL ( <Boolean Exp> )
               | CLIPPING ON
               | CLIPPING OFF
               | <Idnetifier> <Array>
                   =  { <Norms>
                      | ( <Expression> , <Expression> )
                      | <Expression>  }

<Terms>::= <Term> | <Primitives>

<Expression>::= <Terms> { { +| - } <Terms> }₀

<Term>::= <Factor> { { *| /| . } <Factor> }₀

<Factor>::= <Int>
          | <Real>
          | <Identifier> <Array> <Seg Trans>
          | ( <Expression> )

<Seg Trans>::= < { AT: <Range>
                 | SHIFT: <Range>
                 | ROT: { <Real> { D| <Empty> }
                 |      | <Identifier> }
                 | SCALE: <Range>
                 | ,     }₁  >
               | <Empty>

<Int List>::= <Int> : <Int>
```

```
<Primitives>::= POLYLINE <Poly Attri>
                     <Identifier> ( <Int List> )
                 { ; <Poly Attri>
                     <Identifier> ( <Int List> ) }o
               | POLYMARKER <Poly Attri>
                            <Identifier> { ( <Int List> )
                                         | <Empty> }
                  { ; <Poly Attri>
                      <Identifier> { ( <Int List> )
                                   | <Empty>  } }o
                | TEXT <Text Attri> <String>
                            AT <Range>
                       { ; <Text Attri> <String>
                            AT <Range>   }o
                 | FILLAREA <Filla Attri>
                             <Identifier> ( <Int List> )
                        { ; <Filla Attri>
                            <Identifier> ( <Int List> ) }o
                 | CELLARRAY <Cella Attri>
                              <Identifier> ( <Int> ,
                                             <INT>    )
                       { ; <Cella Attri>
                            <Identifier> ( <Int> ,
                                           <INT>   )  }o

<Type>::= TYPE : { <String>
                 | <Idnentifier> }

<Colour>::= COLOUR : { <String>
                     | <Identifier> }

<Poly Attri>::= < { <Type>
                  | <Colour>
                  | TABLE : { <Int>
                            |<Identifier>}
                  | ,  }1 >
                 | <Empty>

<Text Attri>::= < { FONT : { <String>
                           | <Identifier> }
                  | PREC : { <String>
                           | <Identifier> }
                  | EXP : { <Real>
                          | <Identifier> }
                  | SPACE : { <Real>
                            | <Identifier> }
                  | HEIGHT : { <Real>
                             | <Identifier> }
                  | UPVEC : ( <Expression> ,
                              <Expression> )
                  | PATH : { <String>
                           | <Identifier> }
                  | ALIGN : { <String>
                            | <Identifier> }
                 | <Colour>
                 | TABLE : { <Int>
                 |           |<Identifier>}
                 | ,  }1   >
               | <Empty>

<Filla Attri>::= < { INSIDE : { <String>
                                | <Identifier> }
                         { = { PSIZE: ( <Expression> ,
                                        <Expression> )
                             | PREF: ( <Expression> ,
                                        <Expression> )
                                   }1   }0
                   | STYLE : { <String>
                             | <Identifier> }
                   | <Colour>
                   | TABLE : { <Int>
                             |<Identifier>}
                   | ,    }1     >
                 | <Empty>

<Cella Attri>::= < { RANGE : <Range> ; <Range>
                   | MAXROW : <Int>
                   | ,    }1   >

<String>::= ' { <Character> }1 '

<Sub Call>::=  <Identifier>
             { ( { <Identifier> <Array>
                 | <Int>
                 | <Real>  }
               { , { <Identifier> <Array>
                   | <Int>
                   | <Real> }  }0     )
               | <Empty>  }

<Seg Operate1>::= RENAME <Identifier> <Array>
                           <Seg Attri>
                    AS <Identifier> <Array>
                | DELETE <Identifier> <Array>
                      { , <Identifier> <Array> }0
                  { FROM <Identifier> <Array>
                      { , <Identifier> <Array> }0
                  | <Empty>   }

<Redraw Works>::= REDRAW <Identifier> <Array>
                    { , <Identifier> <Array>  }0

<Message>::= MESSAGE <String>
                ON  <Identifier> <Array>
                { , <Identifier> <Array> }0

<Seg Attri>::= <  { VISIBLE
                  | INVISIBLE
```

```
                    | HIGHLIGHT
                    | NORMAL
                    | ,    }1     >
                  | <Empty>

<Read STMT>::= READ ( { <Identifier> <Array>
                      | <Int>       }   ,
                        <Identifier>   )
                 <Identifier> { ( { <Int List>
                                  | <Identifier> } )
                              | <Empty>    }
              { , <Identifier> { ( { <Int List>
                                   | <Identifier> } )
                               | <Empty>   }  }0

<Seg Operate2>::= SEND <Identifier> <Array>
                         <Seg Attri>
                    { , <Identifier> <Array>
                         <Seg Attri>        }0
                   TO <Identifier> <Array>
                    { , <Identifier> <Array> }0
                  | DISPLAY { <Identifier> <Array>
                               <Seg Attri>
                             | <Primitives>   }
                        { { ,| + } { <Identifier> <Array>
                                       <Seg Attri>
                                     | <Primitives> } }0
                     ON <Identifier> <Array> <Norm Trans>
                         { , <Identifier> <Array>
                             <Norm Trans>   }0

<Clear Works>::= ERASE <Identifier> <Array>
                   { , <Identifier> <Array>   }0

<Sub Block>::= { SUBROUTINE
               | <Types> FUNCTION }
               <Identifier> { ( <Idnetifier> { : OPEN
                                               |<Empty> }
                    { , <Identifier> { : OPEN
                                      |<Empty>} }0
                                  )
                            | <Empty>   }
                 { <Declaration> }0
                   BEGIN  <GKSTMT>
                   RETURN  END

<Types>::= CHARACTER { * <Int>
                     | <Empty>   }
         | COMPLEX
         | DOUBLE PRECISION
         | INTEGER
         | LOGICAL
         | REAL
```

APPENDIX B - RESERVED WORDS AND SYMBOLS

For HL/GKS, as an extended language, there are graphical symbols and reserved keywords which are part of the syntax definition. The complete set are shown in the two tables below.

Graphical Symbols

(	:	,
)	*	/
;	+	-
=	.	<
>		

Reserved Keywords

BEGIN	OPEN	VECTOR
SEGMENT	TRANS	WORKSTATION
CLOSE	RENAME	AS
DELETE	FROM	SEND
TO	DISPLAY	ON
CLIPPING	OFF	DO
WHILE	REPEAT	ENDWHILE
UNTIL	WITH	MESSAGE
ABOVE	BELOW	ERASE
REDRAW	AT	SHIFT
ROT	D	SCALE
POLYLINE	TEXT	POLYMARKER
FILLAREA	CELLARRAY	VISIBLE
INVISIBLE	NORMAL	HIGHLIGHT
COLOUR	TYPE	TABLE
FONT	PREC	EXP
SPACE	HEIGHT	UPVEC
PATH	ALIGN	INSIDE
PSIZE	PREF	STYLE
RANGE	MAXROW	

APPENDIX C - EXECUTION INSTRUCTIONS

As for the installation of GKS FORTRAN binding in UBC, all workstation outputs are assigned to individual output metafiles, which are then translated into the graphical file of each terminal by post-processors. The name of each output metafile is the first six characters of the translated workstation name, beginning with the minus sign "-", as a temporary output file. For instance, the metafile name of the workstation screen will be the name -SCREEN.

There are three metafile post-processors (for the Tektronix 4027, Printronix printer, and plotfile) available to view the metafile output from GKS. A detailed description of metafile post-processors is provided in the write-up UBC GKS[Mair84].

HL/GKS programs can be interactively executed by built-in macros. An example of HL/GKS interactive execution is shown below where the lower-case lines represent the prompts of a terminal while the upper-case ones are the input strings of a user.

```
$HLGKS
 go1 $r HL:GKS
 go2 $r FORTRANVS
 go3 $r -fort+GKS:GKS
^give the step name. GO1
^give the name of your file. TEST
 next step is go2.
^do you want to continue? Yes
 next step is go3.
^do you want to continue? YES
$
```

Arbitary entrance to the execution steps can be selected by the input of GO1, Go2, or Go3 respectively. Freedom of continuous processing is provided with the inquiry prompt (do you want to continue?) which can be replied to with Y, YES, OK, N, or NO. Any error in typing causes a return with some warning information provided. An example of single execution of step GO2 is given below.

```
$HLGKS
 go1 $r HL:GKS
 go2 $r FORTRANVS
 go3 $r -fort+GKS:GKS
^give the step name. GO2
 next step is go3.
^do you want to continue? NO
$
```

The running instructions of the post-processors are:

```
     (translation to Tektronix)
$run GKS:Metint427 scards=metafile,
     (translation to printronix printer)
$run GKS:Metintptx scards=meatfile,
     (translation to plotfile)
$run GKS:Metintplt scards=metafile.
```

For detailed running steps, refer to the write-up of UBC GKS[Mair84].

APPENDIX D - A SAMPLE OF GKS OBJECT CODE

```
C  This is a sample of GKS codes produced by the HL/GKS
C  preprocessor
C
      REAL GV1X( 5),GV1Y( 5)
      REAL GV2X( 5),GV2Y( 5)
      REAL GV3X( 5),GV3Y( 5)
      REAL GV4X( 7),GV4Y( 7)
      REAL GVX, GVY
      CHARACTER*4 GWF1
      CHARACTER*4 GWF2
      REAL GTRAN1(2,3), GTRAN2(2,3)
      INTEGER GPOLYL(13),GPOLYM(13)
      INTEGER GTEXT(13),GFILL(13),GG(13)
      CHARACTER*4 GF
      REAL GAX(100),GAY(100)
      INTEGER GT,GN,GSTAT
      DATA GV1X/ 0   , 0.50, 0.70, 0.20, 0   /
      DATA GV1Y/ 0   , 0   , 0.50, 0.50, 0   /
      DATA I/2/
      DATA GV2X/ 0.20, 0.70, 0.50, 0   , 0.20/
      DATA GV2Y/ 0   , 0   , 0.50, 0.50, 0   /
      DATA GV4X/ 0   , 5.00, 5.00, 2.50, 0   , 0
     +  , 0   /
      DATA GV4Y/ 0   , 0   , 3.20, 4.00, 3.20, 0
     +  , 0   /
C
C GKS-system initialization
      DATA GPOLYL/0,0,0,1,1,1,1,1,1,1,1,1,1/
      DATA GPOLYM/1,1,1,0,0,0,1,1,1,1,1,1,1/
      DATA GTEXT /1,1,1,1,1,1,0,0,0,0,1,1,1/
      DATA GFILL /1,1,1,1,1,1,1,1,1,1,0,0,0/
      DATA GG    /1,1,1,1,1,1,1,1,1,1,1,1,1/
      GF='-ERR'
      OPEN(UNIT=20,FILE=GF,STATUS='UNKNOWN')
      CALL GOPKS(20)
      CALL GOPWK(1,8,100)
      CALL GACWK(1)
      GWF1='-PLOTTE'
      OPEN(UNIT=14,IOSTAT=IOS,FILE=GWF1,STATUS='UNKNOWN')
      CALL GOPWK( 2,14,100000)
      GWF2='-SCREEN'
      OPEN(UNIT=15,IOSTAT=IOS,FILE=GWF2,STATUS='UNKNOWN')
      CALL GOPWK( 3,15,100000)
      CALL GMSG( 3,'WELCOME TO GKS')
      CALL GSWN( 1,0.0,5.0,0.0,7.0)
      CALL GSVP( 1,0.0,0.45,0.2,1.0)
      CALL GSWN( 2,1.5,3.5,0.0,7.0)
      CALL GSVP( 2,0.75,0.85,0.2,1.0)
      CALL GSELNT( 1)
      CALL GCRSG( 3)
CALL GSLN( 3)
CALL GPL( 5,GV1X, GV1Y)
CALL GCLSG
CALL GCRSG( 5)
GV3X(1)=0.0
GV3Y(1)=0.0
GV3X(2)=1.0
GV3Y(2)=0.0
GV3X(3)=1.0
GV3Y(3)=1.5
GV3X(4)=0.0
GV3Y(4)=1.5
GV3X(5)=0.0
GV3Y(5)=0.0
GVX=0.8
GVY=0.7
CALL GSLN( 1)
CALL GPL( 5,GV3X, GV3Y)
CALL GSMK( 4)
CALL GPM(1,GVX, GVY)
CALL GCLSG
CALL GCRSG( 1)
CALL GPL( 6,GV4X, GV4Y)
CALL GCLSG
CALL GCRSG( 7)
CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
CALL GINSG( 1,GTRAN1)
CALL GEVTM(0.,0.,2.0,0.0, 0    ,1.0,1.0,0,GTRAN1)
CALL GINSG( 5+1-1,GTRAN1)
CALL GEVTM(0.,0.,0.6,2.0, 0    ,1.0,1.0,0,GTRAN1)
CALL GINSG( 3+1-1,GTRAN1)
CALL GEVTM(0.,0.,3.7,2.0, 0    ,1.0,1.0,0,GTRAN1)
CALL GINSG( 3+1-1,GTRAN1)
CALL GCLSG
CALL GDSG( 1)
CALL GRENSG( 7, 1)
CALL GSELNT( 2)
CALL GCRSG( 4)
CALL GSLN( 3)
CALL GPL( 5,GV2X, GV2Y)
CALL GCLSG
CALL GCRSG( 6)
CALL GSLN( 1)
CALL GPL( 5,GV3X, GV3Y)
CALL GSMK( 4)
CALL GPM(1,GVX, GVY)
CALL GCLSG
CALL GCRSG( 2)
GV4X(4)=3.0
GV4Y(4)=3.7
GV4X(5)=2.0
GV4Y(5)=3.7
GV4X(6)=0.0
GV4Y(6)=3.2
```

```
      CALL GPL( 7,GV4X, GV4Y)
      CALL GSCHSP( 1.5)
      CALL GSCHH( 0.10)
      CALL GTX(1.0,5.0,'WELCOME TO GKS')
      CALL GCLSG
      CALL GCRSG( 7)
      CALL GEVTM(0.,0.,0.,0.,0.,1.,1.,0,GTRAN1)
      CALL GINSG( 2,GTRAN1)
      CALL GEVTM(0.,0.,2.0,0.0, 0    ,1.0,1.0,0,GTRAN1)
      CALL GINSG( 5+2-1,GTRAN1)
      CALL GEVTM(0.,0.,0.8,2.0, 0    ,1.0,1.0,0,GTRAN1)
      CALL GINSG( 3+2-1,GTRAN1)
      CALL GEVTM(0.,0.,3.7,2.0, 0    ,1.0,1.0,0,GTRAN1)
      CALL GINSG( 3+2-1,GTRAN1)
      CALL GCLSG
      CALL GDSG( 2)
      CALL GRENSG( 7, 2)
      CALL GACWK( 2)
      CALL GSHLIT( 1+1-1,1)
      CALL GCSGWK( 2, 1+1-1)
      CALL GDAWK( 2)
      CALL GACWK( 2)
      CALL GSHLIT( 1+2-1,0)
      CALL GCSGWK( 2, 1+2-1)
      CALL GDAWK( 2)
      CALL GMSG( 3,'GOODBYE')
C
C GKS-SYSTEM CLOSE
      CALL GDAWK(1)
      CALL GCLWK(1)
      CALL GCLWK( 2)
      CALL GCLWK( 3)
      CALL GCLKS
      STOP
      END
```