A GRAPHICAL EXTENSION FOR PASCAL BASED ON THE GRAPHICAL KERNEL SYSTEM

By

CYNTHIA LOUISE STARR

B.S., University of Kentucky, 1975

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

December 1987

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

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

Date December 31, 1987

Abstract

The Graphical Kernel System (GKS), the first international standard in the area of computer graphics, was adopted by the International Standards Organization in 1985. The United Kingdom, France, Germany and the United States have also adopted GKS as a national standard. This thesis examines the feasibility of developing a high-level graphical extension to a general-purpose programming language based on the GKS standard.

Because GKS was designed as a subroutine system, programming with it is awkward. The subroutine call provides a low-level mechanism for accessing the graphical capabilities standardized by GKS.

EZ/GKS is a high-level graphical extension to the Pascal/VS language implementing the functionality found in GKS level 2A. The level of abstraction for graphics programming is elevated in EZ/GKS through the use of abstract graphical data types. Operations on graphical data types are provided by structured graphical assignments, high-level graphical statements, graphical expressions and system-defined functions. Complex user-defined data types may be constructed from any of the predefined graphical data types in the usual manner provided by Pascal.

No major syntactic or semantic difficulties were encountered during the design and implementation of EZ/GKS. Thus, it appears that the GKS standard can indeed be elevated successfully to a high-level graphical extension of a general-purpose programming language.

Table of Contents

List of Figures

Acknowledgment

I express my appreciation to Dr. G. F. Schrack for his unfailing guidance and encouragement throughout the course of this project. Thanks also to Dr. R. Woodham for his careful reading of this document.

I thank Tektronics for their donation to the University of Plot-10 GKS. Their contribution allowed work on the GKS Pascal Binding to begin.

I am grateful to the Computer Science Department for their financial support.

I thank Sandra Litchfield, Reeta Tyagi and the staff of Discovery Day Care for their excellent child care. Without their help, this work would never have been completed.

I give special thanks to my husband, Alec Hoon, for patiently waiting more than three years for me to complete this work. I also express my appreciation to Alec for his help proof-reading the numerous draft versions of this thesis.

Chapter 1

Introduction

1.1 The Problem

The Graphical Kernel System (GKS), the first international standard in the area of computer graphics, was adopted by the International Standards Organization (ISO) in 1985 [ISO85a]. The United Kingdom, France, Germany and the United States have also adopted GKS as a national standard. This thesis examines the feasibility of developing a high-level graphical extension to the Pascal/VS language based on the GKS standard.

A system to support graphics programming may be developed by designing a new language or by enhancing the facilities of an existing host language. Developing a new language allows unlimited creative freedom in the language design. The development of an extensive compiler is necessary, however, resulting in a significant lead time for implementation. The non-geometric and arithmetic capabilities provided by a new language usually are not sufficient to support customary programming tasks. In addition, high training costs are incurred in a production environment educating programmers in the use of the language.

When a graphics programming language is based on an existing host language, many of the preceding problems are avoided. Although the language design of the graphical extension is restricted by the characteristics of the host language chosen, significantly less time and effort are required for implementation. All non-geometric and arithmetic functions are automatically inherited from the

host language. The training costs are limited to the costs of teaching experienced programmers the specifics of the graphical extension.

A graphics system based on a host language may be designed either as a subroutine system or as a language extension. With a subroutine system, routines providing graphical capabilities are added to the run time library, allowing the system to be easily extended or transported. No separate pre-compilation is required of graphical application programs. Most programming errors however can only be detected at run time, resulting in a longer lead time necessary for the development of graphical application programs due to the extensive testing required. In practice, subroutine systems are difficult to use and place a large burden on the programmer.

The GKS graphics standard serves both as a guideline for migrating graphical functions into hardware and also as a standardized, device independent means for accessing these graphical capabilities through software (Figure 1). GKS was designed as a subroutine system. Although GKS is easily extended and very

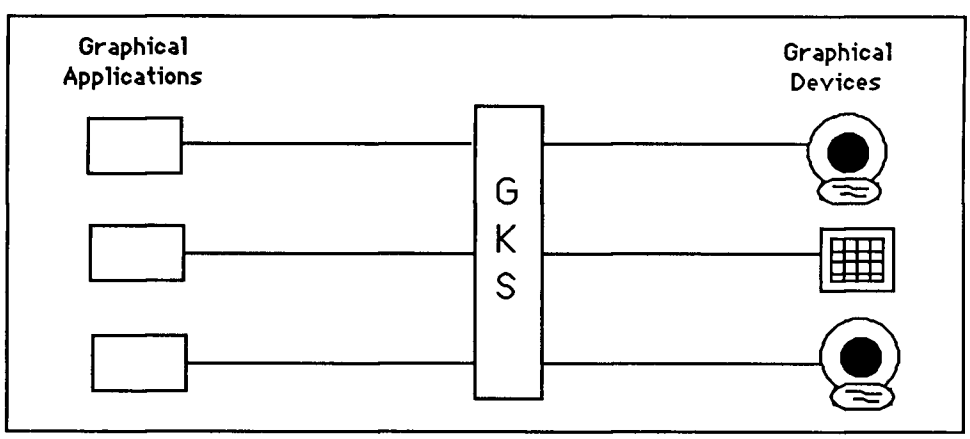


Figure 1 - The Role of GKS

portable, programming with it can be awkward. The subroutine calls with their parameter lists are difficult to remember. The resulting code is hard to understand and maintain. The standard subroutine call provides a low-level mechanism for accessing the graphical capabilities standardized by GKS.

An alternative method by which a host language may be extended is by "the definition of additional data types, expressions and statements not . . . found in the original host language." [GiEn72]. Implementation of a graphical extension requires the development of a pre-compiler or compiler extension. A pre-compilation or extra compilation time is necessary during the development of a graphical application program. Many programming errors can be detected during the pre-compilation phase, however, reducing the testing required for application programs.

The most significant benefit of a language extension is its ease of use. Given a clean and intuitive language syntax, the graphical extension to a standard programming language can increase the productivity of graphics programmers while reducing training costs. The lead time necessary for application program development is thereby reduced. The readable source code generated by using a high level graphical extension can also cut the cost of maintenance for application programs running in a production environment.

Pascal was selected as the host language for this high-level graphical extension based on wide acceptance in the academic community and on characteristics of the language itself. Pascal is an excellent language on which to base a graphical extension. The strong typing of Pascal aids in error detection while allowing the programmer to define new and more complex graphical data types. The use of a

block structure clearly depicts the scope of commands. The dynamic storage allocation mechanism allows data structures to expand as needed throughout program execution. All of these capabilities assisted with the design and implementation of EZ/GKS.

1.2 Goals

EZ/GKS is a high-level graphical extension to Pascal which provides access to the GKS subroutine system via graphical data types and high-level graphical statements embedded within an ordinary Pascal program. The major goal of EZ/GKS is to provide a high-level language support for graphics programming, easing the burden imposed on the programmer by the GKS subroutine system. To achieve this goal, the Pascal language has been extended by adding graphical data types and high-level graphical statements not originally found in the host language (Figure 2). The graphical extension provides support for both individual application programs and for the application-oriented layer. The application-oriented layer may provide modeling, charting or windowing

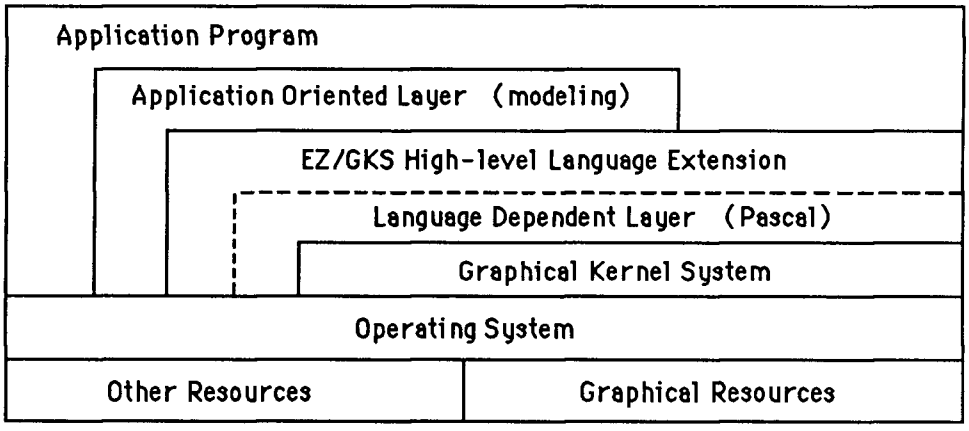


Figure 2 - Layer Model of GKS Incorporating EZ/GKS

sub-systems specifically useful to certain application programs. As in GKS, all application specific functions have been excluded from EZ/GKS, resulting in a general purpose graphical extension.

EZ/GKS is a graphical extension to the Pascal/VS language implementing the functionality found in level 2a of GKS. Level 2a includes basic segmentation with full output and Workstation Independent Segment Storage (WISS), but excludes all forms of graphical input. The language design provides easy-to-use graphical statements integrated with ordinary Pascal statements. Much of the syntax of the graphical statements is consistent with that of the regular Pascal language, allowing a natural representation familiar to Pascal programmers. Many concepts from Pascal are applied in the context of graphical operations, including records, arrays, linked lists, sets and assignment statements. Due to time restrictions, the inquiry, error and metafile functions have been excluded from the scope of this project.

The seven major goals of the EZ/GKS language design are to:

1. Elevate the GKS subroutine system to a high-level extension of the Pascal/VS language in order to simplify graphics programming using GKS.

2. Provide high-level language statements useful in graphical applications, allowing the programmer to focus on concepts relating to the application rather than on the details of graphics programming.

3. Maintain complete compatibility with the concepts of GKS.

4. Maintain the full functionality of GKS by clearly defining the relationships between EZ/GKS and the GKS subroutine system, allowing both to be mixed compatibly within the same application program.

5. Assist the graphics programmer in gradually learning the concepts of GKS, lowering training costs for graphics programmers.

6. Assist programmers in producing successful graphics programs, reducing the lead time necessary to implement graphical applications.

7. Produce clear graphics programs which are easy to read, understand and maintain.

These goals provided the basis for the decisions which governed the design of the EZ/GKS graphical extension.

1.3 Contributions

The benefits derived from the use of high-level programming languages are no longer disputed. High-level languages such as Fortran, Cobol, Pascal and PL/1 have become the norm for developing non-graphical application programs. The utility of a graphical extension to a general purpose programming language was recognized in 1968 when David Smith began the design for GPL/I [Smit71]. Since then many researchers have studied the use of high-level graphical extensions to general purpose programming languages [McLe78]. Of these contributions, the GPL/I, MIRA and HL/GKS graphical extensions have been selected for further discussion in the following sections.

1.3.1 GPL/I

GPL/I was designed as a graphical extension to the PL/I programming language. The design of the language was machine and device independent, defining graphical output conceptually rather than in hardware specific commands. To accomplish this goal, GPL/I proposed the use of graphical data types to assist with the manipulation of graphical data. Operations were provided on each data type in the form of function calls (Figure 3).

The four graphical data types provided by GPL/I were VECTOR, IMAGE, INTERRUPT and GRAPHIC. A variable of data type VECTOR described a line radiating from the origin and was represented by a two-dimensional or three-dimensional point. The IMAGE data type contained a combination of pictorial and attribute values which were derived from VECTOR variables, text and the result of IMAGE functions. An IMAGE variable, which could be any of PL/I's storage classes, was either structured or sequential. Although the internal structure of an IMAGE variable could be quite complex, all details of the structure were hidden from the

VECTOR Data Type	IMAGE Data Type
magnitude vector product scalar product addition subtraction	inclusion connection positioning scaling rotation

Figure 3 - Example of Graphical Data Types in GPL/I with Related Operations

user. The INTERRUPT data type was provided to control interrupts for graphical input. Finally, the GRAPHIC data type allowed three types of graphical files to be declared: DISPLAY, DESIGN and STORAGE.

Variables of these data types could be defined and manipulated by name in assignment statements and expressions. A number of PL/I statements, such as the OPEN, ON, SIGNAL and PUT, were extended to handle the graphical options permitted by the language extension. Additional high-level statement types included in GPL/I were the TAKE, ERASE and ANIMATE statements. Although the use of graphical data types in GPL/I was visionary, the high-level statement types provided for graphical manipulation were minimal.

1.3.2 MIRA

MIRA, a graphical extension to the Pascal language, also provided the use of data types and high-level statements to assist with graphics programming [MaTh81]. MIRA provided two predefined graphical data types similar to the ones in GPL/I. In addition, facilities in MIRA permitted the programmer to construct hierarchical, user-defined graphical data types.

As with GPL/I, MIRA provided a VECTOR data type and a vector arithmetic for operating on variables of the VECTOR type. The graphical statement CONNECT was provided for connecting multiple VECTOR variables.

A structured data type FIGURE was also introduced. The syntax for the declaration of a FIGURE type was similar to that of a Pascal procedure, including in the definition an identifier name, parameters, local declarations and instructions.

The graphical statement INCLUDE permitted the inclusion of other FIGURE types within the declaration of a complex figure, allowing simple FIGURE types to be used to build more complex data types in a hierarchical fashion.

Declaring variables to be of a particular FIGURE type did not create the graphical variable in MIRA. The high-level CREATE statement dynamically created a new FIGURE variable while the DELETE statement destroyed it. The DRAW statement output the FIGURE variable to a graphical device.

As in GPL/I, MIRA provided standard procedures and functions to operate on the graphical variables (Figure 4A & 4B). Several standard FIGURE types which were also provided by the system could be included in more complex FIGURE type declarations (Figure 4C).

The designers of GPL/I and MIRA were at a disadvantage, for the basic graphical programming functions were not standardized until 1985 when the GKS system was adopted by the ISO. The GKS standard defined a common terminology and a

A. Standard Procedures	B. Standard Functions	C. Standard Figure Types
symmetry translation rotation homothety union	angle intersection centre distance	segment line circle square triangle

Figure 4 - Example of Standard Procedures, Standard Functions and Standard Figure Types in MIRA

methodological framework for computer graphics programming.

1.3.3 HL/GKS

GKS was used as the foundation for a graphical extension to the Fortran-77 in HL/GKS [Sun86]. The HL/GKS language extension adopted many of the concepts of GKS, including workstations, segments and transformations. These concepts were represented as graphical data types in the graphical extension. A VECTOR data type was also provided.

Graphical variables could be defined as either individual variables or as single dimensional arrays. High-level graphical statements such as SEND, DISPLAY, MESSAGE, REDRAW and ERASE provided operations on the graphical variables.

HL/GKS made many of the details of GKS transparent to the user, including opening, closing, activating and deactivating workstations. Facilities were provided for updating and partial deletion of segments after closing. Explicit specification of attributes, transformations and output replaced the modal default provided by GKS.

The major disadvantage of HL/GKS is the inflexibility of the host language. The capability for defining structures of non-homogeneous elements is notably absent from Fortran-77, as are the block structure and dynamic storage allocation mechanisms found in most modern programming languages. The host language selected restricts the utility of the HL/GKS graphical extension.

1.4 Organization

Chapter 2 begins by examining the design constraints on the language that distinguish this work from that of its predecessors. The constraints include those imposed by the graphics subroutine system, GKS, and those imposed by the host language, Pascal. The advantages derived from GKS and Pascal are also discussed.

Chapter 3 describes the graphical data types provided by the EZ/GKS language. The distinction between individual and compound graphical data types is explained.

Chapter 4 presents the operations which characterize the graphical data types presented in Chapter 3. Operations include structured graphical assignments, graphical statements, graphical expressions and system-defined functions.

Chapter 5 discusses the development of the EZ/GKS pre-compiler and the translation techniques it utilizes. First, a simple translation is examined where a high-level EZ/GKS statement is directly translated into a GKS subroutine call. The use of the run time library and inquiry functions are described. Finally, methods for handling attribute values and high-level references to workstation tables are explained.

The appendices contain details of the language syntax, data types of attribute values and system-defined functions. Example programs are also provided written in EZ/GKS and GKS, producing identical output.

Chapter 2

Language Design

The goal of EZ/GKS is to provide access to the GKS subroutine system via graphical data types and high-level graphical statements integrated in an ordinary Pascal program. The input to the EZ/GKS pre-processor is a program containing predefined graphical data types and high-level graphical statements (Figure 5). A Pascal program containing the necessary GKS subroutine calls is generated. In this manner, the EZ/GKS language extension conceptually allows access to the facilities of GKS through the use of high-level graphical statements (Figure 6). The use of a mixture of high-level statements and GKS subroutine calls within the same application program is not precluded, thereby allowing an EZ/GKS program access to the full functionality of GKS.

The following sections examine how the language design is constrained by both the subroutine system whose procedures it must access and by the host language in which i t is imbedded. Finally, the benefits derived from the host language are

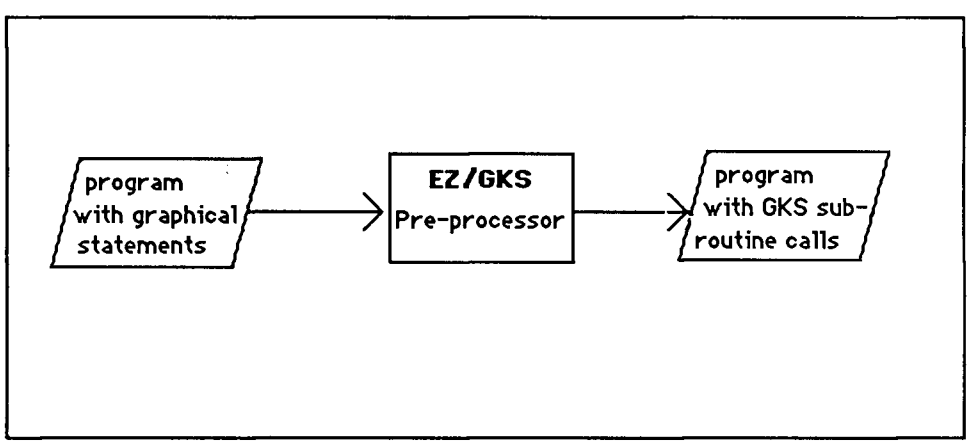


Figure 5 - Translation of an EZ/GKS Program

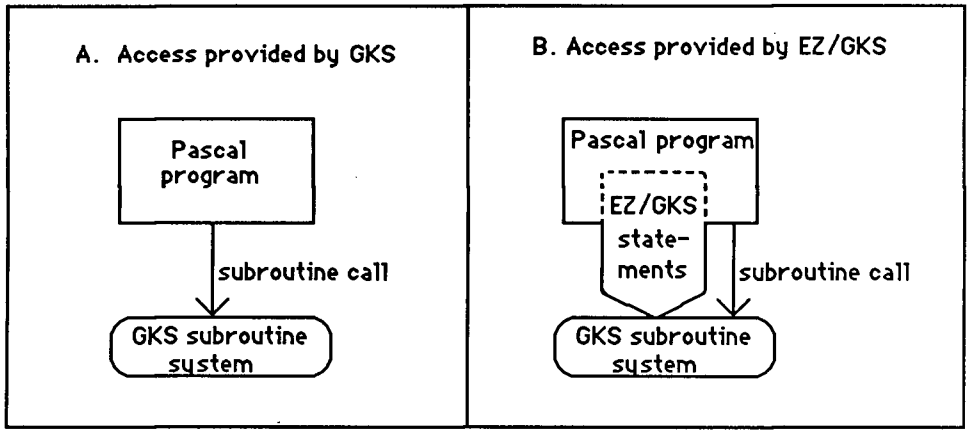


Figure 6 - Conceptual Access to Graphical Capabilities

discussed.

2.1 Contributions and Constraints of GKS

The benefits of a standard for computer graphics were recognized a decade before GKS was adopted by the ISO as an international standard. In 1974, the International Federation for Information Processing (IFIP) Graphics Subcommittee W.G. 5.2 organized a committee to investigate standardization in the field of computer graphics [BoEn82]. As a result, a workshop entitled "Methodology in Computer Graphics" was held in Seillac, France in May of 1976 to investigate the underlying concepts used in computer graphics. The principles formulated at the Seillac workshop provided a foundation for the graphics standards subsequently developed.

Two underlying principles defined at the Seillac workshop were the concept of portability and the differentiation between modeling and viewing functions

[Ende85]. The concept of portability referred to the mobility of graphical application programs, graphical data and graphics programmers between different installations, operating systems, and changing hardware configurations. Providing standardized interfaces to both application programs and to device drivers enhanced portability. Modeling was defined as the process of building a picture from component parts while viewing treated the picture as a whole. The participants at the Seillac workshop decided that only viewing functions were application independent. They determined that modeling functions were application dependent and therefore should be excluded from any graphics standardization efforts.

GKS was chosen as the foundation for the high-level graphical extension to Pascal/VS because of its acceptance as the international standard for computer graphics programming. Designed as a kernel system, it provides the minimal set of functions necessary to access the capabilities of a wide range of graphical devices in a device independent fashion [ISO85a]. Consistent with the principles formulated at the Seillac workshop, GKS includes only the fundamental graphical operations which are application independent. All but the most rudimentary aspects of modeling are excluded. The set of functions defined by GKS provided a foundation for graphics programming which has proved to be sufficient for the majority of graphical applications.

The design of EZ/GKS is compatible with the philosophy of GKS. Unlike most of its predecessors, EZ/GKS excludes all but the basic aspects of modeling. The philosophy behind the design of EZ/GKS is similar to that of GKS: to include only those aspects of graphics programming that are application independent. The designer believes that any application dependent facilities should be added to the

Application-Oriented Layer (Figure 2), not to the language extension. Exclusion of all application dependent features is necessary to produce a general purpose graphical extension.

Instead of expanding the basic modeling facilities provided by GKS, EZ/GKS provides language constructs to assist with the design and implementation of application-oriented graphical software. The graphical data types discussed in Section 3.1 simplify the temporary storage of graphical data while high-level graphical statements assist with manipulation of the data. Compound data types such as the WKSTNSET and SEGMENTSET discussed in Section 3.2 assist with manipulating sets of workstations and segments in an intuitive manner. For example, the SEGMENTSET primitive data type could be utilized to implement concepts such as the inclusion filter and exclusion filter defined in PHIGS [Brow85].

The decision to base the design of a graphical extension on a preexisting subroutine system constrains the design of the language extension twofold. First, the preexisting subroutine system defines the minimal graphical facilities which must be provided by the high-level language. The functions available are specified in detail by the subroutine system, but the methods for providing them are left to the discretion of the language designer. Second, the subroutine system implicitly defines concepts which are an integral part of its philosophy (Figure 7). The underlying concepts and philosophy of the subroutine system must be reflected in the design of the high-level language based upon it.

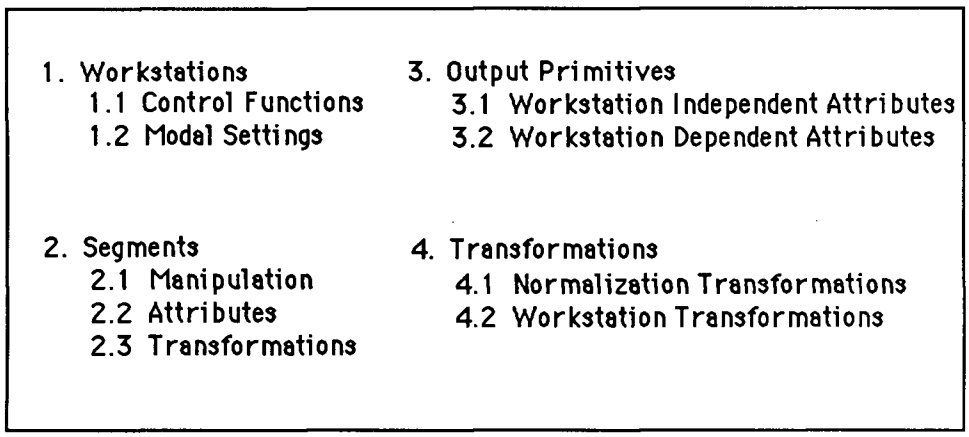


Figure 7 - Outline of the Concepts in GKS

2.2 Contributions and Constraints of Pascal

The host language chosen for a graphical extension delineates the syntactic framework within which the language extension must function. The syntactic framework defines the manner in which the features provided by the host language are specified by the user. The syntax design for a language extension should be orthogonal to the host language. The style should be internally consistent with that of the host, allowing programmers proficient in the host language to easily extend their knowledge to encompass the new capabilities added by the language extension.

The host language also restricts the language extension by its basic structure, philosophy and design. For example in Pascal/VS, modules to be compiled separately must be defined in segments. Communication between Pascal segments and the main program is restricted to parameters passed to the procedures and functions declared therein and to variables defined as DEF or REF. All variables are restricted to the static scoping and strong typing rules of the host language.

Executable statements exist within a block structured environment that provides the foundation for Pascal programs.

Pascal/VS provides a rich set of abstract data types that are inherited by any extension to the language. ENUMERATION and SUBRANGE data types provide the security of range checking on the base type INTEGER. RECORDS of nonhomogeneous data types can be declared and used as elements in multidimensional arrays or in linked lists. Typed pointers along with Pascal's dynamic storage allocation statements provide security over list processing. The SET data type, representing a combination of values from any scalar base type, is provided along with mathematical set operations such as union, intersection and difference. All of these features are inherited when Pascal is selected as the host language for a graphical extension.

Pascal/VS provides the syntactic base for the design of the EZ/GKS graphical extension. Complex data types can be built from the predefined graphical types, Pascal types and user-defined types in the usual manner provided by Pascal, allowing graphical types to be elements in multidimensional arrays, record structures or linked lists. Declaration of graphical variables is compatible with the declaration of other Pascal types, allowing them to be declared as VAR, STATIC, DEF or REF. The scope of EZ/GKS statements is comparable to that of statements in Pascal, requiring the BEGIN END block notation when encompassing more than a single statement. The syntactic structure of Pascal provides a sound foundation on which to base EZ/GKS graphical extension.

Chapter 3

Abstract Graphical Data Types

The solution to a problem may become apparent when examining the problem from an appropriate level of abstraction [Pres82]. At the lowest level of abstraction, all of the procedural details are evident. At higher levels, the focus shifts to the essential elements of the problem; irrelevant low level details are disregarded. Viewing the problem from an appropriate level of abstraction permits alternative solutions to be examined in terms that are meaningful to the problem environment [Dyme84]. Regarding solutions in meaningful terms can assist in determining the best one for the problem.

Early graphics programming systems viewed graphical concepts at a very low level of abstraction. Graphical output was generated by numerous MOVE-TO, PEN-UP and PEN-DOWN commands. All attributes were specified individually in a device-dependent manner.

During the past decade, research in computer graphics has elevated the level of abstraction by introducing such concepts as output primitives, workstations, segments and attribute bundles. Graphics programming, however, still requires attending to tedious detail at a much lower level of abstraction. For example, while the language bindings for GKS define a multitude of data types describing the parameters passed in GKS subroutine calls, no operations on the data types are provided. Therefore, the only operations available are the low-level operations for the general purpose data types provided by the host language, directing the focus on the internal representation of the graphical types rather than on their conceptual use. The following two chapters describe methods by which the

EZ/GKS graphical extension elevates the level of abstraction for graphics programming, thereby easing the burden of developing application and application-oriented graphical software. The graphical data types provided by the EZ/GKS language are described in the remainder of this chapter. Chapter 4 provides details of the operations available for the graphical data types.

Abstract data types have long been recognized as a useful construct for organizing and manipulating graphical data. An abstract data type represents a class of abstract objects and explicitly defines all operations possible on members of the class. Accessing a member of the class by any other means is not permitted. Mallgren defined the term abstract data type as [Mall82]:

> ... a data type that provide(s) a particular well-defined level of abstraction: The semantics of the operations are considered characteristic, and objects are taken as atomic; the implementation of the operations and the representation of the objects are considered irrelevant.

An abstract graphical data type in EZ/GKS may be classified as either an individual or a compound type. Unlike many previously developed graphical languages, a data type in EZ/GKS does not represent a geometric shape such as a circle, square or triangle. Instead, an individual graphical type is an atomic element representing a well-defined concept useful in graphics programming, such as a segment, window or colour (Figure 8A). A compound graphical type represents either a set or a sequence of individual graphical elements which can be referenced and manipulated as a unit (Figure 8B).

With the exception of the workstation constant, complex user-defined data types may be constructed from any of the predefined graphical data types, in the usual

A. Individual Data Types	B. Compound Data Types
POINT WKSTN SEGMENT WINDOW COLOUR SEGXFORM	POLYPOINT WKSTNSET SEGMENTSET NORMXFORM PATTERN BUNDLE

Figure 8 - Abstract Data Types in EZ/GKS

manner provided by Pascal. Graphical types may be elements of multidimensional arrays, record structures or linked lists. The atomic graphical elements within a complex structure may be addressed by using the standard Pascal path notation. The characteristics of the individual and compound graphical data types of the EZ/GKS system are portrayed in the following sections.

3.1 Individual Graphical Types

The individual data types provided by EZ/GKS are the POINT, workstation (WKSTN), SEGMENT, WINDOW, COLOUR and segment transformation (SEGXFORM). Several of these data types have proved useful in previous graphical language extensions. For example, many high-level graphical languages have a representation for a point or vector [Smit71] [MaTh81]. In addition to the vector type, HL/GKS represented the GKS concepts of workstation and segment as graphical data types [Sun86]. Colour was proposed as an abstract data type by Mallgren [Mall82]. The individual data types provided by EZ/GKS incorporate these concepts as well as several others.

The point, a graphical element that identifies a location, is the basis for defining the shape of graphical output primitives. Points in EZ/GKS may be represented by a pair of numeric expressions, or by a point expression. The variable of data type POINT represents a location in the two-dimensional plane. A POINT variable is characterized by its absolute position with respect to the X and Y axes.

The graphical data type WKSTN represents the concept of an abstract graphical workstation as defined by GKS. The GKS workstation is device independent, providing a "logical interface through which the application program controls physical devices" [ISO85a]. Unlike all other graphical data types defined in EZ/GKS, workstations are defined as Pascal structured constants. Each one is associated with a particular file and device type which may not be altered during the execution of the program.

A segment in GKS represents "a collection of display elements that can be manipulated as a unit" [ISO85a]. EZ/GKS maintains compatibility with the segment concept defined in GKS. Thus, SEGMENT variables are handled as autonomous units. A segment is created by defining a sequence of output primitives within the body of the high-level CREATE statement. Once created, the internal representation can not be altered with operations that add or delete primitives from the original segment. The appearance of the graphical output, however, may be changed geometrically by applying segment transformations or visibly by altering the segment's attributes.

The WINDOW data type represents a generalized two-dimensional rectangle aligned with the coordinate axes. A window variable may represent either a window, viewport, workstation window or workstation viewport in EZ/GKS. Which

of these the variable represents is determined by its context within a high-level graphical statement. Window variables representing a window are defined in World Coordinates (WC), while those representing a viewport or workstation window are described in Normalized Device Coordinates (NDC). Window variables representing a workstation viewport are defined by the user in Device Coordinates (DC).

Normalization and workstation transformations describe a mapping between the different coordinate spaces (eg., WC -> NDC -> DC). In EZ/GKS these transformations are described by a mapping between a pair of window variables, each of which defines a rectangular region in a different coordinate space (Section 4.2.2).

In GKS colour values are defined in terms of the RGB colour model. GKS subroutine calls are issued to store the necessary colour values in the workstation colour table of each workstation. The colour attribute of output primitives is specified by the integer index referencing the entry of the workstation colour table containing the desired colour value. Hence, the programmer must remember the current status of all workstation colour tables in order to select the desired index.

EZ/GKS provides the data type COLOUR to elevate the method by which colour attributes are managed by the application programmer. The COLOUR data type represents what the name implies. A limited number of system constants of data type COLOUR are predefined by the EZ/GKS system (Figure 9A). Incorporation of a system such as the Artist's Colour Naming System (ACNS) would be a useful addition to EZ/GKS, as it provides a natural representation for a much broader

A. EZ/GKS Colour Constants		B. Example of ACNS Colours
RED	YELLOW	BRILLIANT ORANGE-YELLOW
		PALE BLUISH-GREEN
GREEN	MAGENTA	BLACKISH BLUE
		DEEP PURPLE-RED
BLUE	CYAN	VERY LIGHT GREY
		DARK GREYISH ORANGE
BLACK	WHITE	MEDIUM RED
		VIVID GREENISH-YELLOW

Figure 9 - Colour Specification in EZ/GKS compared to the ACNS

range of colour constants [Kauf86] (Figure 9B). User-defined values for variables of data type COLOUR may also be computed from colour constants and colour variables using a graphical colour expression (Section 4.3). High-level referencing to workstation tables elevates the manner in which workstation tables are initialized and altered by the programmer (Section 4.2.3). Colour variables may then be referenced by name in graphical statements to define the colour attribute for output primitives.

The shape of a graphical object defined by a series of points may be changed by applying a geometric transformation to each point defining the object. Geometric transformations are defined in [HeBa86] as:

> . . . procedures for calculating new coordinate positions for these points, as required by a specified change in size [,position] and orientation for the object.

In GKS, geometric transformations may only be applied to graphical objects stored in a segment. The SEGXFORM data type in EZ/GKS provides a means for retaining the specification of a geometric segment transformation. Values for variables of type SEGXFORM are computed from segment transformation factors and previously

defined segment transformation variables using a graphical segment expression (Section 4.3).

3.2 Compound Graphical Types

Compound graphical data types represent multiple occurrences of an individual graphical element. A compound graphical type may be viewed as either a set or a sequence of individual graphical elements. The compound types representing a set of graphical elements are the workstation set (WKSTNSET) and the segment set (SEGMENTSET). The POLYPOINT, normalization transformation (NORMXFORM), PATTERN and BUNDLE data types can be conceptualized as a sequence of individual graphical elements.

The compound data types WKSTNSET and SEGMENTSET define a set of workstations or segments respectively, representing a number of selected individual graphical variables. A variable of type WKSTNSET contains a set of workstations to be referenced as a unit in high-level graphical statements. Similarly, a variable of type SEGMENTSET refers to a set of segments to be manipulated simultaneously. Values for variables of these data types are defined by assigning variables of their related individual graphical type (ie. WKSTN or SEGMENT, resp.) in the standard Pascal set-constructor format. Expressions of workstation sets or segment sets may compute a new set membership using any of the set operators defined by the host language. The set operators union, intersection, difference, exclusive union and all relational set operators are inherited by the WKSTNSET and SEGMENTSET data types.

A series of points are frequently used to portray the shape of a graphical

primitive. The POLYPOINT data type provided by EZ/GKS serves to retain the information about a geometric shape. A variable number of points may be stored in a POLYPOINT variable, up to the maximum permitted by the GKS implementation. A series of values may be assigned to a POLYPOINT variable using the structured point assignment (Section 4.1). Individual points may be assigned to or from a POLYPOINT variable using Pascal's usual indexed notation. When an output primitive command references a POLYPOINT variable without indicating a subrange, all points through the maximum initialized value are used to generate the graphical output primitive.

The POLYPOINT value assignment allows values for POLYPOINT variables to be initialized at compile time, assisting with the definition of non-uniform shapes (Figure 10). The POLYPOINT value assignment, similar to ordinary value assignments, may appear within any Pascal/VS value declaration. Variables to be initialized at compile time must be declared as Static or Ref variables. The values assigned in a POLYPOINT value assignment are numeric constants in the format of a structured point assignment (Section 4.1). This declaration form provides a concise method for initializing the geometric shapes of asymmetric graphical objects at compile time.

A normalization transformation "maps positions in the world coordinates to normalized device coordinates " [ISO85a]. The NORMXFORM data type in EZ/GKS retains the specification of a window-viewport mapping between the WC and NDC coordinate spaces. The value of NORMXFORM variable is defined by describing a mapping between two WINDOW variables, the first defining the window in WC and the second defining the viewport in NDC. The current normalization transformation may be set by naming a NORMXFORM variable in a high-level

```
Static
  Star : POLYPOINT;

Value
  Star := (0.95, -0.3; 0, 1; -0.95, -0.3; 0.95, 0.3;
           -0.95,  0.3; 0.95, -0.3);
```

Figure 10 - Example of the Graphical Value Assignment

TRANSFORM NT statement (Section 4.2.2).

A pattern's value is defined in GKS as a two-dimensional matrix of integers which index the colour table of a workstation. The workstation pattern table is an array of pattern values. As with colours, a pattern is defined by a GKS subroutine call that stores the pattern's value in the pattern table of a workstation. The pattern can later be selected by referencing the appropriate index of the workstation pattern table.

In EZ/GKS the PATTERN data type representing a two-dimensional array of colours may be specified by either integer indices of the workstation colour table or by colour variables. The appearance of a pattern specified by a numeric index will depend on the colour values stored in the colour table of the workstation in question. A pattern specified by a colour variable will be composed of identical colour values on each workstation on which it is displayed. The style type attribute of the fill area output primitive may be described by referencing either the desired PATTERN variable or the index of the pattern table entry in which it is stored.

An output primitive attribute defines a property of a particular type of output

primitive (eg. line, marker, fill or text). Attributes may be classified as either geometric or non-geometric. While geometric attributes define properties which affect the size or shape of the entire primitive, non-geometric primitive attributes only affect the appearance of a display element.

GKS workstations maintain four bundle tables, one for each type of output primitive. An entry in a bundle table contains values for all non-geometric attributes associated with the related primitive type.

A variable of data type BUNDLE contains values for the set of nongeometric primitive attributes associated with one type of output primitive (Figure 11). The structured bundle assignment simplifies the assignment of attribute values to a bundle variable. The bundle variable may be subsequently referenced as a unit in high-level graphical statements.

3.3 Summary

Each graphical data type presented in this chapter may be classified as either an individual or compound type. An individual graphical type represents an atomic

LINE	MARKER	FILLAREA	TEXT
TYPE WIDTH COLOUR	TYPE SIZE COLOUR	INTERIOR STYLETYPE COLOUR	FONTPRECISION EXPANSION SPACING COLOUR

Figure 11 - Output Primitive Types and their Compatible Bundle Attributes

concept which provides an abstraction useful in graphics programming. A compound graphical data type represents either a set or a sequence of atomic graphical elements which can be referenced and manipulated as a unit. Chapter 4 expounds on the abstract graphical data types by describing the operations EZ/GKS provides for variables of each graphical data type.

Chapter 4

Operations on Graphical Data Types

A high-level language may employ several methods to define the operations characterizing an abstract data type. Assignment statements may alter a variable's current value. The language syntax may be extended with additional high-level statements to manipulate the variables in a predefined manner. The symbols representing host language operators may be overloaded or new operators may be defined for expressions containing variables of the new data types. Specific characteristics of a variable may be returned to the programmer through the use of system-defined functions.

A variety of these techniques have been incorporated into EZ/GKS to specify the operations on variables of the abstract graphical data types. Structured graphical assignments promote encapsulation of abstract graphical data types by defining a structured format in which a series of values may be assigned to certain types of graphical variables. High-level graphical statements control the generation of graphical output by defining operations on graphical variables. Graphical expressions provide a mechanism for computing the values for particular types of graphical variables. System-defined functions return to the user specific attributes of a graphical variable. These concepts are further examined in the following sections.

4.1 Structured Graphical Assignments

An essential element necessary for implementing an abstract data type is encapsulation. Encapsulation restricts access to the internal representation of an

abstract type. Mallgren defines encapsulation in the following manner [Mall82]:

> . . . the primary role of a data type specification is to make explicit the boundary between an abstraction and its implementation. An encapsulation "defends" this boundary by ensuring that programs cannot access the data type representation.

Structured graphical assignments allow a sequence of values to be assigned to a graphical variable in a single assignment statement. Structured graphical assignments promote encapsulation by defining a format in which to specify a series of values to be assigned to certain types of graphical variables. The use of high-level syntax allows the programmer to initialize variables of complex graphical data types without knowledge of the internal representation of the data type. Encapsulation of graphical data types is enforced by the use of structured graphical assignments. Access to individual components of a graphical data type, except in the manner designated by the high-level language syntax, is prohibited by the EZ/GKS pre-compiler.

Four different graphical structures are defined in the syntax of EZ/GKS: the POINT structure, NORMXFORM structure, BUNDLE structure and PATTERN structure. Although the format of each structure differs, the syntax for each is representative of its related data abstraction.

Organization and manipulation of point data is a fundamental operation in computer graphics. The POINT data type together with the point structure and point expressions (Section 4.3) assist the programmer in manipulating point data in an intuitive manner. A POINT structure defines a sequence of one or more point elements. A point element may be either a point expression or a pair of numerical Pascal expressions which define the X and Y coordinates of a point.

Both formats for point elements may be mixed within a point structure (Figure 12), giving the programmer a flexible construct for defining the geometric shapes of graphical objects.

A point structure is assignment-compatible with variables of data type POINT, WINDOW or POLYPOINT, provided that the semantically appropriate number of point elements are present in the structure. The point structure may also describe the shape of output primitives directly in output primitive commands such as LINE, MARKER and FILL (Section 4.2.1).

A normalization transformation represents a mapping from a window defined in World Coordinates (WC) to a viewport defined in Normalized Device Coordinates (NDC). The NORMXFORM structure defines a normalization transformation by the mapping between two WINDOW variables and may be assigned to graphical variables of type NORMXFORM (Figure 13). The WINDOW variable referenced in the FROM clause of a NORMXFORM structure is interpreted as a window defined in WC, while the variable referenced in the TO clause is interpreted as a viewport defined in NDC. If either clause is omitted, the GKS default of the unit square is

```
Var
  Max        : REAL;        WCwindow  : WINDOW;
  UpperRight : POINT;       FigureX   : POLYPOINT;
  ...
  Max := 12.5;

  UpperRight := (25 , Max * 2 );

  WCwindow := ( ORIGIN ; UpperRight);

  FigureX    := (12.5 , Max ; UpperRight ;
                 12.5 , (Max * 2 ); 25 , 12.5 );
```

Figure 12 - Examples of Structured POINT Assignments

```
VAR
  WCWindow, NDCWindow : WINDOW;
  NT1, NT2, NT3       : NORMXFORM;
. . .

NT1 := FROM WCWindow;

NT2 := TO NDCWindow;

NT3 := FROM WCWindow TO NDCWindow;
```

Figure 13 - Examples of Structured NORMXFORM Assignments

assumed.

A bundle structure, which may be assigned to a BUNDLE variable, provides a high-level syntax for defining a set of nongeometric attributes related to a specific type of output primitive. The BUNDLE structure consists of a list of attribute specifications following a designated output primitive type (Figure 14). Each attribute specification associates the kind of attribute (eg. type, size, spacing, etc.) with an appropriate value. The attributes defined in the structure must be compatible with the named output primitive type (Figure 11). The data type of an attribute's value depends on the primitive type and the attribute kind named (Appendix B).

```
Var
  LineBundle : BUNDLE;
...
LineBundle := LINE ( TYPE is 'DASHED';
                     WIDTH is 2.5 ; COLOUR is RED );
```

Figure 14 - Example of a Structured BUNDLE Assignment

A pattern is a two-dimensional array of colours. The pattern structure allows the value of pattern variables to be defined by syntactically representing this intrinsic property (Figure 15). The pattern structure permits a list of rows of colours to be assigned to a PATTERN variable. A colour may be represented by a COLOUR variable, a COLOUR constant or by an index into the workstation colour table. Any of these representations for a colour may be mixed within a given pattern structure.

4.2 Graphical Statement Types

Graphical statements provide a user-friendly method for manipulating graphical data by performing a set of well-defined operations on the graphical variables named therein. Three categories of graphical statements are distinguished in EZ/GKS: graphical commands, graphical state changes and high-level references to workstation tables. Graphical commands specify how graphical data is defined and determine where it is stored and displayed. Graphical state changes alter the graphical environment by changing the global values that govern the final appearance of the graphical data. High-level references to workstation tables elevate the manner that workstation-dependent attributes, colours and patterns are managed by the programmer. Each of these categories are described in the following sections.

```
Var
  Checked : PATTERN;
...
Checked := ( BLUE, BLUE, 2, 2; BLUE, BLUE, 2, 2;
             2, 2, BLUE, BLUE; 2, 2, BLUE, BLUE);
```

Figure 15 - Example of Structured PATTERN Assignment

4.2.1 Graphical Commands

Graphical commands generate output primitives and manipulate graphical variables representing workstations and segments. Output primitive commands define the geometric properties of graphical output primitives and generate each primitive in its distinctive manner. Workstation commands control the graphical display on selected output surfaces, while segment commands provide for the creation, deletion, storage and display of graphical segments.

Output primitive commands provide a simple method of generating the output primitives of GKS. The commands LINE, MARKER and FILL generate the GKS Polyline, Polymarker and Fill Area primitives, respectively, the shape of which is defined by a polypoint variable or a point structure. The subrange specification may be used to restrict output to a subset of the points provided. The TEXT command defines a high-level syntax within which the location and content of textual output may be described.

High-level graphical statements provide an intuitive mechanism for manipulating variables of type WKSTN and SEGMENT. Workstation commands provide control over display surfaces (Figure 16A). A high-level graphical statement which operates on WKSTN variables may also manipulate variables containing sets of workstation elements. When referencing a variable of type WKSTNSET, the action specified is performed on each workstation in the set.

Segment commands permit the definition of a segment and provide a means for manipulating a segment or set of segments. Of the segment statement types (Figure 16B), the DELETE, SAVE and DISPLAY may also reference variables defined

A. WKSTN Statement Types	B. SEGMENT Statement Types
OUTPUT MESSAGE CLEAR REDRAW UPDATE	CREATE DELETE RENAME SAVE DISPLAY INSERT

Figure 16 - High-level Statements to Manipulate WKSTN and SEGMENT Variables

as segment sets. As with workstation sets, the requested operation referencing a segment set variable is performed once for each segment in the set. Implicit regeneration is temporarily disabled during a segment set operation, preventing the display surface from being redrawn until the set operation is completed.

Graphical output may be generated either modally or explicitly. Modal output causes display on all active workstations as the graphical primitives are being defined. Explicit output displays a graphical segment after it has been defined.

GKS defaults to modal output. In GKS the programmer is responsible for opening, activating, deactivating and closing workstations at the required time in order to generate the desired output on each display surface.

Graphical commands in EZ/GKS provide a simple manner for requesting either modal or explicit output. In addition, the responsibility for opening, activating, deactivating and closing workstations is assumed by the pre-compiler of EZ/GKS, thereby easing the burden on the graphics programmer.

Modal output in EZ/GKS is generated by the OUTPUT statement. The ONLY option restricts the destination workstations to those named in the command. The ALSO option generates output on those named in the command in addition to the workstations that are currently active. The necessary workstations are automatically activated and deactivated when an OUTPUT statement block is entered and restored upon exiting the block. OUTPUT statements may be nested within the compound statement of other OUTPUT statements in the usual Pascal manner, providing a high-level mechanism for controlling output displayed on multiple workstations (Figure 17).

Explicit output allows named segments to be displayed on specified workstations. To provide explicit access in EZ/GKS, all segments created are automatically stored in Workstation Independent Segment Storage (WISS) [Sun86], unless otherwise directed by the modal OUTPUT statement. Segments or sets of segments stored in WISS can be temporarily displayed on any workstation or set of workstations using a DISPLAY command. Such segments are erased when the workstation's display surface is regenerated. Furthermore, segments may be copied from WISS into the local storage of a workstation, Workstation Dependent Segment Storage

```
Var
  ClassA, TeacherA : WKSTNSET;
...

OUTPUT ON ClassA ALSO
  Begin
    NextProblem (ProblemNbr);     { display the next problem }
    OUTPUT ON TeacherA ONLY
      NextAnswer (ProblemNbr)     { display the related answer }
  End;
```

Figure 17 - Example of Nested OUTPUT Commands

(WDSS), using a SAVE command. All visible segments saved in a workstation's WDSS are permanently displayed on the its output surface and are redrawn each time the display surface is regenerated. The DELETE statement allows the named segment(s) to be removed from a workstation's WDSS. The DISPLAY and SAVE commands provide explicit output by naming the segment(s) to be displayed and the workstation(s) on which they are to be shown.

4.2.2 Graphical State Changes

Graphical state changes alter the graphical environment by changing the global state of GKS or by altering the state of individual workstations or segments. The state of GKS, workstations and segments in turn determine the manner in which graphical output is displayed by defining transformations, attributes and other settings that govern its final appearance.

A unique state change operator has been created to distinguish graphical state changes from ordinary Pascal and EZ/GKS commands. The operator's symbol is "<-" and can be read as "is set to". Statements using the state change operator set transformations, attributes and miscellaneous other global graphical variables used by GKS.

Three kinds of transformations alter the final appearance of graphical output: the normalization transformation, workstation transformation and segment transformation. All three kinds are set using the TRANSFORM statement (Figure 18).

The reserved word NT is an abbreviated reference to the current Normalization

Transformation in EZ/GKS. A TRANSFORM NT statement sets the current window and viewport by naming a NORMXFORM variable which is initialized to the desired window-viewport mapping (Figure 18A). A simple method is thus provided for setting the current normalization transformation by selecting one of possibly several normalization transformation variables defined in an application program.

The TRANSFORM statement referencing a workstation or workstation set variable defines a new workstation window and workstation viewport for the respective device(s) using syntax similar to the NORMXFORM structure (Figure 18B). The window variable identified in the FROM clause is the workstation window specified in NDC. The variable identified in the TO clause is the workstation viewport in device coordinates (DC).

Finally, the TRANSFORM statement referencing a segment or segment set variable applies a segment transformation to the segment(s) named (Figure 18C). When a segment set variable is referenced, implicit regeneration is disabled until the set

```
VAR
  NT1        : NORMXFORM;      FloorPlan : SEGMENTSET;
  PlanSize   : SEGXFORM;       Planners  : WKSTNSET;
  Zoom, Show : WINDOW;
```

A. Normalization Transformation

```
TRANSFORM NT <- NT1;
```

B. Workstation Transformation

```
TRANSFORM Planners <- FROM Zoom TO Show;
```

C. Segment Transformation

```
TRANSFORM FloorPlan <- PlanSize;
```

Figure 18 - Examples of the TRANSFORM Statement

operation is complete, allowing all segments to be transformed before the display surface is regenerated.

GKS maintains a set of thirteen aspect source flags (ASF's), one for each kind of nongeometric primitive attribute. At the time an output primitive is generated, the ASF determines the source from which the value for each attribute will be obtained. If the ASF indicates INDIVIDUAL, the value is obtained from the individual attribute stored in the GKS state list. Individual attributes appear as similar as possible on all workstations. If the ASF indicates BUNDLED, the value is obtained from a bundle table entry in the workstation. The appearance of a bundled attribute may differ from workstation to workstation.

In GKS, the ASF's are maintained in a single array and are set by a single subroutine call. In EZ/GKS, the aspect source flags relating to a primitive type may be set using the ASF statement (Figure 19A). An optional EXCEPT clause allows specific attribute types to be excluded, implicitly setting their ASF opposite to the value named. ASF's for other primitive types not named in the statement remain unchanged.

The ATTRIBUTES statement defines segment attributes as well as individual primitive attributes (Figure 19B). Segment attributes are altered by referencing a segment or segment set variable in an ATTRIBUTES statement followed by a list of segment attribute kinds, each with its related value. Individual primitive attributes may be specified in two ways: by referencing a bundle variable containing the desired attribute values or by listing the attribute values in a format similar to a BUNDLE structure. Unlike the BUNDLE structure, however, the attribute types listed in an ATTRIBUTES statement may be either geometric or

```
A. ASF Statement
   ASF of Line <- individual EXCEPT Colour;

B. ATTRIBUTES Statement
   ATTRIBUTES of Line <- (Type IS 'DOTTED'; Width IS 2.0 );

C. BUNDLE Statement
   BUNDLE of Line <- 2;
```

Figure 19 - Example of the ASF, ATTRIBUTES and BUNDLE Statements

nongeometric.

When the ASF of an output primitive attribute is BUNDLED, the appearance of the attribute for a primitive subsequently defined is workstation dependent. The value for the attribute is obtained from the appropriate workstation bundle table. The current index designating the bundle table entry to be used is obtained from the GKS state list. In EZ/GKS, the entry of the workstation bundle table is selected by the BUNDLE statement (Figure 19C). The high-level FIND statement assists the programmer in locating the desired bundle index (Section 4.2.3).

Other miscellaneous high-level statements defining graphical state changes set the value for the segment priority, deferral mode and clipping.

4.2.3 High-Level References to Workstation Tables

In each GKS workstation capable of output, six tables retain the values of workstation-dependent attributes. The colour table is an array of RGB colour values. The pattern table is an array of pattern values, each of which is

represented by a two-dimensional array of indices referencing entries in the workstation colour table. The other four tables are bundle tables, one for each type of output primitive, containing bundles of nongeometric attributes. In the bundle tables, colours are also represented by an index into the workstation colour table.

Although a GKS installation may predefine a limited number of values in the workstation tables, extending the number of values or customizing the tables for a particular application program requires considerable effort. Every table entry to be referenced must be defined by a GKS subroutine call. Also, in order to select the appropriate table index, the programmer must remember the current status of all workstation tables.

EZ/GKS elevates the manner by which workstation tables are managed utilizing graphical variables and high-level graphical statements. The graphical variables COLOUR, PATTERN and BUNDLE retain complex attribute values initialized by structured graphical assignments. The high-level statements STORE, CHANGE, FIND and RELEASE assist with manipulating and referencing the workstation tables in a more friendly manner.

The STORE statement stores the value of COLOUR, PATTERN or BUNDLE variables in the appropriate workstation table(s). Two forms of the STORE statement are provided: one names a list of variables to be sent to a single workstation set (Figure 20A) while the other defines a variable to be sent to each of several workstation sets (Figure 20B). If a single workstation set is named, each variable's value is stored in the same table entry in all workstations in the set. Subsequent reference to the variable's name will be interpreted by the pre-compiler to

```
CONST
  Wkstn1 = (fileA, Jupiter7);
  Wkstn2 = (fileB, Tek4027);

A. SEND Red, Blue, Green TO Wkstn2;

B. SEND ( Red TO Wkstn1; Blue TO Wkstn2);
```

Figure 20 - Examples of the SEND Statement

reference the appropriate workstation table index.

The second format associates a different variable with each workstation set named. A workstation table index free on all workstations is selected. The value stored in this table entry in each workstation is determined by its associated graphical variable. Hence, all the variables named in a SEND statement of this form may be associated with only one type of workstation table. In addition, the workstation sets named within a SEND statement may not intersect.

The FIND statement assists with finding a particular index for a workstation table entry, permitting the programmer to recall the current status of the workstation tables. For example, the statement "FIND index FOR red ON Wkstn1; blue ON Wkstn2" would set the variable named "index" to the colour table entry having RED in Wkstn2 and BLUE in Wkstn2. If the requested combination of values is not located in the named workstations, the value of zero is returned.

After one or more SEND statements have been issued to store a graphical variable in workstation tables, a CHANGE statement may be used to change the values

stored. The CHANGE statement provides a high-level mechanism for changing the values in workstation tables that were initialized by a SEND statement. When a CHANGE statement defines a new value for a graphical variable, its value is changed in all workstation tables to which the graphical variable had been sent. The CHANGE statement provides a concise method for altering the values in workstation tables.

The RELEASE statement frees the variable named from the indicated group of workstations. When a variable is freed, the association between the variable and the workstation table entries is broken. The variable name then no longer may be used to reference the workstation table entries to which it had been sent. The value in the workstation table remains unchanged until it is selected by the system for use in a subsequent SEND statement. When this index is selected for use in another SEND statement, the old value in the table entry is replaced by the new one defined.

The SEND, CHANGE, FIND and RELEASE statements elevate the workstation table manipulation methods defined by GKS, assisting the programmer with initializing, altering and referencing workstation tables.

4.3 Graphical Expressions

Expressions provide a means for calculating the value of a variable. The values for variables of the graphical data types POINT, COLOUR and SEGXFORM may be computed using graphical expressions.

Factors in a POINT expression are variables of data type POINT (Figure 21A). The

Pascal operators "+" and "-" have been overloaded to permit addition and subtraction of point variables. The default order of evaluation is the same as Pascal: from left to right. Like Pascal, any pair of factors may be parenthesized to alter the order of evaluation.

The colour expression permits new colour values to be derived by conceptually mixing ratios of colour constants and previously defined COLOUR variables (Figure 21B). The ratio of colours to mix is specified using a "part" notation. A specification as "3 PARTS GREEN + 2 PARTS BLUE" defines a new colour value created by mixing the colours green and blue in a ratio of 3:2. The colour expression provides the programmer with a simple mechanism for defining new colour values.

The segment transformation expression accumulates the composite value of a segment transformation by combining previously defined SEGXFORM variables and segment transformation factors (Figure 21C). Segment transformation factors allow specification of uniform or differential scaling, rotation, translation, shearing and reflection in an easy manner (Figure 22). Any series of

	Var pointA, pointB, pointC : POINT;
A. Point Expression	(pointC + (pointB - pointA))
B. Colour Expression	3 parts WHITE + 1 part RED
C. Segment Expression	ROTATE 45 ABOUT pointC + SHIFT (5,7)

Figure 21 - Examples of Graphical Expressions

SEGXFORM variables and segment transformation factors may be accumulated in a segment transformation expression.

4.4 System-Defined Functions

System-defined functions in EZ/GKS relay information about a graphical variable to the programmer (Appendix C). System-defined functions also defend encapsulation of abstract data types by defining an interface through which communication with the programmer is channeled, thereby avoiding access to the internal representation of graphical variables. As a result, any subsequent change in the internal representation of graphical variables can be handled entirely within the EZ/GKS system and will require no change in graphical application programs using the system.

4.5 Summary

In summary, the operations described on graphical variables in EZ/GKS include structured graphical assignments, high-level graphical statements, graphical expressions and system-defined functions. Structured graphical assignments defend the encapsulation of abstract graphical data types by defining structured formats in which to initialize certain types of graphical variables. High-level graphical statements define operations on graphical variables, alter the

SCALE	ROTATE	SHIFT
SHEAR	REFLECT	IDENTITY

Figure 22- Factor Types for Segment Transformations

graphical state and elevate the manner in which workstation tables are managed by the programmer. Graphical expressions compute new values for certain types of graphical variables, while system-defined functions return to the user specific information about a graphical variable. Together, these techniques elevate the level of abstraction necessary for graphics programming by hiding many of the tedious details required in GKS programming.

Chapter 5

Implementation

5.1 Development of the EZ/GKS Pre-Compiler

The EZ/GKS pre-compiler is a portable Pascal program which generates the appropriate GKS subroutine calls from an application program containing high-level graphical statements. A Pascal program extended with graphical statements defined in the EZ/GKS syntax is the input for the pre-compiler. The output consists of a listing of two files: the original input program and a generated Pascal program including the necessary GKS subroutine calls and supporting code.

The EZ/GKS pre-compiler was developed with the assistance of the Top-Down Compiler Writing System (CWS-TD) developed by Bochmann, Lecarme and Ward [ScCh86]. The CWS-TD system generates a complete translator by analysing an integrated description of an LL(1) language. The integrated description defines the language syntax by production rules specified in Backus Naur Form (BNF). Semantic actions are described by sections of Pascal code interlaced within the BNF. CWS-TD inserts the user-defined semantic actions into the generated translator at the designated places.

Some modifications of the CWS-TD system were necessary to facilitate the development of the EZ/GKS translator. Minor modifications include the generation of the input file listing. The underscore is now permitted in identifier names.

All identifiers handled in the CWS-TD system are case sensitive. Since the host language used for EZ/GKS is case insensitive, all symbols read from the input file are changed to lower case prior to being parsed by the translator. Thus, all reserved words and identifiers used in EZ/GKS appear to be case insensitive as well.

When implementing a language extension, a technique must be used to distinguish between statements of the host language and those of the graphical extension. Graphical statements defined by the extension require translation while statements of the host language should be copied unchanged to the file of generated code. One method of identification is to recognize only those statements which belong to the graphical extension, assuming that all others belong to the host language. Graphical statements can easily be identified as such if each statement is begun with a reserved word that is not reserved by the host language.

Two problems arise when this method is used. One occurs when the first reserved word in a graphical statement is mistyped. As the translator does not recognize the mistyped word, it assumes that the statement belongs to the host language, copying it verbatim to the file of translated code. The second problem arises when the symbol terminating a host language statement is omitted. As the missing terminator is not detected by the translator, more than one statement is copied to the output file, creating errors in the translated code if the following statement is a graphical statement in need of translation. Neither of these errors can be detected by the translator. Hence, no error message can be issued to warn the programmer.

A safer method for distinguishing statements of the host language from those of the graphical extension is used in the language LIG [BuDi82]. In LIG, both host statements and those of the graphical extension are positively identified by the initial reserved word in each statement. Graphical statements are translated while statements of the host language are copied without modification to the file of translated code. Statements which do not belong to either the host or extension can easily be identified as errors, thereby preventing the first problem described above.

For assignment statements, LIG examines the first character of the assignment operator to determine whether the statement is a Pascal assignment or a graphical assignment. The operator ":=" distinguishes Pascal assignments while the operators "<-" and "<=" distinguish graphical assignments. The responsibility for deciding if an assignment statement is host or extension is thus shifted from the translator to the application programmer.

Although this method provides more security than the one previously described, it was not deemed adequate for the purposes of EZ/GKS. If the symbol terminating a host statement is omitted, the second problem described above will still occur. In addition, the author believes that the responsibility for distinguishing between statements of the host language and graphical extension should rest with the translator, not with the application programmer.

Because Pascal is a strongly typed language, the EZ/GKS graphical extension benefits by being strongly typed as well. By the data type of a variable, the translator can easily determine if an assignment statement belongs to the host language or to the graphical extension. The data type also assists with the

translation of graphical statements and the generation of meaningful error messages.

The EZ/GKS translator parses both the host language and the graphical extension. In so doing, no statements are ever flushed to the output file, avoiding the problem of bypassing an erroneous statement without warning the programmer. By parsing Pascal, the translator is able to derive the data types of all variables declared. Information on the data type of variables referenced in high-level statements is valuable throughout the translation of EZ/GKS graphical statements.

In order to assist EZ/GKS with the task of echoing parsed host-language symbols to the output file, a modification was made to the CWS-TD system. A global boolean flag has been defined to control the printing of the previous symbol parsed onto the file of generated code. When the flag is set, the previous symbol parsed is copied to the output file. The flag is manipulated as needed within the semantic actions of the integrated description, providing a mechanism to allow host-language statements to pass through the translator unchanged.

The final modification to the CWS-TD system consists of the addition of three global buffers used to accumulate groups of symbols to be referenced as a unit in the integrated description. Variable references, factors and expressions are saved in the buffers. Each buffer is controlled by a boolean flag which is manipulated as needed from semantic actions within the integrated description. When a flag is set, symbols parsed are appended to the related buffer. The buffer can subsequently be referenced in semantic actions and inserted into the generated code in the appropriate places.

5.2 Translation Techniques

The encapsulation of graphical data types is difficult to achieve when the Pascal language is chosen as the host language. As previously noted, the Pascal language is strongly typed. Therefore, all data types referenced in constant and variable declarations must be fully described in the data type declarations. Pointers are also typed. Thus, much of the implementation of graphical data types is visible in the translated code.

The designer of EZ/GKS assumes that the application programmer does not alter the output from the pre-processor. The pre-processor prevents programmer access to the implementation of graphical variables from within the application's source code. To enforce the concept of encapsulation, the only use of graphical data types and variables permitted to the application programmer are those expressly designated by the syntax of the EZ/GKS language.

5.2.1 Simple Translation

EZ/GKS begins translation of an application program by including the GKS constant and type declarations together with the external procedure declarations for the GKS subroutine calls. The declarations for predefined graphical data types referenced in the application program are subsequently generated with the declaration of numerous graphical variables reserved for internal use by the translator. The external procedure declarations for routines in the EZ/GKS run-time library are also generated.

Upon entering the main program body of an EZ/GKS program, the GKS subroutine

calls to open GKS, open WISS and activate WISS are automatically generated. In addition, a call to GKS is generated to open each workstation which is declared as a workstation constant within the program.

Subsequent calls to activate and deactivate workstations are generated throughout the program as implied by the high-level workstation commands. For example, if an explicit SEND statement references a workstation which is not currently active, EZ/GKS automatically activates it. After the segments have been transmitted, the workstation is returned to its original state.

Workstations are automatically deactivated and closed upon leaving the main procedure. GKS is subsequently closed.

Simple translation of a high-level statement into a GKS subroutine call is achieved using variables predefined by EZ/GKS and user-defined graphical variables as the actual parameters for the appropriate GKS subroutine calls (Figure 23). Frequently, assignment statements are generated to assign values to a predefined EZ/GKS variable. For example, when an output primitive command such as LINE,

A. An EZ/GKS Statement

```
CONST HomeWkstn = WKSTN (8, Tek4027)
. . .
MESSAGE '* *  Error in Input Data  * *' to HomeWkstn;
```

B. Generated Code

```
CONST HomeWkstn = 1;
VAR   GXYString : string;
. . .
GXYString := '* *  Error in Input Data  * *';
GMessageStr (HomeWkstn, length (GXYString), GXYString);
```

Figure 23 - Simple Translation of an EZ/GKS Statement

MARKER or FILL is defined by a point structure, a statement is generated to assign each expression in the point structure to a sequential element of a predefined EZ/GKS array variable. The array variable is subsequently passed as the actual parameter of the GKS subroutine call generated by the translator.

The graphical data types SEGMENTSET and WKSTNSET allow the programmer to designate an action to be performed on each member of the set(s) named. Translation of high-level statements referencing SEGMENTSET or WKSTNSET variables require the GKS subroutine call to be generated within a FOR loop. A conditional statement within the loop allows execution of the GKS subroutine call if the loop variable is a member of the designated set. Statements referencing both a segment set and workstation set variable require the generation of two nested FOR loops (Figure 24).

A. An EZ/GKS Statement

```
VAR BldgDesign: SEGMENTSET;      Planners: WKSTNSET;
. . .
DISPLAY BldgDesign TO Planners;
```

B. Generated Code

```
VAR BldgDesign: SEGMENTSET;      Planners: WKSTNSET;
    GXVi, GXVj: INTEGER;
. . .
FOR GXVi := 0 TO GXVMaxSetSize DO
  FOR GXVj := 0 TO GXCMaxSetSize DO
      IF (GXVi  IN Planners) &
         (GXVj  IN BldgDesign)
      THEN GCopySegWs (GXVi, GXVj);
```

Figure 24 - Translation Generated by References to Workstation and Segment Sets

5.2.2 The Run-Time Library and Inquiry Functions

The code generated in Figure 24B is not sufficient to accomplish the action specified in Figure 24A. If output is requested on a workstation that is inactive, the action would not be visible on the workstation requested. In addition, if a workstation in the set allows implicit regeneration, its display surface may be redrawn after the receipt of each segment.

The EZ/GKS run-time library contains procedures and functions that are useful repeatedly throughout the translated application program. All routine identifiers reserved for use by the translator begin with the prefix GX. In addition, the library contains the user-accessible routines defined in Appendix C.

At run time, the library is loaded with the object code of the application program. Procedures in the run-time library assist with translation by defining a simple interface (ie. the subroutine call) for requesting a complex operation. The subroutine call is generated by the translator and is inserted into the generated code whenever the complex operation is needed. Routines in the run-time library also serve to hide from the application programmer the data types, static variables and implementation details of the procedures and functions used by the translator.

Many procedures within the run-time library use GKS inquiry functions to determine the current state of GKS and of the individual workstations. For example, the function GXWsSet calls a GKS inquiry function to obtain a list of the workstations currently active or open. The list is converted into a workstation set and returned to the caller. The routine GXSuppR uses the set of active

workstations returned from GXWsSet and issues a GKS inquiry function to determine the deferral state of each workstation in the set. If the deferral state is allowed, a call is generated to temporarily suppress implicit regeneration. The routine GXRestR restores the deferral state of each workstation suppressed by GXSuppR to its original state. Procedures such as these in the run-time library are called from the translated program to alter and restore the state of GKS and of the individual workstations as needed throughout the translated application program.

5.2.3 Translation of Attribute Values

Attribute values in the Pascal binding of GKS are specified using a variety of enumerated data types and integers identifying different attribute styles. To allow consistent specification of attribute values, the values for all attribute types having enumerated set of values are defined by string constants or variables in EZ/GKS (Appendix B). The parameter to the GKS subroutine call defining the attribute's value is a call to a function in the run-time library returning an integer value. The value returned by the function call is coerced into the data type appropriate for the actual parameter being generated (Figure 25). This technique allows the programmer to indicate attribute values using mnemonic strings rather than integer values.

Upon initialization of an EZ/GKS program, a procedure in the run-time library is called to read the Attribute Value File. The Attribute Value File contains a mnemonic name, context and integer value for each valid enumerated attribute value. Two arrays are created within the application program: the Context Limit Array and the Attribute Value Array. The Context Limit Array defines the

A. An EZ/GKS Statement

```
ATTRIBUTES of FILL <- (INTERIOR is 'SOLID');
```

B. Generated Code

```
GSetFillIntStyle (GTInterior (GXAtrVal (GXVArray,
                  GXVLimitArray [13]. min,
                  GXVLimitArray [13]. max, 'SOLID' )));
```

Figure 25 - Translation Generated to Set an Attribute Value

beginning and ending indices within which the values for a given attribute type may be found in the Attribute Value Array. The Attribute Value Array stores the mnemonic names representing attribute values and an associated numeric value.

When an attribute value is set by referencing a particular string, the routine GXAtrVal in the run-time library is called to search for the string within the appropriate context limits. If the string is found, the related value is returned. Otherwise, a designated default value is returned.

The Attribute Value File may be extended to include additional mnemonic names for the installation-dependent values of attributes such as line type, marker type, font and hatch style. No change in the EZ/GKS translator is required to accommodate additional attribute values.

5.2.4 Workstation Table Management

Workstation tables in GKS are addressed by an integer index. The programmer must recall the current status of the tables in order to specify the index of the

desired colour, pattern or bundle entry.

EZ/GKS provides the high-level statements STORE, CHANGE, FIND and RELEASE to elevate the manner in which workstation tables are referenced by the programmer. High-level statements permit values to be specified by referencing a colour, pattern or bundle variable. If an ambiguity arises by referencing a graphical variable, the workstation index may be located with the help of the FIND command.

Management of workstation tables is accomplished through the use of one Master Workstation Array (MWA) and six Index Arrays (IA), one for each type of workstation table. The MWA stores the union of all workstations to which a graphical variable has been sent and references an Index List. The Index List identifies the workstation table index used for each subset of workstations. Each entry of an IA stores the union of the workstations for which the related index is available. The size of an IA is the same as the maximum size for the related type of workstation table.

The Workstation Description Table in GKS stores the allowable number of table indices for each type of workstation. Inquiry functions in EZ/GKS access this information in order to appropriately initialize the IAs, thereby preventing any indices out of range from being selected in a SEND statement.

Each variable of type colour, pattern or bundle has associated with it an index into the MWA. If the index is zero, the variable's value has not been sent to a workstation. When a variable is first sent to a set of workstations, an unused entry in the MWA is selected and the index value is stored in the variable. The

appropriate IA is searched to find the first index available on all workstations in the set. The workstation union is updated by the workstation set to indicate that the index is in use. The workstation set union is updated in the MWA and an Index node is inserted in the list.

Processing for the RELEASE statement is the reverse of the SEND statement. A node in the Index List is updated or deleted. Workstations are removed from the workstation set union stored in the MWA and the related IA.

When the value of a graphical variable is altered in an assignment statement, the connection to all table indices is broken. The Index List is deleted and the associated MWA is designated as unused. The workstation set union in the IA is updated to show the new indices available and the value of the MWA index stored in the variable is reset to zero.

The CHANGE statement updates the values in all workstations to which a variable has been stored. The Index List assists by identifying the index entry under which the variable's old value had been stored. The value in each workstation in the list is updated with the new value defined in the statement.

The FIND command searches the Index Lists associated with the specified graphical variables to locate one index in common on all of the workstations. When an available index is located, its is returned. An index value of zero is returned if no index is available on all of the designated workstations.

The SEND, CHANGE, FIND and RELEASE statements assist the application programmer with the management of workstation tables.

Chapter 6

Conclusion

This thesis has demonstrated the feasibility of elevating the functionality defined in the GKS subroutine system to a high-level graphical extension of a general-purpose programming language. Although the design of EZ/GKS is dependent on the features present in Pascal/VS, the concepts presented are essentially language independent. Equivalent functionality could be provided in other general purpose programming languages with varying degrees of difficulty.

The GKS subroutine system has the advantage over a high-level language extension in being relatively language independent. The standard subroutine call is an interface mechanism available in most general purpose programming languages. A particular procedure defined in GKS requires little alteration from one language to another. As a result, a programmer proficient in using GKS through one language will have little trouble adapting this knowledge to use GKS in a different language.

A high-level graphical extension is more dependent than a subroutine system on the features and facilities of the host language. As a result, an implementation of a high-level graphical extension providing similar functionality will vary more from one host language to another. The author believes, however, that the benefits of graphics programming using a high-level language far exceed this disadvantage. A high-level language extension is able to hide much of the detail required in GKS programming, making the high-level language easier to learn and use.

Despite efforts to do so, the objective of making EZ/GKS compatible with the host language has not been entirely accomplished. For example, any Pascal data type may be used to declare either constants or variables. In EZ/GKS, the workstation data type may only be used to declare workstation constants while all other graphical data types may be only used to declare graphical variables. While the ability to define certain types of graphical constants could be added to EZ/GKS, the semantics of the graphical types prohibits complete compatibility with Pascal in this regard. In practice, this minor variation should pose no significant difficulty.

Three areas related to this topic have not been considered within the scope of this project and are prime candidates for further investigation. First, the inquiry, error and metafile functions of level 2A were excluded from the scope of this work. While error and metafile functions are areas of small concern, finding a user-friendly manner to handle the information accessible through the host of inquiry functions provided by GKS is a significant issue.

Secondly, a method for handling GKS input functions associated with levels 2b and 2c has not been addressed. The ability to manage graphical input through a high-level extension to a general purpose programming language could revolutionize interactive graphics programming.

Finally, a method for extending this work to encompass the new GKS-3D [ISO85b] draft standard could also be considered.

References

[BoEn82] Bono, Peter R., Jose L. Encarnacao, F. Robert A. Hopgood and Paul J. W. ten Hagen, "GKS - The First Graphics Standard," *IEEE Computer Graphics & Applications,* Vol. 2 (July 1982): 9-23.

[Brow85] Brown, Maxine D., Understanding PHIGS, Megatek Corporation, San Diego, CA., 1985.

[BuDi82] Burger, S., E. Dietrich and G. F. Schrack, "Development of a Preprocessor for the Pascal-extended Graphical Language LIG/P with the aid of Compiler/Compiler," in: W. Henhapl, Ed., GI-Fachgesprach, Munich, (1982): 120-154.

[Dyme84] Dyment, Doug, CPSC 538B - Software Engineering Class Notes, University of British Columbia, Fall term, 1984.

[Ende85] Enderle, Gunter, "Guest Editor's Introduction, Computer Graphics Standards," *Computers & Graphics,* Vol. 9 (No. 1, 1985): 1-8.

[GiEn72] Giloi, W.K., J. Encarancao and W. Kestner, "APL-G APL Extended for Graphics," ONLINE '72 International Conference on Online Interactive Computing, Uxbridge, England (Sept 1972): 579-599.

[HeBa86] Hearn, Donald and M. Pauline Baker, Computer Graphics, Englewood Cliffs, NJ.: Prentice-Hall, 1986.

[IBM81] Pascal/VS Language Reference Manual 2nd Ed., IBM Corporation, 1981.

[ISO85a] ISO DIS 7942, Graphical Kernel System Functional Description, Oct. 1985.

[ISO85b] ISO-DP 8805, GKS-3D Functional Description, March, 1985.

[Kauf86] Kaufman, Arie, "Computer Artist's Colour Naming System," *The Visual Computer,* Vol. 2 (No. 4, 1986): 255-260.

[Mall82] Mallgren, William R., Formal Specification of Interactive Graphics Programming Languages, Cambridge, MA.: MIT Press, 1982.

[MaTh81] Magnenat-Thalmann, Nadia and Daniel Thalmann, "A Graphical Pascal Extension Based on Graphical Types," *Software: Practice & Experience,* Vol. 11 (Jan., 1981): 53-61.

[McLe78] McLean, M.J., "A Survey of Interactive Graphics Software," *The Australian Computer Journal,* Vol. 10 (Feb. 1978): 11-22.

[Pres82] Pressman, Roger S., Software Engineering: A Practitioner's Approach, New York, NY.: McGraw Hill, 1982.

[Smit71] Smith, David N., "GPL/I - A PL/I Extension for Computer Graphics," Spring Joint Computer Conference, Atlantic City, NJ. (May 1971): 511-528.

[Sun86] Sun, Hanqui, A High-Level Graphics Language Based on the Graphical Kernel System, M. A. Sc. Thesis, University of British Columbia, 1986.

[ScCh86] Schrack, G.F, Gordon Gheng, Harold Leung and Benjamin Yu, CWS-TD User's Manual 2nd ed., University of British Columbia, Department of Electrical Engineering, Feb. 1986.

Appendix A

Syntax Specification for EZ/GKS
Outline

I. Modifications to Pascal syntax diagrams

II. Declaration of graphical data types

III. Graphical initialization

IV. Structured graphical assignments

V. High-level graphical statements

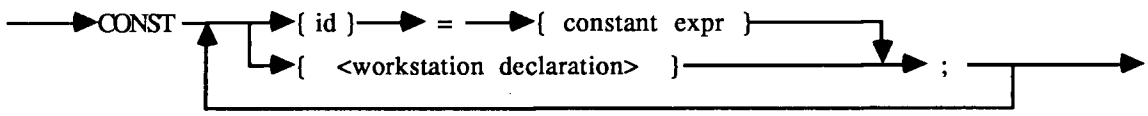
- A. Graphical commands
  - 1. Workstation commands
  - 2. Output primitive commands
  - 3. Segment commands
- B. Graphical State Changes
  - 1. Transformations
  - 2. Attribute specification
  - 3. Miscellaneous settings
- C. High-level referencing to workstation tables

VI. Graphical expressions

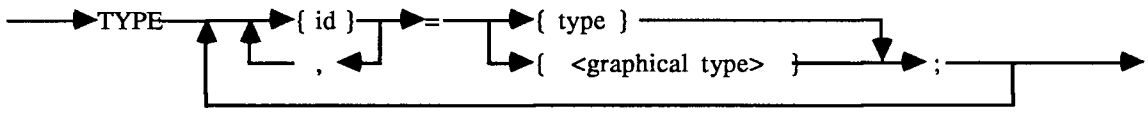
VII. Summary of notation

I. Modifications to Pascal/VS Syntax Diagrams

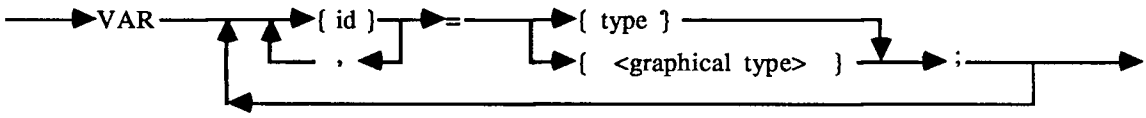
1. Constant-dcl:



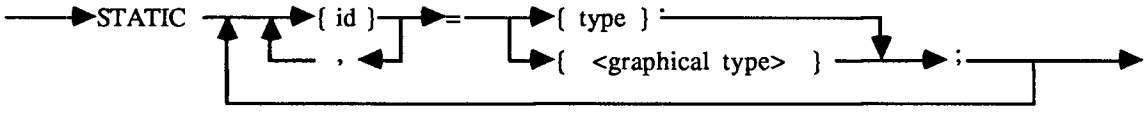
2. Type-dcl



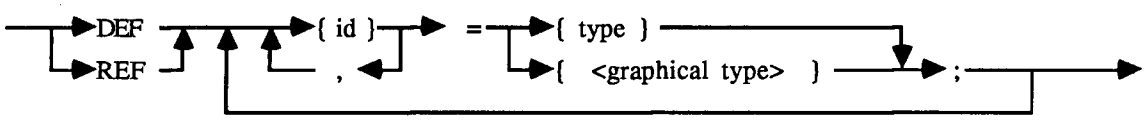
3. Var-dcl



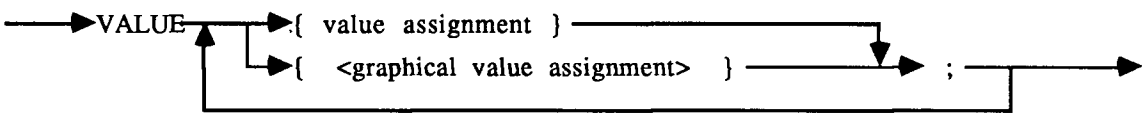
4. Static-dcl



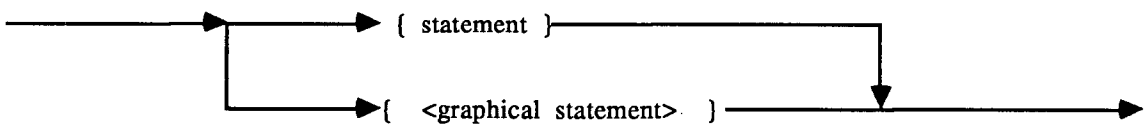
5. Def-dcl



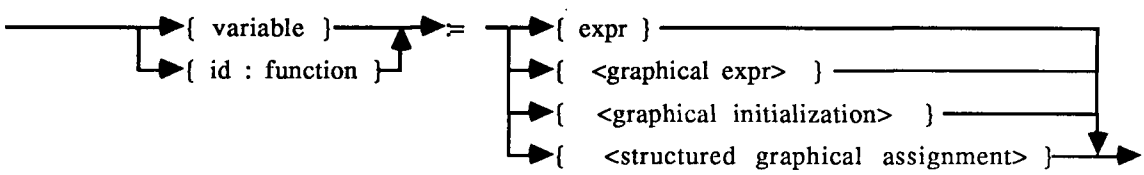
6. Value-dcl



7. Statement



8. Assignment-statement



II. Declaration of graphical data types:

<workstation declaration> ::= <variable* : wkstn> = **WKSTN**
(<integer> , <device type>)

<device type> ::= **PLOTFILE**
| **GKSMETAOUT**
| **GKSMETAIN**
| **TEK4027**
| **JUPITER7**
| **PRINTRONIX**
| **QMS**
| < other device type** >

<graphical type> ::= **POINT**
| **SEGMENT**
| **WINDOW**
| **BUNDLE**
| **COLOUR**
| **POLYPOINT**
| **SEGMENTSET**
| **WKSTNSET**
| **NORMXFORM**
| **PATTERN**
| **SEGXFORM**

<graphical value assignment> ::= <variable* : polypoint>
:= <constant point list>

<constant point list> ::= (<constant point> { ; <constant point> }$^{+}$)

<constant point> ::= <signed number> , <signed number>

<signed number> ::= { + | - | EMPTY } <unsigned number>

<unsigned number> ::= <integer> | <real number>

<integer>	::=	<digit>$^{+}$
<real number>	::=	<digit>$^{+}$. <digit>*
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

III. Graphical initialization:

<graphical initialization> ::= "

IV. Structured Graphical Assignments:

```
<structured graphical assignment>
                        ::=     <point structure>
                          |     <normxform structure>
                          |     <bundle structure>
                          |     <pattern structure>

<point structure>       ::=  (    {    <expr : point>
                               |    <expr * : numeric>
                                    , <expr * : numeric>
                               }
                             { ;  {    <expr : point>
                               |    <expr * : numeric>
                                    , <expr * : numeric>
                               }
                             }*
                             )

<normxform structure>   ::=  { FROM   <variable* : window>
                                 {    TO    <variable* : window>
                                 |    EMPTY                 }
                             | TO     <variable* : window>  }

<bundle structure>      ::=  <primitive type> <nongeometric attribute list>

<primitive type>        ::=     LINE          |    MARKER
                          |     FILLAREA      |    TEXT

<nongeometric attribute list>  ::=  ( <nongeometric attribute>
                                        { ; <nongeometric attribute> }*
                                    )

<nongeometric attribute>       ::=  <nongeometric attribute type>
                                        IS    <attribute value**>
```

<nongeometric attribute type> ::= **TYPE**
| **WIDTH**
| **COLOUR**
| **SIZE**
| **INTERIOR**
| **STYLETYPE**
| **FONTPRECISION**
| **EXPANSION**
| **SPACING**

<pattern structure> ::= (< pattern element >
{ , < pattern element > }$^{+}$
{ ; < pattern element >
{ , < pattern element > }$^{+}$ }*
)

<pattern element> ::= <colour>
| <variable*: integer>
| <integer>

<colour> ::= <variable* : colour>
| <colour constant>

<colour constant> ::= **RED** | **GREEN**
| **BLUE** | **YELLOW**
| **MAGENTA** | **CYAN**
| **BLACK** | **WHITE**

V. High-level graphical statements:

<graphical statement> ::= <graphical command>
| <graphical state changes>
| <workstation table manipulation>

A. Graphical commands:

<graphical command> ::= <workstation command>
| <output primitive command>
| <segment command>

1. Workstation commands:

<workstation command> ::= <output to workstation>
| <message to workstation>
| <clear workstation>
| <redraw workstation>
| <update workstation>

<output to workstation> ::= **OUTPUT TO** <workstation group>
{ **ONLY** | **ALSO** }
<statement*>

<message to workstation> ::= **MESSAGE** { <string*>
| <variable* : string> }
TO <workstation group>

<clear workstation> ::= **CLEAR** <workstation group>
{ **ALWAYS**
| EMPTY }

<redraw workstation> ::= **REDRAW** <workstation group>

<update workstation> ::= **UPDATE** <workstation group>
{ **NOW**
| EMPTY }

<workstation group> ::= { <id* : wkstn>
| <variable* : wkstnset> }

2. Output primitive commands:

<output primitive command> ::= <line primitive>
| <marker primitive>
| <fillarea primitive>
| <text primitive>
| <cell array primitive>

<line primitive> ::= **LINE** <subrange> **AT**
{ <variable* : polypoint>
| <point structure> }

<marker primitive> ::= **MARKER** <subrange> **AT**
{ <variable* : polypoint>
| <point structure> }

<fillarea primitive> ::= **FILL** <subrange> **AT**
{ <variable* : polypoint>
| <point structure> }

<text primitive> ::= **TEXT** { <string*> | <variable*: string> }
AT <point>

<cell array primitive> ::= **CELLARRAY IN** <variable*:window>
OF <pattern>

<subrange> ::= [1 .. <integer value>]
| EMPTY

<point> ::= <variable* : point>
| <point structure>

3. Segment commands

<segment command> ::= <create segment>
| <delete segments>
| <rename segment>
| <save segments>
| <display segments>
| <insert segment>

<create segment> ::= **CREATE** <variable* : segment>
<statement>

<delete segments> ::= **DELETE** <segment group>
{ **FROM** <workstation group>
| EMPTY }

<segment group> ::= <variable* : segment>
| <variable* : segmentset>

<rename segment> ::= **RENAME** <variable* : segment>
TO <variable* : segment>

<save segments> ::= **SAVE** <segment group>
TO <workstation group>

<display segments> ::= **DISPLAY** { <segment group> **TO**
| EMPTY }
<workstation group>

<insert segment> ::= **INSERT** <variable* : segment>
{ **WITH** <variable* : segxform>
| EMPTY }

B. Graphical state changes:

```
<graphical state change>   ::=   <transformation>
                             |   <attribute specification>
                             |   <miscellaneous settings>
```

1. Transformation:

```
<transformation>   ::=   TRANSFORM
                         { NT          <-    <variable* : normxform>
                         | <segment group>
                                       <-    <variable* : segxform>
                         | <workstation group>
                                       <-
                                 {     FROM  <variable* : window>
                                       { TO  <variable* : window>
                                       | EMPTY }
                                 |     TO    <variable* : window> } }
```

2. Attribute specification:

```
<attribute specification>   ::=   <set aspect source>
                                |   <set attributes>
                                |   <select bundled attributes>

<set aspect source>   ::=   ASF OF <primitive type>
                                <-    <attribute source>
                                { EXCEPT <nongeometric attribute type>
                                      { , <nongeometric attribute type> }*
                                | EMPTY }

<attribute source>   ::=   INDIVIDUAL | BUNDLED
```

```
<set attributes>              ::=  ATTRIBUTES OF
                                   {     <primitive type>
                                         <-    {     <variable* : bundle>
                                               |     <attribute list>  }
                                   |     <segment group>
                                         <-    <segment attribute list>
                                   }

<attribute list>              ::=  (  <attribute >  {  ;  <attribute>  }*  )

<attribute>                   ::=        <nongeometric attribute>
                                   |     <geometric attribute>

<geometric attribute>         ::=  <geometric attribute type>
                                         IS    <attribute value**>

<geometric attribute type>    ::=        PATTERNSIZE
                                   |     PATTERNSTART
                                   |     PATH
                                   |     HEIGHT
                                   |     ALIGNMENT
                                   |     ANGLE

<segment group>               ::=        <variable* : segment>
                                   |     <variable* : segmentset>

<segment attribute list>      ::=  (     <segment attribute>
                                         { ; <segment attribute> }*  )

<segment attribute>           ::=  <segment attribute type> IS <attribute value**>

<segment attribute type>      ::=  HIGHLIGHTING  |  VISIBILITY

<select bundled attributes>   ::=  BUNDLE OF <primitive type>
                                         <-  {     <integer>
                                             |     <variable*: integer>
                                             |     <variable*: bundle>  }
```

3. Miscellaneous settings:

<miscellaneous settings> ::= <set segment priority>
| <set implicit regeneration>
| <set deferral state>
| <set clipping>

<set segment priority> ::= **PRIORITY OF** <segment group>
<- <real value>

<set deferral> ::= **DEFERRAL OF** <workstation group>
<- { <defer item>
| <defer list> }

<defer list> ::= (<defer item> { ; <defer item> }*)

<defer item> ::= **IMPLICITREDRAW IS**
{ **SUPPRESSED** | **ALLOWED** }
| **UPDATETIME IS**
{ **NOW** | **LATER** }

<set clipping> ::= **CLIPPING <-** { **ON** | **OFF** }

C. High-level referencing of workstation tables:

<workstation table manipulation>
::= <store in workstation tables>
| <change workstation table entries>
| <release from workstation tables>

<store in workstation tables> ::= **STORE**
{ (<workstation table entry>
IN <workstation group>
{ ; <workstation table entry>
IN <workstation group> }*)
| <workstation table list>
IN <workstation group>

```
                                    }

<workstation table entry>   ::=     <variable* : colour>
                            |       <variable* : pattern>
                            |       <variable* : bundle>

<workstation table list>    ::=     ( <workstation table entry>
                                    { , <workstation table entry> }* )

<change workstation table entry>
                            ::=     CHANGE
                                    {  <variable* : colour>     TO    <expr: colour>
                                    |  <variable* : pattern>
                                                TO    <pattern structure>
                                    |  <variable* : bundle>
                                                TO    <nongeometric attribute list>
                                    }

<find workstation index>    ::=     FIND  <variable*: integer>
                                          FOR   <workstation table entry>
                                                ON    <workstation group>
                                          {  ;  <workstation table entry>
                                                ON    <workstation group>  }*

<release from workstation table>
                            ::=     RELEASE
                                                {     <workstation table entry>
                                                |     <workstation table list>
                                                }
                                                {     FROM  <workstation group>
                                                |     EMPTY
                                                }
```

VI. Graphical Expressions

<graphical expr> ::= <expr : point>
| <expr : colour>
| <expr : segxform>

<expr : point> ::= <variable* : point>
{ + <variable* : point>
| - <variable* : point> }*

<expr : colour> ::= { <integer> | <variable* : integer> }
{ **PART** | **PARTS** } <colour>
{ + { <integer> | <variable* : integer> }
{ **PART** | **PARTS** } <colour> }+

<expr : segxform> ::= {
<variable* : segxform>
| <segxform factor> }
{ + { <variable* : segxform>
| <segxform factor> }
}+

<segxform factor> ::= **SCALE** { <real value>
| <point> }
FROM <point>

| **ROTATE** <real value>
ABOUT <point>

| **SHIFT** <point>

| **SHEAR** <real value> %
IN <direction>

| **REFLECT** **IN** <direction>

| **IDENTITY**

<direction> ::= **X** | **Y**

VII. Summary of notation

SYMBOL	DEFINITION
<x>	Nonterminal symbol x
<x : y>	Nonterminal x of data type y
x	Terminal symbol x
::=	"is defined as"
{ }	Grouping
|	Alternation
$\langle x \rangle^{+}$	One or more repetitions of <x>
$\langle x \rangle^{*}$	Zero or more repetitions of <x>
$\langle x^{*} \rangle$	Nonterminal x is defined by the host language (For definition of x, see [IBM81])
$\langle x^{**} \rangle$	See Appendix B for details

Appendix B

B.1 Nongeometric Output Primitive Attributes

Primitive	Attribute Type	Data Type	Values
LINE	TYPE	STRING	SOLID, DASHED, DOTTED, DASHDOT @
	WIDTH	REAL	any >= 0
	COLOUR	COLOUR*	any
MARKER	TYPE	STRING	DOT, PLUS, STAR, X, O @
	SIZE	REAL	any >= 0
	COLOUR	COLOUR*	any
FILLAREA	INTERIOR	STRING	HOLLOW, SOLID, PATTERN, HATCH
	STYLETYPE	PATTERN* or STRING	any @
	COLOUR	COLOUR*	any
TEXT	FONTPRECISION	structure	(font ; precision)
	font	STRING	@
	precision	STRING	STRING, CHAR, STROKE
	EXPANSION	REAL	any >= 0
	SPACING	REAL	any
	COLOUR	COLOUR*	any
			NOTES: @ indicates that implementation dependent names can be added to the values listed * indicates that an integer index of the workstation table may be substituted

Appendix B

B.2 Geometric Output Primitives

Primitive	Attribute Type	Data Type	Values
FILLAREA	PATTERNSIZE	POINT	any
	PATTERNSTART	POINT	any
TEXT	PATH	STRING	RIGHT, LEFT, UP, DOWN
	HEIGHT	REAL	any > 0
	ALIGNMENT	structure	(horizontal ; vertical)
	horizontal	STRING	NORMAL, LEFT, CENTRE, RIGHT
	vertical	STRING	NORMAL, TOP, CAP, HALF, BASE, BOTTOM
	ANGLE	REAL	any

Appendix B

B.3 Segment Attributes

Attribute Type	Data Type	Values
VISIBILITY	STRING	VISIBLE, INVISIBLE
HIGHLIGHTING	STRING	NORMAL, HIGHLIGHT

Appendix C

System Defined Functions

Name	Description	Parameter Data Types	Result
XCoord	Get X coordinate	POINT	REAL
YCoord	Get Y coordinate	POINT	REAL
LoLeft	Get lower left corner	WINDOW	POINT
LoRight	Get lower right corner	WINDOW	POINT
UpLeft	Get upper left corner	WINDOW	POINT
UpRight	Get upper right corner	WINDOW	POINT
RedVal	Get RED component of RGB value	COLOUR	REAL
GreenVal	Get GREEN component of RGB value	COLOUR	REAL
BlueVal	Get BLUE component of RGB value	COLOUR	REAL
MaxPts	Get index value of the last initialize point	POLYPOINT	INTEGER
XMax	Get the largest X value	POLYPOINT	REAL
XMin	Get the smallest X value	POLYPOINT	REAL
YMax	Get the largest Y value	POLYPOINT	REAL
YMin	Get the smallest Y value	POLYPOINT	REAL
GetPt*	Get a point	POLYPOINT X INTEGER	POINT
GetWin	Get the window	NORMXFORM	WINDOW
GetVP	Get the viewport	NORMXFORM	WINDOW
RowLen	Get row length	PATTERN	INTEGER
ColLen	Get column length	PATTERN	INTEGER

* Synonym: GetPt (P1, I1) is P1 [I1]

Appendix D

Example Programs

```
Program EZGKS_Demo1;

  {*************************************************************************
  *                                                                        *
  *  This program demonstrates the use of the EZ/GKS precompiler.          *
  *                                                                        *
  *************************************************************************}


  CONST
    Printer    = WKSTN (21, QMS);

  TYPE
    NationRec  = record
      flag     : SEGMENT;
      position : SEGXFORM;
      End;

    Nation = (Canada, USA, England, China, France, Germany,
              USSR, Japan);

    NationArray = array [Canada .. Japan] of NationRec;

  VAR
    ArrayOfNations : NationArray;
    Star           : SEGMENT;
    Pi             : real;



    PROCEDURE GenerateFlags (Var NatnArr : NationArray);
      { This routine calls a routine to generate the graphical
        segment for each country's flag.                      }

      Var
        Grey     : GAColArray;
        FlagSize : POLYPOINT;


      PROCEDURE ComputeArc (CONST center: POINT; CONST radius, alpha, beta: real;
                            CONST ClockWise: boolean; CONST InitialIndex, NbrPts:
                            integer; VAR result: POLYPOINT);
        { This routine computes an arc centered at CENTER with a
          radius of RADIUS between angles ALPHA and BETA.  The
          direction of the arc is determined by ClockWise and the
          distance between the points is determined by the NBRPTS
          and the length of the arc requested.  The points computed
          are stored in the RESULT beginning with the INITIALINDEX. }

          Var
```

```
    I : integer;
    temp, incre : real;
    holdpt : POINT;

  BEGIN
    Result := '';
    If alpha >= beta
    Then if ClockWise
         Then incre := -(alpha - beta) / (NbrPts - 1)
         Else incre := (2 * Pi - (alpha - beta)) /
                       (NbrPts - 1)
    Else if Clockwise
         Then incre := -(2 * Pi - (beta - alpha)) /
                        (NbrPts - 1)
         Else incre := (alpha - beta) / (NbrPts - 1);

    For i := 0 to NbrPts - 1 Do
       Begin
         temp := alpha + i * incre;
         holdpt := (Xcoord(center) + radius * cos(temp),
                    Ycoord(center) + radius * sin(temp));
         result [i + InitialIndex] := holdpt;
       End;
  END; { Compute Arc }


PROCEDURE DetermineSize (Name : Nation; Var OutLine : POLYPOINT);
   { This routine determines the size of the flag based on
     the aspect ratio stored.                                  }

   CONST
     Length = 20;

   TYPE
     AspectArray = ARRAY [Canada .. Japan] of real;

   STATIC
     AspectRatios : AspectArray;

   VALUE
     AspectRatios := aspectArray (0.5, 0.5128, 0.5428, 0.67857,
                     0.6428, 0.6129, 0.57142, 0.6428);
   VAR
     bottom : real;

   BEGIN
     OutLine := (-10, 10; 10, 10);
     bottom  := (length * AspectRatios[name]) - 10;
     Outline [3] := ( 10, bottom);
     OutLine [4] := (-10, bottom);
     OutLine [5] := OutLine [1];
```

```
  END;  { DetermineSize }


PROCEDURE CreateStar (Const FillStar: boolean; Var StarSeg: SEGMENT);
  { This routine creates a star from -1,-1 to 1,1 in WC.
    If FillStar is true, a filled star is created. Otherwise,
    an outline is created. }

  VAR
    StarUnits : WINDOW;
    StarNT    : NORMXFORM;

  STATIC
    StarPoints   : POLYPOINT;

  VALUE
    StarPoints   :=  (  0.95, -0.3;  0  ,  1  ;
                       -0.95, -0.3; 0.95,  0.3;
                       -0.95,  0.3; 0.95, -0.3);
  BEGIN
    Create StarSeg
      Begin
        Attributes of Fill <- (colour is 0);
        StarUnits := (-1, -1; 1, 1);
        StarNT    := FROM StarUnits;
        Transform NT <- StarNT;
        If FillStar
        Then Attributes of Fill <- (interior is 'solid')
        Else Attributes of Fill <- (interior is 'hollow');
        Fill at StarPoints;
      End; { create StarSeg }
  END;     { CreateStar }


PROCEDURE DoCanada (outline : POLYPOINT; VAR FlagSeg : SEGMENT);
   { Define the national flag of Canada }

   STATIC
      MapleLeaf : POLYPOINT;

   VALUE
      MapleLeaf := ( -0.3, -10.0; -0.3, -2.4; -3.3, -4.3;
                     -3.3,  -3.3; -5.8, -3.3; -4.4, -1.5;
                     -5.8,  -1.2; -4.0,  0.0; -7.9,  2.9;
                     -6.9,   2.7; -8.3,  4.6; -5.8,  4.6;
                     -6.3,   5.9; -2.0,  3.3; -2.0,  7.9;
                     -1.0,   7.4;  0.0,  9.4);

   VAR
      TempPT : POINT;
      TempPolyPt : POLYPOINT;
```

```
   BEGIN
      Create FlagSeg
         Begin
           Line at OutLine;
           Attributes of Fill <- (interior is 'solid');
           Fill at MapleLeaf;
           TempPt := (5,0);
           TempPolyPt := (Outline[1]; Outline[1] + TempPt;
                    Outline[4] + TempPt; Outline[4]);
           Fill at TempPolyPt;
           TempPolyPt := (Outline[2] - TempPt; Outline[2];
                    Outline[3] - TempPt; Outline[3]);
           Fill at TempPolyPt;
           Text 'Canada' at (-6, -8);
         End;
   END;


PROCEDURE DoUSA (outline : POLYPOINT; VAR FlagSeg : SEGMENT);
   { Define the national flag of the U.S.A. }

   VAR
     i, j : integer;
     flagwidth, stripewidth, Ypos : real;
     Stripe, canton : POLYPOINT;
     LowLimit, Position, StepRight, StepDown : POINT;
     StarSeg : SEGMENT;
     StarXform, SaveXform : SEGXFORM;
     InsertStar : boolean;
     Xinc, Yinc : real;

   BEGIN
     Xinc := 0.4;
     Yinc := 0.3065;
     LowLimit := outline[3];
     FlagWidth := 10 + ABS (Ycoord (LowLimit));
     StripeWidth := FlagWidth / 13;
     Ypos := -10;
     Attributes of fill <- (interior is 'SOLID';
                            colour is 1);
     Create FlagSeg
        Begin
          Line at OutLine;
          For i := 1 to 7 DO
              Begin
                Stripe := (-10, Ypos; 10, Ypos;
                           10, Ypos + StripeWidth;
                          -10, Ypos + StripeWidth);
                Fill at Stripe;
                Ypos := Ypos + (2 * StripeWidth);
```

```
            End;
          Canton := (-10, 10; -1.2, 10; -1.2, (7 * StripeWidth);
                     -10, (7 * StripeWidth));
          Fill at Canton;
          CreateStar (true, StarSeg);
          Position := (-10 + Xinc , 10 - Yinc);
          StarXForm := Scale 0.3 from origin +
                       Shift position;
          InsertStar := true;
          StepRight  := (Xinc, 0);
          StepDown   := (Yinc, 0);
          For j := 1 to 9 Do
              Begin
                SaveXform := StarXform;
                For i := 1 to 11 Do
                    Begin
                      If InsertStar
                      Then Insert StarSeg with StarXform;
                      InsertStar := not InsertStar;
                      StarXform := StarXform +
                                   Shift StepRight;
                    End;
                 StarXform := SaveXform +
                              Shift StepDown;
              End;
           Text 'U.S.A.' at (-6, -8);
        End;
    END; { DoUSA }


PROCEDURE DoFrance (OutLine : POLYPOINT; Var FlagSeg: SEGMENT);
   { Define the national flag of France }

   VAR
     StripeWidth : real;
     VertIncre, TwoByTwo    : POINT;

   Begin
     TwoByTwo := (2, 2);
     StripeWidth := 20 / 3;
     VertIncre := (StripeWidth, 0);
     Create FlagSeg
        Begin
          Line at Outline;
          Attributes of fill <- (interior is 'solid';
                                 colour is 1);
          Fill at ( Outline[1]; Outline[1] + VertIncre;
                    Outline[4] + VertIncre; Outline[4]);
          Attributes of Fill <- (interior is 'pattern';
                                 StyleType is 'grey';
                                 Patternstart is origin;
```

```
                                PatternSize is TwoByTwo);
          Fill at (Outline[2]; Outline[3];
                   Outline[3] - VertIncre;
                   Outline[2] - VertIncre);
          Text 'France' at (-6, -8);
        End;
    END; { DoFrance }


PROCEDURE DoEngland (OutLine : POLYPOINT; Var FlagSeg: SEGMENT);
    { Define the national flag of England }
    VAR
      LowLimit, Center, Yincre1, Yincre2, LeftSide, RightSide : Point;
      Midy, angle : real;
      Stripe, Cross : SEGMENT;
      CrossXform, StripeXform : SEGXFORM;
      i : integer;

    BEGIN
      LowLimit := Outline[4];
      Midy := Ycoord(lowlimit) / 2;
      Center := (0, Midy);
      Yincre1 := (0, 1);
      Yincre2 := (0, 2);
      LeftSide := (-10, MidY);
      RightSide := (10, MidY);

      Create Stripe
        Begin
          Attributes of Fill <- (interior is 'solid';
                                 colour is 0);
          Fill at (Leftside + Yincre2; Rightside + Yincre2;
                   Rightside - Yincre2; Leftside - Yincre2);
          Attributes of Fill <- (colour is 1);
          Fill at (Leftside + Yincre1; Rightside + Yincre1;
                   Rightside - Yincre1; Leftside - Yincre1);
        End;

      Create Cross
        Begin
          Fill at (center; 0, MidY - 0.5; -10,
                   Ycoord(lowlimit) - 0.5; lowlimit);
        End;

      Create FlagSeg
        Begin
          Attributes of Fill <- (interior is 'solid';
                                 colour is 1);
          Line at OutLine;
          Attributes of Fill <- (colour is 0);
          Fill at (Outline[2] + Yincre1; Outline[2] - Yincre1;
```

```
                Outline[4] - Yincre1; Outline[4] + Yincre1);
        Fill at (Outline[1] + Yincre1; Outline[1] - Yincre1;
                Outline[3] - Yincre1; Outline[3] + Yincre1);
        CrossXform := '';
        For i := 1 to 4 Do
            Begin
              Insert Cross with CrossXform;
              Angle := 0.5 * Pi;
              CrossXform := CrossXform + Rotate Angle
                            about Center;
            End;
        StripeXform := '';
        Insert stripe with StripeXform;
        StripeXform := StripeXform + Rotate Pi about Center;
        Text 'England' at (-6, -8);
      End;
  END; { DoEngland }


PROCEDURE DoChina (OutLine : POLYPOINT; Var FlagSeg: SEGMENT);
  { Define the national flag of China }

  VAR
    i : integer;
    Position: SEGXFORM;
    Angle, Incre : real;

  BEGIN
    Attributes of Fill <- (interior is 'solid');
    Create FlagSeg
      Begin
        Fill at OutLine;
        Position := Scale 2.1 from Origin + Shift (-7, 4.42);
        Insert Star with Position;
        Angle := 0.25 * Pi;
        Incre := -0.1875 * Pi;
        Position := Scale 0.53 from Origin + Shift (-4, 4.42)
                    + Rotate Angle about (-7, 4.42);
        For i := 1 to 4 Do
            Begin
              Insert Star with Position;
              Position := Position + Rotate Angle about
                          (-7, 4.42);
            End;
        Text 'China' at (-6, -8);
      End;
  END;  { DoChina }


PROCEDURE DoGermany (OutLine : POLYPOINT; Var FlagSeg: SEGMENT);
  { Define the national flag of Germany }
```

```
   VAR
     FlagWidth, BandWidth : Real;
     LowLimit, TwoByTwo : Point;

   BEGIN
     LowLimit := Outline[3];
     FlagWidth := 10 + ABS (Ycoord (LowLimit));
     BandWidth := FlagWidth / 3;
     Create FlagSeg
        Begin
          Line at Outline;
          Attributes of Fill <- (interior is 'solid';
                                 colour is 1);
          TwoByTwo := (2,2);
          Fill at (-10, 10; 10, 10; 10, 10 - BandWidth;
                   -10, 10 - BandWidth);
          Attributes of Fill <- (interior is 'Pattern';
                                 StyleType is 'grey';
                                 PatternSize is TwoByTwo;
                                 PatternStart is origin);
          Fill at (-10, 10 - BandWidth; 10, 10 - BandWidth;
                    10, 10 - (2 * BandWidth);
                   -10, 10 - (2 * BandWidth) );
          Text 'Germany' at (-6, -8);
        End;
   END;


PROCEDURE DoUSSR (OutLine : POLYPOINT; Var FlagSeg: SEGMENT);
   { Define the national flag of the U.S.S.R. }

   VAR
     StarSeg : SEGMENT;
     StarPosition : POINT;
     StarXform    : SEGXFORM;

   STATIC
     Hammer, Cycle : POLYPOINT;

   VALUE
     Hammer := (-7.5, 6.5; -7.3, 6.7; -8, 7.4; -7.8, 7.5;
                -8.2, 7.8; -8.5, 7.5; -8.4, 7.2; -8.3, 7.3);

   BEGIN
     { ComputeCycle (cycle);   }
     CreateStar (false, StarSeg);
     StarPosition := (-8,  9);
     StarXform := Scale 0.5 from Origin +
                  Shift StarPosition;
     Create FlagSeg
```

```
        Begin
          Attributes of Fill <- (interior is 'solid';
                                 colour is 1);
          Fill at Outline;
          Insert StarSeg with StarXform;
          Attributes of Fill <- (colour is 0);
          Fill at Hammer;
          { Fill at Cycle; }
          Text 'U.S.S.R.' at (-6, -8);
        End;
    END;   { DoUSSR }


PROCEDURE DoJapan (OutLine : POLYPOINT; Var FlagSeg: SEGMENT);
    { Define the national flag of Japan }

    VAR
      Circle : POLYPOINT;

    BEGIN
      Create FlagSeg
        Begin
          Line at OutLine;
          ComputeArc (origin, 5, 0, 2 * Pi, true, 1,
                      100, circle);
          Attributes of Fill <- (interior is 'solid');
          Fill at Circle;
          Text 'Japan' at (-6, -8);
        End;
    END;   { DoJapan }


BEGIN
  Clipping <- 'on';
  Attributes of Text <- (height is 20.0; expansion is 1.5);
  Grey[1,1] := 0;
  Grey[1,2] := 1;
  Grey[2,1] := 1;
  Grey[2,2] := 0;
  GSetPatternRep (Printer, 1, 2, 2, Grey);
  DetermineSize (Canada, Flagsize);
  DoCanada (FlagSize, NatnArr[Canada].Flag);
  DetermineSize (USA, Flagsize);
  DoUSA (FlagSize, NatnArr[USA].Flag);
  DetermineSize (England, Flagsize);
  DoEngland (FlagSize, NatnArr[England].Flag);
  DetermineSize (China, Flagsize);
  DoChina (FlagSize, NatnArr[China].Flag);
  DetermineSize (France, Flagsize);
  DoFrance (FlagSize, NatnArr[France].Flag);
  DetermineSize (Germany, Flagsize);
```

```
    DoGermany (FlagSize, NatnArr[Germany].Flag);
    DetermineSize (USSR, Flagsize);
    DoUSSR (FlagSize, NatnArr[USSR].Flag);
    DetermineSize (Japan, Flagsize);
    DoJapan (FlagSize, NatnArr[Japan].Flag);
  END;    { GenerateFlags }


PROCEDURE GeneratePositions (Var NatnArr : NationArray);
  { This routine generates the transformations to
    later be applied to the flags previously generated.
    Transformations are also stored in the NatnArr along
    with the related flag segment to which they will
    be applied. }

  Var
    i             : nation;
    angle, incre  : real;
    InitLocatn    : SEGXFORM;

  BEGIN
    InitLocatn := ROTATE -0.5 ABOUT Origin +
                  TRANSLATE (40, 0);
    Angle      := 0.125 * Pi;
    Incre      := 0.25 * Pi;
    For i := Canada to Japan Do
        Begin
          InitLocatn := InitLocatn +
                        ROTATE Angle ABOUT origin;
          NatnArr[i].position := InitLocatn;
          If i = China
          Then Begin
                 Angle      := 0.125 * Pi;
                 InitLocatn := ROTATE -0.5 ABOUT Origin +
                               TRANSLATE (-40, 0) +
                               ROTATE Angle ABOUT Origin;
               End  { I = 4 }
          Else Angle := Angle + Incre;
        End;  { For Stmt }
  END;    { GeneratePositions }


PROCEDURE DisplayExhibit (NatnArr : NationArray);
  { This routine generates the graphical output
    using the segments and transformation previously
    defined.                                          }

  VAR
    ExhibitUnits, NDCUnits : WINDOW;
    ExhibitNT              : NORMXFORM;
    Exhibit                : SEGMENT;
```

```
    i                       : nation;

BEGIN
      For I := Canada to Japan Do
          Begin
            Display NatnArr[i].flag to Printer;
            Clear Printer;
          End;
      ExhibitUnits := (-100, -65; 100, 65);
      NDCUnits     := (0.0681, 0.09375;
                       0.9318, 0.84275);
      ExhibitNT    := FROM ExhibitUnits  TO NDCUnits;
      TRANSFORM NT <- ExhibitNT;

      Create Exhibit;
      Begin
        For i := Canada to Japan Do
            Insert NatnArr[i].flag with
                   NatnArr[i].position;
        Attributes of Text <- (height is 130.0;
                               expansion is 2.0;
                               spacing is 0.5);
        Text 'PEACE' at (-10, 30);
        Text 'ON'    at ( -5, 0 );
        Text 'EARTH' at (-10, -30);
      End;  { Create Seg }
    Display Exhibit to Printer;
END;        { DisplayExhibit }


BEGIN   { main }
  Pi := 3.141592;
  GenerateFlags (ArrayOfNations);
  GeneratePositions (ArrayOfNations);
  DisplayExhibit (ArrayOfNations);
END.
```

```
program ezgks_demo1 ;
 {*********************************************************************}
 {**  This is the program translated into GKS subroutine calls.      **}
 {*********************************************************************}

      %Print Off
%Include GKS_BASICS
%Include GKS_SEGMENTS
%Print On

const
  GXCMaxSetSize       =  31;
  GXCMaxBundleSetSize =  10;
  GXCMaxATrVal        = 50;
  GXCMaxAtrTypes      = 26;

type
  GXTPrimitive  = ( line, marker, fillarea, graphicText);

  GXTSetRange   =  0..GXCMaxSetSize;

  GXTEZSet      =  packed set of GXTSetRange;

  Wkstn  =  GXTSetRange;

  Point  =  GRPoint;

  GXTPolyPointRange =  0..GCMaxPoint;

  PolyPoint  =  record
     points      : GAPointArray;
     max         : GXTPolyPointRange;
  End;

  Colour  =  record
     masterindex  : GTint0;
     colour       : GRcol;
  End;

  Pattern  =  record
     masterindex  : GTint0;
     RowLength,
     ColumnLength : integer;
     Pattern      : GAColArray;
     ColourSource : GAColArray;
       { 0 = colour table; 1 = MWT; }
  End;

  Bundle  =  record
     masterindex    :  GTint0;
     AttrInit       :  Array [1..4] of boolean;
```

```
      ColourSource    :  Integer;
        {0 = colour table; 1 = MWT }
      Attributes      :  GRPrimRep;
    End;

  Segment  =  GXTSetRange;

  WkstnSet  =  GXTEZSet;

  SegmentSet =  GXTEZSet;

  Window  =  record
     validPercent : boolean;
     corner       : GRbound;
  End;

  NormXform  =  GTint1;

  SegXform   =  GAMatrix;

  GXTWsOperation = (GXSave, GXRestore);

  GXTTypeSave = (GXOnly, GXAlso);

  GXTWkstnSetType = (GXOpenWs, GXActiveWs);

  GXTAtrValRec = record
    AtrName : string;
    AtrValue: integer;
    DefeultFlag : integer;
    End;

  GXTAvArrayType = array [1..GXCMaxAtrVal] of GXTAtrValRec;

  GXTContextLimit = record
     Min, Max : integer;
    End;

  GXTCtxArrayType = array [0..GXCMaxAtrTypes] of GXTContextLimit;

const
  red           =  colour (0, (1,0,0));
  green         =  colour (0, (0,1,0));
  blue          =  colour (0, (0,0,1));
  yellow        =  colour (0, (1,1,0));
  magenta       =  colour (0, (1,0,1));
  cyan          =  colour (0, (0,1,1));
  black         =  colour (0, (0,0,0));
  white         =  colour (0, (1,1,1));
  origin        =  point  (0.0, 0.0);
  CentralSegStore =  wkstn  (0);
```

```
  identity     =  segxform ( ( 1.0, 0.0, 0.0),
                             (0.0, 1.0, 0.0));

var
        GXVpoint, GXVpoint2 : point;
        GXVactiveWkstns, GXVopenWkstns : wkstnset;
        GXVpolypoint    : polypoint;
        GXVcolour       : colour;
        GXVpattern      : pattern;
        GXVbundle       : bundle;
        GXVsegment      : segment;
        GXVwkstnset     : wkstnset;
        GXVsegmentset   : segmentset;
        GXVwindow       : window;
        GXVnormxform    : normxform;
        GXVsegxform     : segxform;
        GXVfontprec     : GRFontPrec;
        GXValign        : GRalign;
        GXVvector       : GAVector;
        GXVstring       : GAstring;
        GXVi, GXVj      : integer;
        GXVr1, GXVr2    : real;

        GXVAsf          : GAasf;
        GXVAvArray      : GXTAVArrayType;
        GXVLimitArray   : GXTCtxArrayType;
        GXVAtr          : text;

      Function GXSetMax (CONST NbrRange, NbrPts : integer)
                         : integer;
          External;
      Procedure GXSetPt (CONST PtSet : polypoint;
                         VAR   Pt : point) ;
         External;
      Procedure GXSetWin (CONST PtSet : polypoint;
                          VAR   Win   : window );
         External;
      Function GXNTNum : integer;
         External;
      Function GXSegNum : integer;
         External;
      Function GXAddPT (CONST PTa, PTb : point) : point;
         External;
      Function GXSubPT (CONST PTa, PTb : point) : point;
         External;
      Function GXWsSet (CONST typeset : GXTWkstnSetType) : wkstnset;
         External;
      Procedure GXSuppR (Var changedSet : WkstnSet);
         External;
      Procedure GXRestR (Const changedset : WkstnSet);
         External;
```

```
Procedure GXStChg (op : GXTWsOperation;
          SaveMethod : GXTTypeSave; SaveSet : WkstnSet);
   External;
Procedure GXState  (op : GXTWsOperation;
          WkstnsNamed : Wkstnset; SegSetOp : boolean);
   External;
Procedure GXGetASF (Var asfFlags : GAAsf;
          ASFtype : GTasf; PrimNbr : integer);
   External;
Procedure GXInitAv (CONST maxAtr, MaxCtx : integer;
          Var AVArray : GXTAvArrayType;
          Var LimitArray: GXTCtxArrayType);
   External;
Function GxAtrVal (CONST AtrArray : GXTAvArrayType;
   CONST min, max: integer; CONST searchname : string): integer;
   External;
Procedure GXColExp (CONST processflag : boolean;
          CONST NbrParts : integer;
          CONST InColour : GRcol;
          Var ExprColour : GRcol);
   External;
Procedure GXAccum (VAR Xform1 : segxform;
                   CONST Xform2 : segxform);
   External;
Function Xcoord (CONST Pt : point) : real;
   External;
Function Ycoord (CONST Pt : point) : real;
   External;
Function LoLeft (CONST win1 : window) :point;
   External;
Function LoRight (CONST win1 : window) :point;
   External;
Function UpLeft (CONST win1 : window) :point;
   External;
Function UpRight (CONST win1 : window) :point;
   External;
Function RedVal (CONST ColRec : colour): real;
   External;
Function GreenVal (CONST ColRec : colour): real;
   External;
Function BlueVal (CONST ColRec : colour): real;
   External;
Function MaxPts (CONST PolyPtRec : polypoint): integer;
   External;
Function XMax (CONST PolyPtRec : polypoint): real;
   External;
Function XMin (CONST PolyPtRec : polypoint): real;
   External;
Function YMax (CONST PolyPtRec : polypoint): real;
   External;
Function YMin (CONST PolyPtRec : polypoint): real;
```

```
   External;
Function GetPt (CONST PolyPtRec : polypoint;
                CONST index : integer): point;
   External;
Function GetWin (CONST NTNbr : normxform): window;
   External;
Function GetVP (CONST NTNbr : normxform): window;
   External;
Function RowLen (CONST PattRec : pattern): integer;
   External;
Function ColLen (CONST PattRec : pattern): integer;
   External;
const
  printer = wkstn (  1) ;


type
  nationrec = record
    flag : segment ;
    position : segxform ;
    end
  ;

  nation = ( canada , usa , england , china , france , germany ,
  ussr , japan ) ;

  nationarray = array [ canada .. japan ] of nationrec ;


var
  arrayofnations : nationarray ;
  star : segment ;
  pi : real ;




procedure generateflags ( var natnarr : nationarray ) ;


  var
    grey : gacolarray ;
    flagsize : polypoint ;



  procedure computearc ( const center : point ; const radius , alpha , beta : real ;
    const clockwise : boolean ; const initialindex , nbrpts :
    integer ; var result : polypoint ) ;
```

```
  var
    i : integer ;
    temp , incre : real ;
    holdpt : point ;


    begin
    result.max := 0;
    if alpha >= beta
    then if clockwise
    then incre := - ( alpha - beta ) / ( nbrpts -  1 )

    else incre := (  2 * pi - ( alpha - beta ) ) /
    ( nbrpts -  1 )

    else if clockwise
    then incre := - (  2 * pi - ( beta - alpha ) ) /
    ( nbrpts -  1 )

    else incre := ( alpha - beta ) / ( nbrpts -  1 ) ;

    for i := 0 to nbrpts -  1 do

        begin
        temp := alpha + i * incre ;
        GXVpolypoint.points [ 1].x := xcoord(center)+radius*cos(temp) ;
        GXVpolypoint.points [ 1].y := ycoord(center)+radius*sin(temp);
        GXVpolypoint.max := 1;
        GXSetPT (GXVPolyPoint, GXVPoint);
        holdpt := GXVpoint;
        result.points[i+initialindex] := holdpt;
        If i+initialindex > result.max
        Then result.max := i+initialindex;

      end
    ;

  end ;


procedure determinesize ( name : nation ; var outline : polypoint ) ;


  const
    length =  20 ;


  type
```

```
    aspectarray = array [ canada .. japan ] of real ;

  static
    aspectratios : aspectarray ;

  value
    aspectratios := aspectarray ( 0.5 , 0.5128 , 0.5428 , 0.67857 ,
    0.6428 , 0.6129 , 0.57142 , 0.6428 ) ;

  var
    bottom : real ;


    begin
    GXVpolypoint.points [ 1].x := - 10 ;
    GXVpolypoint.points [ 1].y :=  10;
    GXVpolypoint.points [ 2].x :=  10 ;
    GXVpolypoint.points [ 2].y :=  10;
    GXVpolypoint.max :=  2;
    outline := GXVpolypoint;
    bottom := ( length * aspectratios [ name ] ) -  10 ;
    GXVpolypoint.points [ 1].x :=  10 ;
    GXVpolypoint.points [ 1].y := bottom;
    GXVpolypoint.max :=  1;
    outline.points[ 3] := GXVpolypoint.points [1];
    If  3 > outline.max
    Then outline.max :=  3;
    GXVpolypoint.points [ 1].x := - 10 ;
    GXVpolypoint.points [ 1].y := bottom;
    GXVpolypoint.max :=  1;
    outline.points[ 4] := GXVpolypoint.points [1];
    If  4 > outline.max
    Then outline.max :=  4;
    outline.points[ 5] := outline.points[ 1] ;
    If  5 > outline.max
    Then outline.max :=  5;

  end ;


procedure createstar ( const fillstar : boolean ; var starseg : segment ) ;


  var
    starunits : window ;
    starnt : normxform ;
```

```
  static
    starpoints : polypoint ;


  value
    starpoints := polypoint ( GAPointArray (
    GRPoint ( 0.95 , -0.3 ) ,
    GRpoint ( 0.0 , 1.0 ) ,
    GRpoint ( -0.95 , -0.3 ) ,
    GRpoint ( 0.95 , 0.3 ) ,
    GRpoint ( -0.95 , 0.3 ) ,
    GRpoint ( 0.95 , -0.3 ) ,
    GRPoint (0.0, 0.0) :  94) ,            6 );


    begin
    starseg := GXSegNum;
    GCreateSeg ( starseg );

      begin
      GSetFillColInd (0);
      GXVpolypoint.points [ 1].x := - 1 ;
      GXVpolypoint.points [ 1].y := - 1;
      GXVpolypoint.points [ 2].x :=  1 ;
      GXVpolypoint.points [ 2].y :=  1;
      GXVpolypoint.max :=  2;
      GXSetWin (GXVpolypoint, GXVWindow);
      starunits := GXVwindow;
      starnt := GXNTNum;
      GSetWindow (starnt, starunits.corner );
      GSelectNTran ( starnt);
      if fillstar
      then GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')))
      else GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'hollow')))
      GXVi := GXSetMax (0, starpoints.max);
      GFill ( GXVi, starpoints.points ) ;

    end ;
    GCloseSeg;

  end ;


procedure docanada ( outline : polypoint ; var flagseg : segment ) ;


  static
    mapleleaf : polypoint ;
```

```
value
  mapleleaf := polypoint ( GAPointArray (
  GRPoint ( -0.3 , -10.0 ) ,
  GRpoint ( -0.3 , -2.4 ) ,
  GRpoint ( -3.3 , -4.3 ) ,
  GRpoint ( -3.3 , -3.3 ) ,
  GRpoint ( -5.8 , -3.3 ) ,
  GRpoint ( -4.4 , -1.5 ) ,
  GRpoint ( -5.8 , -1.2 ) ,
  GRpoint ( -4.0 , 0.0 ) ,
  GRpoint ( -7.9 , 2.9 ) ,
  GRpoint ( -6.9 , 2.7 ) ,
  GRpoint ( -8.3 , 4.6 ) ,
  GRpoint ( -5.8 , 4.6 ) ,
  GRpoint ( -6.3 , 5.9 ) ,
  GRpoint ( -2.0 , 3.3 ) ,
  GRpoint ( -2.0 , 7.9 ) ,
  GRpoint ( -1.0 , 7.4 ) ,
  GRpoint ( 0.0 , 9.4 ) ,
  GRPoint (0.0, 0.0) :  83) ,          17 );


var
  temppt : point ;
  temppolypt : polypoint ;



  begin
  flagseg := GXSegNum;
  GCreateSeg ( flagseg );

    begin
    GXVi := GXSetMax (0, outline.max);
    GPolyline ( GXVi, outline.points ) ;
    GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
    GXVi := GXSetMax (0, mapleleaf.max);
    GFill ( GXVi, mapleleaf.points ) ;
    GXVpolypoint.points [ 1].x :=  5 ;
    GXVpolypoint.points [ 1].y := 0;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
    temppt := GXVpoint;
    GXVpolypoint.points [ 1] := outline.points[ 1] ;
    GXVpolypoint.points [ 2] := GXAddPT (outline.points[ 1] , temppt);
    GXVpolypoint.points [ 3] := GXAddPT (outline.points[ 4] , temppt);
    GXVpolypoint.points [ 4] := outline.points[ 4] ;
    GXVpolypoint.max :=  4;
    temppolypt := GXVPolyPoint ;
    GXVi := GXSetMax (0, temppolypt.max);
```

```
      GFill ( GXVi, temppolypt.points ) ;
      GXVpolypoint.points [ 1] := GXSubPT (outline.points[ 2] , temppt);
      GXVpolypoint.points [ 2] := outline.points[ 2] ;
      GXVpolypoint.points [ 3] := GXSubPT (outline.points[ 3] , temppt);
      GXVpolypoint.points [ 4] := outline.points[ 3] ;
      GXVpolypoint.max :=  4;
      temppolypt := GXVPolyPoint ;
      GXVi := GXSetMax (0, temppolypt.max);
      GFill ( GXVi, temppolypt.points ) ;
      GXVstring := 'Canada' ;
      GXVpolypoint.points [ 1].x := - 6 ;
      GXVpolypoint.points [ 1].y := - 8;
      GXVpolypoint.max :=  1;
      GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

    end ;
    GCloseSeg;

  end ;


procedure dousa ( outline : polypoint ; var flagseg : segment ) ;


  var
    i , j : integer ;
    flagwidth , stripewidth , ypos : real ;
    stripe , canton : polypoint ;
    lowlimit , position , stepright , stepdown : point ;
    starseg : segment ;
    starxform , savexform : segxform ;
    insertstar : boolean ;
    xinc , yinc : real ;



    begin
    xinc := 0.4 ;
    yinc := 0.3065 ;
    lowlimit := outline.points[ 3] ;
    flagwidth :=  10 + abs ( ycoord ( lowlimit ) ) ;
    stripewidth := flagwidth /  13 ;
    ypos := -  10 ;
    GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'SOLID')));
    GSetFillColInd ( 1);
    flagseg := GXSegNum;
    GCreateSeg ( flagseg );

      begin
      GXVi := GXSetMax (0, outline.max);
```

```
GPolyline ( GXVi, outline.points ) ;
for i :=  1 to  7 do

    begin
    GXVpolypoint.points [ 1].x := - 10 ;
    GXVpolypoint.points [ 1].y := ypos;
    GXVpolypoint.points [ 2].x :=  10 ;
    GXVpolypoint.points [ 2].y := ypos;
    GXVpolypoint.points [ 3].x :=  10 ;
    GXVpolypoint.points [ 3].y := ypos+stripewidth;
    GXVpolypoint.points [ 4].x := - 10 ;
    GXVpolypoint.points [ 4].y := ypos+stripewidth;
    GXVpolypoint.max :=  4;
    stripe := GXVpolypoint;
    GXVi := GXSetMax (0, stripe.max);
    GFill ( GXVi, stripe.points ) ;
    ypos := ypos + (  2 * stripewidth ) ;

  end
;
GXVpolypoint.points [ 1].x := - 10 ;
GXVpolypoint.points [ 1].y :=  10;
GXVpolypoint.points [ 2].x := -1.2 ;
GXVpolypoint.points [ 2].y :=  10;
GXVpolypoint.points [ 3].x := -1.2 ;
GXVpolypoint.points [ 3].y := ( 7*stripewidth);
GXVpolypoint.points [ 4].x := - 10 ;
GXVpolypoint.points [ 4].y := ( 7*stripewidth);
GXVpolypoint.max :=  4;
canton := GXVpolypoint;
GXVi := GXSetMax (0, canton.max);
GFill ( GXVi, canton.points ) ;
createstar ( true , starseg ) ;
GXVpolypoint.points [ 1].x := - 10+xinc ;
GXVpolypoint.points [ 1].y :=  10-yinc;
GXVpolypoint.max :=  1;
GXSetPT (GXVPolyPoint, GXVPoint);
position := GXVpoint;
GXVSegXform := Identity;
GXVPoint := origin;
GXVvector[1] := 0.3;
GXVvector[2] := 0.3;
GAccumTran (GXVSegxform , GXVPoint, origin, 0,
  GXVvector, NDC, GXVSegXform);
GXVvector[1] := 1.0;
GXVvector[2] := 1.0;
GXVPoint := position;
Gaccumtran (GXVSegXform, Origin, GXVPoint, 0,
  GXVvector, WC, GXVSegxform);
starxform := GXVsegxform;
insertstar := true ;
```

```
GXVpolypoint.points [ 1].x := xinc ;
GXVpolypoint.points [ 1].y := 0;
GXVpolypoint.max :=  1;
GXSetPT (GXVPolyPoint, GXVPoint);
stepright := GXVpoint;
GXVpolypoint.points [ 1].x := yinc ;
GXVpolypoint.points [ 1].y := 0;
GXVpolypoint.max :=  1;
GXSetPT (GXVPolyPoint, GXVPoint);
stepdown := GXVpoint;
for j :=  1 to  9 do

    begin
    GXVSegXform := Identity;
    GXAccum (GXVSegxform, starxform);
    savexform := GXVsegxform;
    for i :=  1 to  11 do

        begin
        if insertstar
        then GXVSegXForm := starxform;
        GInsertSeg (starseg, GXVSegXForm);
        insertstar := not insertstar ;
        GXVSegXform := Identity;
        GXAccum (GXVSegxform, starxform);
        GXVvector[1] := 1.0;
        GXVvector[2] := 1.0;
        GXVPoint := stepright;
        Gaccumtran (GXVSegXform, Origin, GXVPoint, 0,
          GXVvector, WC, GXVSegxform);
        starxform := GXVsegxform;

      end
    ;
    GXVSegXform := Identity;
    GXAccum (GXVSegxform, savexform);
    GXVvector[1] := 1.0;
    GXVvector[2] := 1.0;
    GXVPoint := stepdown;
    Gaccumtran (GXVSegXform, Origin, GXVPoint, 0,
      GXVvector, WC, GXVSegxform);
    starxform := GXVsegxform;

  end
;
GXVstring := 'U.S.A.' ;
GXVpolypoint.points [ 1].x := - 6 ;
GXVpolypoint.points [ 1].y := - 8;
GXVpolypoint.max :=  1;
GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );
```

```
    end ;
    GCloseSeg;

  end ;


procedure dofrance ( outline : polypoint ; var flagseg : segment ) ;


  var
    stripewidth : real ;
    vertincre , twobytwo : point ;



    begin
    GXVpolypoint.points [ 1].x :=  2 ;
    GXVpolypoint.points [ 1].y :=  2;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
    twobytwo := GXVpoint;
    stripewidth :=  20 /  3 ;
    GXVpolypoint.points [ 1].x := stripewidth ;
    GXVpolypoint.points [ 1].y := 0;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
    vertincre := GXVpoint;
    flagseg := GXSegNum;
    GCreateSeg ( flagseg );

      begin
      GXVi := GXSetMax (0, outline.max);
      GPolyline ( GXVi, outline.points ) ;
      GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
      GSetFillColInd ( 1);
      GXVpolypoint.points [ 1] := outline.points[ 1] ;
      GXVpolypoint.points [ 2] := GXAddPT (outline.points[ 1] , vertincre);
      GXVpolypoint.points [ 3] := GXAddPT (outline.points[ 4] , vertincre);
      GXVpolypoint.points [ 4] := outline.points[ 4] ;
      GXVpolypoint.max :=  4;
      GXVi := GXSetMax (0,  4);
      GFill ( GXVi, GXVpolypoint.points ) ;
      GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'pattern')));
      GSetFillStyleInd (GXAtrVal (GXVAVArray, GXVLimitArray [ 14].min, GXVLimitArray [ 14].max, 'grey'));
      GSetPatternRefPoint (origin);
      GXVvector[1] := twobytwo.x;
      GXVvector[2] := twobytwo.y;
      GSetPatternSize (GXVvector);
      GXVpolypoint.points [ 1] := outline.points[ 2] ;
      GXVpolypoint.points [ 2] := outline.points[ 3] ;
```

```
      GXVpolypoint.points [ 3] := GXSubPT (outline.points[ 3] , vertincre);
      GXVpolypoint.points [ 4] := (GXSubPT (outline.points[ 2] , vertincre));
      GXVpolypoint.max :=  4;
      GXVi := GXSetMax (0,  4);
      GFill ( GXVi, GXVpolypoint.points ) ;
      GXVstring := 'France' ;
      GXVpolypoint.points [ 1].x := - 6 ;
      GXVpolypoint.points [ 1].y := - 8;
      GXVpolypoint.max :=  1;
      GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

    end ;
    GCloseSeg;

  end ;


procedure doengland ( outline : polypoint ; var flagseg : segment ) ;

  var
    lowlimit , center , yincre1 , yincre2 , leftside , rightside : point ;
    midy , angle : real ;
    stripe , cross : segment ;
    crossxform , stripexform : segxform ;
    i : integer ;


    begin
    lowlimit := outline.points[ 4] ;
    midy := ycoord ( lowlimit ) /  2 ;
    GXVpolypoint.points [ 1].x := 0 ;
    GXVpolypoint.points [ 1].y := midy;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
    center := GXVpoint;
    GXVpolypoint.points [ 1].x := 0 ;
    GXVpolypoint.points [ 1].y :=  1;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
    yincre1 := GXVpoint;
    GXVpolypoint.points [ 1].x := 0 ;
    GXVpolypoint.points [ 1].y :=  2;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
    yincre2 := GXVpoint;
    GXVpolypoint.points [ 1].x := - 10 ;
    GXVpolypoint.points [ 1].y := midy;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
```

```
leftside := GXVpoint;
GXVpolypoint.points [ 1].x :=  10 ;
GXVpolypoint.points [ 1].y := midy;
GXVpolypoint.max :=  1;
GXSetPT (GXVPolyPoint, GXVPoint);
rightside := GXVpoint;

stripe := GXSegNum;
GCreateSeg ( stripe );

  begin
  GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
  GSetFillColInd (0);
  GXVpolypoint.points [ 1] := GXAddPT (leftside, yincre2);
  GXVpolypoint.points [ 2] := GXAddPT (rightside, yincre2);
  GXVpolypoint.points [ 3] := GXSubPT (rightside, yincre2);
  GXVpolypoint.points [ 4] := (GXSubPT (leftside, yincre2));
  GXVpolypoint.max :=  4;
  GXVi := GXSetMax (0,  4);
  GFill ( GXVi, GXVpolypoint.points ) ;
  GSetFillColInd ( 1);
  GXVpolypoint.points [ 1] := GXAddPT (leftside, yincre1);
  GXVpolypoint.points [ 2] := GXAddPT (rightside, yincre1);
  GXVpolypoint.points [ 3] := GXSubPT (rightside, yincre1);
  GXVpolypoint.points [ 4] := (GXSubPT (leftside, yincre1));
  GXVpolypoint.max :=  4;
  GXVi := GXSetMax (0,  4);
  GFill ( GXVi, GXVpolypoint.points ) ;

end ;
GCloseSeg;

cross := GXSegNum;
GCreateSeg ( cross );

  begin
  GXVpolypoint.points [ 1] := center;
  GXVpolypoint.points [ 2].x := 0 ;
  GXVpolypoint.points [ 2].y := midy-0.5;
  GXVpolypoint.points [ 3].x := - 10 ;
  GXVpolypoint.points [ 3].y := ycoord(lowlimit)-0.5;
  GXVpolypoint.points [ 4] := lowlimit;
  GXVpolypoint.max :=  4;
  GXVi := GXSetMax (0,  4);
  GFill ( GXVi, GXVpolypoint.points ) ;

end ;
GCloseSeg;

flagseg := GXSegNum;
GCreateSeg ( flagseg );
```

```
begin
GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
GSetFillColInd ( 1);
GXVi := GXSetMax (0, outline.max);
GPolyline ( GXVi, outline.points ) ;
GSetFillColInd (0);
GXVpolypoint.points [ 1] := GXAddPT (outline.points[ 2] , yincre1);
GXVpolypoint.points [ 2] := GXSubPT (outline.points[ 2] , yincre1);
GXVpolypoint.points [ 3] := GXSubPT (outline.points[ 4] , yincre1);
GXVpolypoint.points [ 4] := (GXAddPT (outline.points[ 4] , yincre1));
GXVpolypoint.max :=  4;
GXVi := GXSetMax (0,  4);
GFill ( GXVi, GXVpolypoint.points ) ;
GXVpolypoint.points [ 1] := GXAddPT (outline.points[ 1] , yincre1);
GXVpolypoint.points [ 2] := GXSubPT (outline.points[ 1] , yincre1);
GXVpolypoint.points [ 3] := GXSubPT (outline.points[ 3] , yincre1);
GXVpolypoint.points [ 4] := (GXAddPT (outline.points[ 3] , yincre1));
GXVpolypoint.max :=  4;
GXVi := GXSetMax (0,  4);
GFill ( GXVi, GXVpolypoint.points ) ;
GXVSegXform := Identity;
crossxform:= GXVSegXform;
for i :=  1 to  4 do

    begin
    GXVSegXForm := crossxform;
    GInsertSeg (cross, GXVSegXForm);
    angle := 0.5 * pi ;
    GXVSegXform := Identity;
    GXAccum (GXVSegxform, crossxform);
    GXVvector[1] := 1.0;
    GXVvector[2] := 1.0;
    GXVPoint := center;
    GAccumtran (GXVSegXform, GXVPoint, origin,angle,
      GXVvector, WC, GXVSegxform);
    crossxform := GXVsegxform;

  end
;
GXVSegXform := Identity;
stripexform:= GXVSegXform;
GXVSegXForm := stripexform;
GInsertSeg (stripe, GXVSegXForm);
GXVSegXform := Identity;
GXAccum (GXVSegxform, stripexform);
GXVvector[1] := 1.0;
GXVvector[2] := 1.0;
GXVPoint := center;
GAccumtran (GXVSegXform, GXVPoint, origin,pi,
  GXVvector, WC, GXVSegxform);
```

```
      stripexform := GXVsegxform;
      GXVstring := 'England' ;
      GXVpolypoint.points [ 1].x := - 6 ;
      GXVpolypoint.points [ 1].y := - 8;
      GXVpolypoint.max :=  1;
      GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

    end ;
    GCloseSeg;

  end ;


procedure dochina ( outline : polypoint ; var flagseg : segment ) ;


  var
    i : integer ;
    position : segxform ;
    angle , incre : real ;


    begin
    GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
    flagseg := GXSegNum;
    GCreateSeg ( flagseg );

      begin
      GXVi := GXSetMax (0, outline.max);
      GFill ( GXVi, outline.points ) ;
      GXVSegXform := Identity;
      GXVPoint := origin;
      GXVvector[1] := 2.1;
      GXVvector[2] := 2.1;
      GAccumTran (GXVSegxform , GXVPoint, origin, 0,
        GXVvector, NDC, GXVSegXform);
      GXVvector[1] := 1.0;
      GXVvector[2] := 1.0;
      GXVPoint.x := -7;
      GXVPoint.y := 4.42;
      Gaccumtran (GXVSegXform, Origin, GXVPoint, 0,
        GXVvector, WC, GXVSegxform);
      position := GXVsegxform;
      GXVSegXForm := position;
      GInsertSeg (star, GXVSegXForm);
      angle := 0.25 * pi ;
      incre := - 0.1875 * pi ;
      GXVSegXform := Identity;
      GXVPoint := origin;
```

```
      GXVvector[1] := 0.53;
      GXVvector[2] := 0.53;
      GAccumTran (GXVSegxform , GXVPoint, origin, 0,
        GXVvector, NDC, GXVSegXform);
      GXVvector[1] := 1.0;
      GXVvector[2] := 1.0;
      GXVPoint.x := -4;
      GXVPoint.y := 4.42;
      Gaccumtran (GXVSegXform, Origin, GXVPoint, 0,
        GXVvector, WC, GXVSegxform);
      GXVvector[1] := 1.0;
      GXVvector[2] := 1.0;
      GXVPoint.x := -7;
      GXVPoint.y := 4.42;
      GAccumtran (GXVSegXform, GXVPoint, origin,angle,
        GXVvector, WC, GXVSegxform);
      position := GXVsegxform;
      for i :=  1 to  4 do

          begin
          GXVSegXForm := position;
          GInsertSeg (star, GXVSegXForm);
          GXVSegXform := Identity;
          GXAccum (GXVSegxform, position);
          GXVvector[1] := 1.0;
          GXVvector[2] := 1.0;
          GXVPoint.x := -7;
          GXVPoint.y := 4.42;
          GAccumtran (GXVSegXform, GXVPoint, origin,angle,
            GXVvector, WC, GXVSegxform);
          position := GXVsegxform;

        end
      ;
      GXVstring := 'China' ;
      GXVpolypoint.points [ 1].x := - 6 ;
      GXVpolypoint.points [ 1].y := - 8;
      GXVpolypoint.max :=  1;
      GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

    end ;
    GCloseSeg;

  end ;


procedure dogermany ( outline : polypoint ; var flagseg : segment ) ;

  var
```

```
flagwidth , bandwidth : real ;
lowlimit , twobytwo : point ;


begin
lowlimit := outline.points[ 3] ;
flagwidth :=  10 + abs ( ycoord ( lowlimit ) ) ;
bandwidth := flagwidth /  3 ;
flagseg := GXSegNum;
GCreateSeg ( flagseg );

  begin
  GXVi := GXSetMax (0, outline.max);
  GPolyline ( GXVi, outline.points ) ;
  GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
  GSetFillColInd ( 1);
  GXVpolypoint.points [ 1].x :=  2 ;
  GXVpolypoint.points [ 1].y :=  2;
  GXVpolypoint.max :=  1;
  GXSetPT (GXVPolyPoint, GXVPoint);
  twobytwo := GXVpoint;
  GXVpolypoint.points [ 1].x := - 10 ;
  GXVpolypoint.points [ 1].y :=  10;
  GXVpolypoint.points [ 2].x :=  10 ;
  GXVpolypoint.points [ 2].y :=  10;
  GXVpolypoint.points [ 3].x :=  10 ;
  GXVpolypoint.points [ 3].y :=  10-bandwidth;
  GXVpolypoint.points [ 4].x := - 10 ;
  GXVpolypoint.points [ 4].y :=  10-bandwidth;
  GXVpolypoint.max :=  4;
  GXVi := GXSetMax (0,  4);
  GFill ( GXVi, GXVpolypoint.points ) ;
  GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'Pattern')));
  GSetFillStyleInd (GXAtrVal (GXVAVArray, GXVLimitArray [ 14].min, GXVLimitArray [ 14].max, 'grey'));
  GXVvector[1] := twobytwo.x;
  GXVvector[2] := twobytwo.y;
  GSetPatternSize (GXVvector);
  GSetPatternRefPoint (origin);
  GXVpolypoint.points [ 1].x := - 10 ;
  GXVpolypoint.points [ 1].y :=  10-bandwidth;
  GXVpolypoint.points [ 2].x :=  10 ;
  GXVpolypoint.points [ 2].y :=  10-bandwidth;
  GXVpolypoint.points [ 3].x :=  10 ;
  GXVpolypoint.points [ 3].y :=  10-( 2*bandwidth);
  GXVpolypoint.points [ 4].x := - 10 ;
  GXVpolypoint.points [ 4].y :=  10-( 2*bandwidth);
  GXVpolypoint.max :=  4;
  GXVi := GXSetMax (0,  4);
  GFill ( GXVi, GXVpolypoint.points ) ;
  GXVstring := 'Germany' ;
```

```
      GXVpolypoint.points [ 1].x := - 6 ;
      GXVpolypoint.points [ 1].y := - 8;
      GXVpolypoint.max :=  1;
      GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

    end ;
    GCloseSeg;

  end ;


procedure doussr ( outline : polypoint ; var flagseg : segment ) ;


  var
    starseg : segment ;
    starposition : point ;
    starxform : segxform ;


  static
    hammer , cycle : polypoint ;


  value
    hammer := polypoint ( GAPointArray (
    GRPoint ( -7.5 , 6.5 ) ,
    GRpoint ( -7.3 , 6.7 ) ,
    GRpoint ( -8.0 , 7.4 ) ,
    GRpoint ( -7.8 , 7.5 ) ,
    GRpoint ( -8.2 , 7.8 ) ,
    GRpoint ( -8.5 , 7.5 ) ,
    GRpoint ( -8.4 , 7.2 ) ,
    GRpoint ( -8.3 , 7.3 ) ,
    GRPoint (0.0, 0.0) :  92) ,            8 );



    begin

    createstar ( false , starseg ) ;
    GXVpolypoint.points [ 1].x := - 8 ;
    GXVpolypoint.points [ 1].y :=  9;
    GXVpolypoint.max :=  1;
    GXSetPT (GXVPolyPoint, GXVPoint);
    starposition := GXVpoint;
    GXVSegXform := Identity;
    GXVPoint := origin;
    GXVvector[1] := 0.5;
    GXVvector[2] := 0.5;
```

```
    GAccumTran (GXVSegxform , GXVPoint, origin, 0,
      GXVvector, NDC, GXVSegXform);
    GXVvector[1] := 1.0;
    GXVvector[2] := 1.0;
    GXVPoint := starposition;
    Gaccumtran (GXVSegXform, Origin, GXVPoint, 0,
      GXVvector, WC, GXVSegxform);
    starxform := GXVsegxform;
    flagseg := GXSegNum;
    GCreateSeg ( flagseg );

      begin
      GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
      GSetFillColInd ( 1);
      GXVi := GXSetMax (0, outline.max);
      GFill ( GXVi, outline.points ) ;
      GXVSegXForm := starxform;
      GInsertSeg (starseg, GXVSegXForm);
      GSetFillColInd (0);
      GXVi := GXSetMax (0, hammer.max);
      GFill ( GXVi, hammer.points ) ;

      GXVstring := 'U.S.S.R.' ;
      GXVpolypoint.points [ 1].x := - 6 ;
      GXVpolypoint.points [ 1].y := - 8;
      GXVpolypoint.max :=  1;
      GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

    end ;
    GCloseSeg;

  end ;


procedure dojapan ( outline : polypoint ; var flagseg : segment ) ;

  var
    circle : polypoint ;


    begin
    flagseg := GXSegNum;
    GCreateSeg ( flagseg );

      begin
      GXVi := GXSetMax (0, outline.max);
      GPolyline ( GXVi, outline.points ) ;
      computearc ( origin ,  5 , 0 ,  2 * pi , true ,  1 ,
```

```
        100 , circle ) ;
        GSetFillIntStyle (GTinterior (GXAtrVal (GXVAVArray, GXVLimitArray [ 13].min, GXVLimitArray [ 13].max, 'solid')));
        GXVi := GXSetMax (0, circle.max);
        GFill ( GXVi, circle.points ) ;
        GXVstring := 'Japan' ;
        GXVpolypoint.points [ 1].x := - 6 ;
        GXVpolypoint.points [ 1].y := - 8;
        GXVpolypoint.max :=  1;
        GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

      end ;
      GCloseSeg;

    end ;



    begin
    ;
    GSetCharHeight (20.0);
    GSetCharExpan (1.5);
    grey[ 1, 1] := 0;
    grey[ 1, 2] := 1;
    grey[ 2, 1] := 1;
    grey[2, 2] := 0;
    gsetpatternrep ( printer ,  1 ,  2 ,  2 , grey ) ;
    determinesize ( canada , flagsize ) ;
    docanada ( flagsize , natnarr [ canada ] . flag ) ;
    determinesize ( usa , flagsize ) ;
    dousa ( flagsize , natnarr [ usa ] . flag ) ;
    determinesize ( england , flagsize ) ;
    doengland ( flagsize , natnarr [ england ] . flag ) ;
    determinesize ( china , flagsize ) ;
    dochina ( flagsize , natnarr [ china ] . flag ) ;
    determinesize ( france , flagsize ) ;
    dofrance ( flagsize , natnarr [ france ] . flag ) ;
    determinesize ( germany , flagsize ) ;
    dogermany ( flagsize , natnarr [ germany ] . flag ) ;
    determinesize ( ussr , flagsize ) ;
    doussr ( flagsize , natnarr [ ussr ] . flag ) ;
    determinesize ( japan , flagsize ) ;
    dojapan ( flagsize , natnarr [ japan ] . flag ) ;

  end ;


procedure generateposition ( var natnarr : nationarray ) ;
```

```
var
  i : nation ;
  angle , incre : real ;
  initlocatn : segxform ;



  begin
  GXVSegXform := Identity;
  GXVvector[1] := 1.0;
  GXVvector[2] := 1.0;
  GXVPoint := origin;
  GAccumtran (GXVSegXform, GXVPoint, origin,0.5,
    GXVvector, WC, GXVSegxform);
  initlocatn := GXVsegxform;
  angle := 0.125 * pi ;
  incre := 0.25 * pi ;
  for i := canada to japan do

      begin
      GXVSegXform := Identity;
      GXAccum (GXVSegxform, initlocatn);
      GXVvector[1] := 1.0;
      GXVvector[2] := 1.0;
      GXVPoint := origin;
      GAccumtran (GXVSegXform, GXVPoint, origin,angle,
        GXVvector, WC, GXVSegxform);
      initlocatn := GXVsegxform;
      GXVSegXform := Identity;
      GXAccum (GXVSegxform, initlocatn);
      natnarr[i].position := GXVsegxform;
      if i = china
      then
        begin
        angle := 0.125 * pi ;
        GXVSegXform := Identity;
        GXVvector[1] := 1.0;
        GXVvector[2] := 1.0;
        GXVPoint := origin;
        GAccumtran (GXVSegXform, GXVPoint, origin,0.5,
          GXVvector, WC, GXVSegxform);
        GXVvector[1] := 1.0;
        GXVvector[2] := 1.0;
        GXVPoint := origin;
        GAccumtran (GXVSegXform, GXVPoint, origin,angle,
          GXVvector, WC, GXVSegxform);
        initlocatn := GXVsegxform;

      end

      else angle := angle + incre ;
```

```
      end
    ;

  end ;


procedure displayexhibit ( natnarr : nationarray ) ;


  var
    exhibitunits , ndcunits : window ;
    exhibitnt : normxform ;
    exhibit : segment ;
    i : nation ;



    begin
    for i := canada to japan do

        begin
        GXState ( GXsave, [printer], false);
        GCopySegWs (printer, natnarr[i].flag);
        GXState ( GXrestore, [printer], false);
        GXState ( GXsave, [printer], false);
        GClearWs (printer, ClearCond );
        GXState ( GXrestore, [printer], false);

      end
    ;
    GXVpolypoint.points [ 1].x := - 100 ;
    GXVpolypoint.points [ 1].y := - 65;
    GXVpolypoint.points [ 2].x :=  100 ;
    GXVpolypoint.points [ 2].y :=  65;
    GXVpolypoint.max :=  2;
    GXSetWin (GXVpolypoint, GXVWindow);
    exhibitunits := GXVwindow;
    GXVpolypoint.points [ 1].x := 0.0681 ;
    GXVpolypoint.points [ 1].y := 0.09375;
    GXVpolypoint.points [ 2].x := 0.9318 ;
    GXVpolypoint.points [ 2].y := 0.84275;
    GXVpolypoint.max :=  2;
    GXSetWin (GXVpolypoint, GXVWindow);
    ndcunits := GXVwindow;
    exhibitnt := GXNTNum;
    GSetWindow (exhibitnt, exhibitunits.corner );
    GSetViewport (exhibitnt, ndcunits.corner );
    GSelectNTran ( exhibitnt);
```

```
  exhibit := GXSegNum;
  GCreateSeg ( exhibit );
  ;
  GCloseSeg;

    begin
    for i := canada to japan do
      GXVSegXForm := natnarr[i].position;
      GInsertSeg (natnarr[i].flag, GXVSegXForm)
    ;
    GSetCharHeight (130.0);
    GSetCharExpan (2.0);
    GSetCharSpacing (0.5);
    GXVstring := 'PEACE' ;
    GXVpolypoint.points [ 1].x := - 10 ;
    GXVpolypoint.points [ 1].y :=  30;
    GXVpolypoint.max :=  1;
    GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );
    GXVstring := 'ON' ;
    GXVpolypoint.points [ 1].x := - 5 ;
    GXVpolypoint.points [ 1].y := 0;
    GXVpolypoint.max :=  1;
    GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );
    GXVstring := 'EARTH' ;
    GXVpolypoint.points [ 1].x := - 10 ;
    GXVpolypoint.points [ 1].y := - 30;
    GXVpolypoint.max :=  1;
    GTextString ( GXVPolyPoint.points [1] , length(GXVString), GXVString );

  end ;
  GXState ( GXsave, [printer], false);
  GCopySegWs (printer, exhibit);
  GXState ( GXrestore, [printer], false);

end ;



begin
GopenGKS (0,1000);
GopenWs (CentralSegStore, 0,  3) ;
GActivateWS (CentralSegStore);
GopenWs (  1,  21,  7370);
GXInitAv (GXCMaxAtrVal, GXCMaxAtrTypes, GXVAvarray, GXVLimitArray);
pi := 3.141592 ;
generateflags ( arrayofnations ) ;
generateposition ( arrayofnations ) ;
displayexhibit ( arrayofnations ) ;

GXVopenWkstns := GXWsSet (GXopenWs);
```

```
        GXVactiveWkstns := GXWsSet (GXActiveWs);
        For GXVi := 0 to GXCMaxSetSize DO
          Begin
            If GXVi IN GXVActiveWkstns
            Then GDeactivateWS (GXVi);
            If GXVi IN GXVopenWkstns
            Then GcloseWs (GXVi);
          End;
        GcloseGKS;
end .
```