17636

NATIONAL LIBRARY
OTTAWA

CANADA

BIBLIOTHÈQUE NATIONALE
OTTAWA

NAME OF AUTHOR.......NAM NG

TITLE OF THESIS...AN ENVIRONMENT - INDEPENDENT

......GRAPHICS FACILITY

....

UNIVERSITY........U OF ALBERTA

DEGREE FOR WHICH THESIS WAS PRESENTED.....Ph. D

YEAR THIS DEGREE GRANTED.........1973

Permission is hereby granted to THE NATIONAL LIBRARY

OF CANADA to microfilm this thesis and to lend or sell copies

of the film.

The author reserves other publication rights, and

neither the thesis nor extensive extracts from it may be

printed or otherwise reproduced without the author's

written permission.

(Signed)................

PERMANENT ADDRESS:

Flat 836 Hoy An Lau,

30 Aberdeen Reservoir Road

Aberdeen, Hong Kong.

DATED....Aug 31......1973

NL-91 (10-68)

THE UNIVERSITY OF ALBERTA

AN ENVIRONMENT-INDEPENDENT GRAPHICS FACILITY

by

Ⓒ NAM NG

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1973

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and
recommend to the Faculty of Graduate Studies and Research,
for acceptance, a thesis entitled "AN ENVIRONMENT-
INDEPENDENT GRAPHICS FACILITY", submitted by NAM NG
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

..........*T.A. Marsland*..........
Supervisor

..........*Barry J. Mailloux*..........

..........*W.K. Dawson*..........

..........*J.M. ...*..........

..........*Alan C. Shaw*..........
External Examiner

Date .....*Aug. 14, 1973*.....

ABSTRACT

A graphics language has been designed and implemented to provide graphics facilities to the applications programmer. The language is strictly graphics-oriented, but its graphics operations are general. It includes the definition of a picture description scheme which can be used to describe large classes of pictures. A technique is developed which allows the graphics software to be accessed from a number of high-level host languages without modification of the graphics software or the host language. Details of a current implementation are given. An approach for extending the picture description scheme to picture analysis is also described.

ACKNOWLEDGMENT

TABLE OF CONTENTS

LIST OF FIGURES

x

Chapter I

Introduction

To provide graphics software for general use, it is important that it be "environment-independent". Existing graphics software and systems are often display-hardware and host dependent. Furthermore, Nake[26] has noted that "...there are still some unifying concepts to be developed" in graphics languages. It was the intent of this research to design an environment-independent graphics facility, and to specify and implement a graphics language which is elegant and powerful.

Hardware is the most restrictive environment. Often, graphics software is tailored to certain display hardware so that more can be displayed on the screen. However, display hardware has improved in the last five years. Fast-executing display files have become less crucial. But, because of hardware-dependency, the software produced often results in wasted efforts as hardware changes. Therefore, it is important that the design principles of graphics software be independent of hardware changes if such software is to remain of value for some period of time.

Host dependency is another area that has been overlooked by many researchers. There is no doubt that many graphics languages (e.g. see survey by Williams[43]) have been

produced. However, the general user's response is usually one of dismay. One of the major obstacles is that either the facility is tailored and restricted to a certain host language, in which case only some users will appreciate its utility; or it is self-contained, in which case it demands learning another language which is definitely to the distaste of the casual user.

It should be noted that this thesis is not an attempt to produce a super graphics language. It is impractical, if not impossible, to do so. Rather, it is intended to demonstrate a philosophy for creating a practical and general graphics facility.

With existing high-level languages, it is felt that there are sufficient non-graphics capabilities, and there is no need to duplicate these efforts. Graphics software is therefore viewed as a general facility of a computer system, in a way not much different from software packages for handling ordinary input/output devices. Only operations that are inherent in graphics have to be provided. On its own, the software is sufficient to produce pictures of all kinds. When interfaced with other high-level languages, the graphics operations appear to the user as built-in facilities of the system.

One obvious approach is to extend the compilers of the high-level languages. However, their very diversity

excludes the practicability of modifying all of them and their compilers to accommodate graphics, at least for the present moment.

At the University of Alberta, a system has been implemented which allows the graphics language to be used in conjunction with existing high-level languages. The graphics statements are suitably flagged so that they can be distinguished easily from the host statements. The program is scanned by a preprocessor, which detects the flagged statements, converts them into host procedure calls and thus provides the linkage to the graphics software. In this process, it is necessary to ensure that communication between the two languages, which is usually in the form of transfer of values, be properly established. The different conventions of program linkages require that the graphics software be built in a linkage-convention-independent manner. It is hence the responsibility of the preprocessor, in the scanning process, to build the necessary interface to provide proper linkages.

The main advantage of this approach is that only a single graphics software package has to be built for universal use among a number of high-level languages. The overhead costs include the preprocessing time for building the interface, and the linkage time in transferring control between the two languages. However, there is a further advantage in this

approach. The facilities available in the host language also reduce the capabilities required in the graphics language, and only operations which are inherent in picture description and input/output need be catered to. This avoids re-inventing features that are already present in the host languages. More time can then be devoted to the development of a language that is general, as far as graphics operations are concerned. With such a facility, users will be able to use graphics in the host language of their own choice.

Not only should the graphics software be universal, it is also important that it be user-oriented. Because of the graphics-operations-alone requirement, it is possible to build a graphics language that is relatively simple. A powerful yet simple graphics language has been defined and is presented in this thesis. It has five types of statements, which include a picture description scheme, device-independent input and output, and facilities for picture transformation and information retrieval. A large class of pictures can be described in the language. It allows one to build pictures from subpictures and basic drawing primitives. Transformations within picture definitions are allowed. No output code for display is generated when the pictures are defined; hence, subpictures need not be defined before they are used in the definition of other pictures. Thus, the user can describe his pictures

in the manner that is most natural to him. Labelling of picture components, which is necessary for identification purposes in interactive graphics, is done by sequential indexing, without explicit user specification.

Pictures are called by names, and defined by their components and corresponding attributes. A tree-like data structure is maintained for the pictures as they are built, and the user has control over every component of the pictures, which thus can be easily modified. The data structure together with hardware-related drawing primitives make easy to output the pictures on any device at any time.

To be really hardware independent, however, more than just a data-structure is required. Since most graphics systems do not possess all the necessary hardware, the organization of display files must be specified in a hardware-independent manner. Usually, display files are either structured or sequential. Here, a mixed strategy is used to ensure uniform treatment of display files independent of the availability of different hardware. The scheme is basically structured, and remains so as long as the hardware allows. If a certain picture transformation cannot be achieved by the available display hardware, then a sequential display description is used. A picture vector, as defined by Duffin,[6] is generated for that picture so that

it can be easily manipulated by software. On the other hand, if a new hardware feature is added to the system, then the sequential display description is replaced by a structured description. In this way, the basic construction philosophy of the display file remains the same no matter what hardware is available. If a purely structured or sequential construction is required for any particular hardware, the technique is still applicable.

Another important feature of this language lies in the uniformity of picture description, independent of the source of graphic information. Input from the display terminal is treated the same as a picture definition using the picture description scheme, the data structure being updated correspondingly. The language is thus input-device independent.

To summarize, the major contribution of this thesis lies in the design and implementation of a general graphics language. The picture description scheme is simple but versatile. The place of a graphics language in relation to existing high-level languages is also identified. This is important since it strongly influences the design of the language. By constructing it to be hardware and host independent, a general and environment-independent graphics facility is achieved.

Organization of the Thesis

In Chapter II, a survey of the development of graphics languages is given. The advantages and disadvantages of the different approaches to provide graphics software are discussed. The major contribution of the thesis--the design of a new graphics language--is described in Chapter III. The philosophy of the design, aimed at providing a practical graphics facility for the general user, is first discussed. The features of the language are then presented, with details of the simple but versatile picture description scheme, the data structure, and the construction of efficient and hardware-independent display files. Chapter IV describes another important contribution--the provision of a universal graphics facility. The technique used for providing such a facility for use among different high-level languages is discussed. Specific examples using PL/I and ALGOL W as hosts are given. Implementation of the facility with FORTRAN as the host language is then described in Chapter V. Implementation details, including techniques to provide good operating efficiency of the system, are described. The nature of the implementation is such that extension to other host languages presents no fundamental problems. Demonstration of the working system is also given. In Chapter VI, the suitability of the picture description scheme for picture analysis is discussed. Summary of results and work for future research are given in

the final chapter, including suggestions of how the
efficiency of the system can be improved.

Chapter II

History and Development of Graphics Languages

Since Sutherland[40] first used "...a cathode ray tube display and light pen to draw pictures while monitoring user's motion (using the light pen) and building a structured set of data representing the data being drawn", there has been ever-growing interest in the area of computer graphics. Graphics hardware has developed from the initial cathode-ray-tube display and light pen to the present-day plasma displays, liquid-crystal displays, laser displays, three-dimensional input devices and many others. In parallel with hardware development, graphics software has also been a major area of research. Different approaches to the development of software have resulted in diversified areas of interest. Languages for description and generation of pictures, techniques of handling data structures and man-machine communication are some of the frequent areas of concern. While many useful ideas have been conceived in the different areas, there are limitations in the different approaches which are discussed in the following sections.

2.1. Extension of Existing High-level Languages to Accommodate Graphics

In this approach, either the graphics software is provided as subroutine packages of certain high-level

languages (usually FORTRAN) or the high-level language is modified to include graphical operations. In the first category, the software is usually available as library routines. Examples are CALCOMP[2], GSP[12], and GRIDSUB[17]. There is no explicit picture description scheme -- pictures are formed by making appropriate calls to the package, and the picture-data structure is not accessible to the user. These packages are best described as utility packages, since they do provide some utility to the graphics users.

In the second category, existing high-level languages are modified or extended to include graphics operations. Hurwitz et al.[14] have first modified FORTRAN and introduced the use of display variables. There is a picture description scheme which allows the user to describe pictures in terms of display functions, which are either built-in or user-defined. Again, there is no user-accessible picture-data structure, and little consideration has been given to the construction of display files and hardware dependency of the software.

Later work has extended graphics into languages other than FORTRAN. Newman[29] has introduced ALGOL-like display procedures in generating information for display. Transformation within calls to these procedures allows simpler description of pictures. In trying to overcome display-hardware dependency he has proposed the use of

sequential display files. But, as he has noted, "the principal complaint leveled against display procedures is that they are inefficient. Whenever a minor change is made to a picture, the entire frame must be regenerated". The scheme is also not suitable for remote display terminals because of the large quantity of data that has to be transmitted each time a change is made. Nevertheless, the picture description scheme is a marked improvement over the previous ones. It is indeed more flexible to generate and modify pictures with display procedures.

In extending PL/I to include graphics, Smith[38] has also developed some useful concepts in defining pictures. He has suggested that "...images (or pictures) should be defined in terms of concepts rather than specific hardware commands". In his picture description scheme, images can be combined to form new images using image operators. Five operators -- inclusion, connection, positioning, scaling and rotation -- are provided for the description of pictures, but no provision is made for easy extension to other operators. Smith has also considered the building of image data. "Image data structures should be included in the language in such a way that the details of the structure are hidden from the user and so that the structural mechanism may be readily changed". The interrupt capability of PL/I has also been extended to process display device interrupts. It is evident that this system relies heavily on the use of some

features of PL/I.

The main drawback of this approach is that an appreciable amount of work is required to modify the compilers of the high-level languages. The software is also restricted to the high-level language involved.

2.2. Application-Oriented Picture Description Languages

In this approach, the aim is to provide means of describing and generating pictures for certain applications. An explicit picture description scheme is usually provided. Frank[8] invented the B-line for describing and manipulating his drawings. Applications are mainly for production of drawings for publication purposes. Some graphical arithmetic facilities are also provided. The language is embedded in FORTRAN. A macro processor is used to convert the graphics statements into FORTRAN statements before compilation. Mezei's SPARTA[23] is a procedure-oriented language for manipulation of arbitrary line drawings. The software is again available in FORTRAN. Application is mainly for computer art productions. Various mathematical transformations are permitted. A follow-up work by Duffin[6] has produced a language for line drawings in APL. Pictures are considered to be vectors formed by their coordinates and attributes. Good arithmetic capabilities are provided for different transformations of the pictures. However, the APL notation is rather awkward since expressions are evaluated

from right to left.

The usual drawback (and advantage) of this approach is that the language developed is application-oriented. Very little can be done outside the application areas. Often, the software is also host dependent.

2.3. Data Structure Languages for Graphics Systems

Since data structures are important in interactive systems, much work has been done on data structures for general-purpose graphics systems. There are low-level languages such as L6 [19], DSPL[41], CORAL[40], and ASP[20]. There are also high-level languages such as Associative Programming Language[5] and LEAP[33]. Detailed surveys of such languages can be found in Gray[11] and Williams[43]. Researchers in this area believe that good data structures are mandatory for problem modelling, relational and structural descriptions, and hence graphics systems. However, the diversified application areas in graphics require data structures that are not easily predetermined. In an effort to overcome such problems, Williams[44] has designed an extensible general purpose graphical language which allows users to define their own data types and operators. FORTRAN has been used as the host language.

While a good data structure could be useful for certain applications, it is often not necessary to provide complex

data structuring mechanisms, especially when graphics facilities are provided in another high-level language. Applications such as map storage and retrieval[4] and computer art[23] need only simple data structures, e.g. sequential description of data, and simple graphical transformations. Little or no reference to complex data structures may be required.

The main disadvantage of this approach is that the picture description schemes become more and more obscure as the data structure grows in complexity. The average user may never use such complex facilities. Besides, the facility still has to depend on the use of a high-level host language and much programming effort is duplicated.

2.4.   Linguistic Approach to Picture Description and Generation

By analogy with the linguistic approach to programming languages, formalized descriptions of pictures have also been developed. The target here is to provide descriptions of pictures for analysis and recognition. Narasimhan[27] was among the pioneers in this approach. The Bubble Chamber Analysis and the language for describing English alphabets are well known[28]. Although the picture description scheme is quite restricted, it has provided a good start in this area. Kirsch[18] used a two-dimensional approach to describe right triangles. The scheme however becomes much too

involved for complex pictures. Ledley's description of chromosomes[21] is also well posed, but as Miller and Shaw[25] have pointed out, "It is difficult to generalize this approach to figures other than closed curves...". Shaw's PDL[34] language has provided some spark in the field. A good formalism is developed for the language, and it has been applied to both picture analysis[35][36] and generation[9]. One drawback is that a picture can only be concatenated at two specific points, its head and tail. It is often necessary to break down a picture into many undesired small pieces before it can be described. Feder[7] has extended the number of concatenation points by allowing the user to specify them when the pictures are defined, but this can become very clumsy if many connection points are required.

No doubt many of these picture description schemes and their formalization are good. While such formalism has so far found little use in interactive graphics, development in this direction could be highly beneficial.

2.5. Discussion

In all these approaches, some good picture description schemes and data structures are provided. However, they often cater to only one aspect of graphics. There is also a lack of generality in some of the previously mentioned systems. The language are usually hardware dependent, one way or another, especially when they are tailored to provide

efficiency on certain display hardware. With rapidly changing display hardware, the software could be out of date before it gets implemented. Thus, it can be seen that some careful planning is needed to provide a good graphics facility. A global consideration of the different aspects is mandatory. The design and implementation of a general graphics facility is described in the next few chapters.

Chapter III

The Graphics Language

3.1. Philosophy of the design

The design of a graphics language is not much different from that of other computer languages. While some general criteria do apply, e.g. completeness, orthogonality, etc., there are other unique requirements in its design. Before defining the language, the philosophy of how the graphics facility is to be provided must be clearly specified.

The graphics language which has been designed and implemented is intended primarily to provide a versatile interactive graphics facility for the applications programmer. It has also been the aim of the project to implement the graphics facility in such a way that it can be accessed from more than one high-level programming language.

The most involved consideration is the provision of a picture description scheme. It is obvious that host-language-dependent description schemes are no longer applicable. An important criterion is that the scheme should be as simple and natural as possible. Often, too much emphasis has been placed on developing data structures that can model certain problems, while neglecting the more important aspect -- the pictorial aspect -- of a graphics language. Although a data structuring mechanism for

describing pictures should be maintained internally in the computer, it should be the pictures, and not data structure, with which the user has to deal. The user should be able to think of pictures as pictures, and not as pointers, blocks, etc. The structuring mechanism should be transparent to those users who do not care to use it, but accessible to those who want to manipulate it.

A powerful but elegant picture description scheme is desirable. An ability to describe the pictures as one sees them is important. It gives the user a sense of actuality. A rotated square, say, should be described as a rotated square, or as a square rotated a certain amount, and not as four points in space that have no obvious geometrical relations. The user should be able to structure his pictures at will, and manipulate the picture geometrically. Basic picture manipulation functions, such as scaling, rotation, windowing, translation, etc., should be available in the language. Facilities for identification of objects or pictures displayed on the screen should be incorporated. Information associated with any picture should be readily accessible.

In view of the rapidly changing display hardware and the diversified graphics system configurations, it is also desirable that the graphics software be as hardware-independent as possible. Where it is appropriate, the

software specification should be general enough for different system configurations.

One other aspect that was given careful consideration is the problem of man-machine interaction. Good man-machine communication is essential in developing any interactive system. This criterion also applies to a graphics system. Here, the interaction occurs mainly at the display terminal, where the user is responding to pictures displayed on the screen. Convenient means of communicating with the graphics system would be helpful for the user. It is the responsibility of the display supervisor and the interactive section of the graphics language to ensure that this criterion is met.

3.2. The Graphics Language

With the above design objectives, a graphics language has been defined. The language basically consists of five statements, namely, the Picture Definition Statement, the DRAW Statement, the DECODE Statement, the RETRIEVE Statement, and the FUNCTION Declaration Statement. A formal definition of the syntax of the language is given in Appendix I. Detailed descriptions of the capabilities of the different statements are given in the following sections.

3.2.1. The Picture Definition Statement

The picture definition statement is used for describing and manipulating pictures. In essence, it provides the picture description scheme. Under this scheme, pictures are constructed from subpictures and drawing primitives. The drawing primitives have been chosen to be hardware-oriented but independent of any particular hardware. The choice has two important implications:

1. Since the drawing primitives are closely related to hardware, it is easy to output the pictures on most graphical devices.

2. Since the drawing primitives are independent of any specific hardware, it is possible to use the same picture description for different graphical devices.

The drawing primitives include:

(i)   P -- which denotes a point

(ii) V -- which denotes a vector

(iii) S -- which denotes a symbol, i.e., the alpha-numeric and special characters

(iv) L -- which denotes a sequence of vectors

(v)   T -- which denotes a sequence of symbols.

It should be noted that on most graphical devices these primitives are easily provided via software, if not already available in the hardware. They are also the most frequently used, and are sufficient to describe most

pictures.

In this scheme, pictures are given names -- the picture names, and are defined by their components. Each component can be either a "valuated" picture or a drawing primitive. A valuated picture is a picture with its attributes and transformations specified. The general format of a valuated picture is

<pictname>(<x>,<y>,<pos>,<lpdet>,<blink>,<blank>)*<transform>
where

    <pictname> specifies the picture name,

    <x> and <y> are the X and Y coordinates,

    <pos>   specifies absolute or relative positioning,

    <lpdet>   specifies the light-pen detectability of the picture,

    <blink>   specifies whether the picture is to be shown blinking,

    <blank>   specifies whether the picture is to be blanked,

    <transform> specifies the transformations to be applied.

The attributes of valuated pictures and primitives can be divided into two categories. In the first category, the positional information of the picture is provided. Assuming that the two dimensional Cartesi coordinate system is being used, then this corresponds the X and Y

coordinates. Other coordinate systems, e.g. Polar coordinates, are possible alternatives. The <pos> attribute indicates where the picture component is placed in relation to the origin of the previous component or the origin of the picture being defined, depending on whether the positioning attribute is specified to be relative or absolute. It is always assumed that the drawing device is placed at the origin of the picture before and after drawing it. For a drawing primitive, the drawing device is left at the point where it last finishes drawing. Detailed description of the position attributes are given in Appendix II.

The second category provide list of graphical hardware attributes, including light detectability, blinking or blanking of pictures, solid or dashed plotting of lines, and other appropriate attributes. Primitives have widely different lists of attributes, since they differ widely in nature. For example, the primitives L and T have the attribute <num> which specifies the number of vectors or symbols that are to be plotted. Detailed description of the attributes can be found in Appendix III.

Each attribute can have values 0, 1 or 2. The value 0 implies that the attribute value is unspecified, which is the same as the default value. The value 1 implies that the commonly used attribute value is in effect, such as absolute positioning of pictures (i.e. with respect to the picture

origin), solid plotting of vectors, etc. The value 2 would imply an attribute value opposite to that for the value 1. However, the actual attribute value of a picture or primitive is determined at picture output time, when the picture data structure is evaluated to give the output.

There is a hierarchy for the evaluation of attribute values. An attribute value specified at a higher level of the picture imposes the same attribute value throughout all its subpictures and associated primitives, no matter how the attributes are specified. But, if the value is unspecified, i.e., either the value 0 is specified or the default value is given by leaving it blank, then the assignment of the attribute value is delayed until a subpicture has a specified attribute value, or until the primitives are reached, whereupon the attribute value is set to 1.

This hierarchy of attribute evaluation has the advantage that it is possible to construct pictures that are identical but have different attributes without defining the picture more than once. For example, the following picture definition statements require only one definition for the picture SQUARE:

    PICT1 = SQUARE(X,Y,0,0,2,0)
    PICT2 = SQUARE(U,V,0,0,1,0)

No matter how the picture SQUARE is constructed, PICT1 is a blinking SQUARE, while PICT2 is not. Thus, by specifying

the attribute value at the appropriate level of the picture,
it is possible to describe pictures with different
attributes.

When a picture is defined, each component of the picture
is given an implicit identification by sequential indexing
as the components are added to form the picture. Any
subsequent change to the picture will cause the components
to be re-indexed automatically. Thus, while the user can
refer to the components by the identification, he does not
have to worry about providing indexing or identification to
the pictures.

A tree-like data structure is maintained for the pictures
as they are defined. The structure essentially provides an
internal description of the pictures in the computer.
Whenever a picture definition statement is executed, the
appropriate information is entered into the data structure,
which thus represents the pictures that a user has defined
at any time during execution of his program. It should be
noted that no code or display file is generated at this
stage. It is only when the pictures are output that the
data structure is used for generating the appropriate code.
There are a number of advantages in maintaining a picture
data structure and delaying the generation of output code:

    (1) Since no code is generated until output, it is not

        necessary to describe a picture before it can be

used for defining other pictures. Top-down description of pictures now becomes permissible, which often provides a more natural way to describe pictures. Essentially one can describe the pictures with increasing details, in the way he sees it.

(2) It is not necessary to generate display code for those pictures that are defined but not displayed, which occurs frequently. In such a case, unnecessary code generation is avoided and there would be saving in execution time.

(3) The data structure provides an internal, hardware-independent description of the pictures, and is easy to convert the pictures for output on any graphical device.

(4) The data structure also provides a means of easy identification of picture parts, and retains information that can be retrieved later.

In this picture description scheme, only one operator is provided for the construction of pictures. The '+' operator is used to group the picture components to form higher-level pictures. Since no other operators are provided, no special relations, such as "close to", "on top of", etc., can be specified in the language. However, a good combination of the picture description scheme, the facility for retrieving information about the pictures, together with logical and arithmetic capabilities of the host language will provide

many relations at such a description level.   he user is encouraged to incorporate them as desired.

A special reserved word, NULL, is introduced for deleting pictures.   It serves to denote an empty picture. Assigning this special reserved word to a picture variable in a picture definition statement will cause the picture to be deleted and left empty. This can also be used to delete part of a picture and will be described later.

Transformation of pictures can be specified within the picture definition statement. The '*' operator is used to specify that a transformation is required. For example, the statement:

PICT(X,Y) * SCALE(SX,SY) * ROTATE(THETA)

results in the picture PICT positioned at location (X,Y), then scaled by the factors SX and SY in the X and Y directions respectively, and rotated in the counter-clockwise direction by an angle THETA with respect to the X-axis. It should be noted that the sequence of transformations is left-to-right, in the same order as one would normally describe pictures, thus provides a more user-oriented way of describing pictures in the language. In fact, it is also easier to parse the statement in this manner than in the conventional way of specifying functions.

It is not until output time that the transformations specified in picture definition statements are executed.

Only the data structure is updated to include new information so that the transformations can be correctly carried out when the display code is generated. The identities of the pictures are thus retained, since all the relevant information is included in the data structure. On the other hand, if the transformations were performed at the time the picture definition statement is encountered, it would only result in a new set of points in space which could not be easily identified.

Another important feature of the picture description scheme is that a picture component selection feature is also incorporated, which allows manipulation of every single component of the pictures. Each component of a picture can be referenced, and hence modified, using the special symbol '.' . Thus, PICT.N refers to the N-th component of the picture PICT if N is an integer, or the M-th component if M is the value of the integer variable N in the host language at the time when the variable is evaluated. The component selection operator can appear on the left- or right-hand side of the picture definition statement. When it appears on the left-hand side, it causes that particular picture component to be modified as specified by the right-hand side. When it appears on the right-hand side, it implies that a copy of that picture component is to be used here within the new definition. Each time such a change is encountered, the data structure is also appropriately

updated. If a certain component is assigned to be NULL, that component is simply deleted.

At present, only one level of selection is provided in the language. Although it can in theory be extended to multiple-level selections, a one-level selection is in fact sufficient for most picture description purposes. However, in the identification of pictures via light-pen interrupts, a multiple-level identification vector is necessary to identify each component uniquely. More details will be described in section 3.2.3.

To illustrate the use of the picture description scheme, consider the following picture definition statement:

PICT = PICT1(ATTLIST1) + PICT2(ATTLIST2)

On execution of this statement, the data structure is updated to give the following structure for the picture PICT:

```
                    PICT
                    /\
                   /  \
                  /    \
                 1      2
                /        \
               /          \
    PICT1(ATTLIST1)      PICT2(ATTLIST2)
```
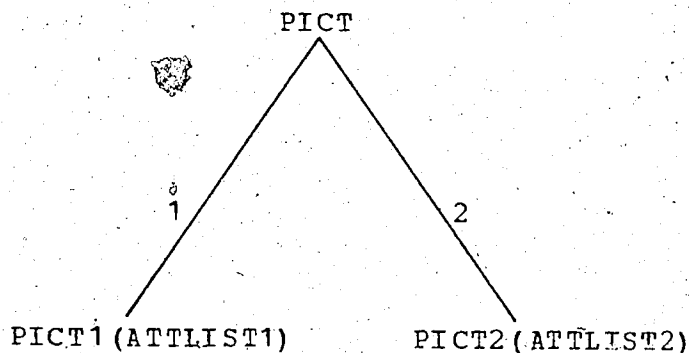
This means that the picture PICT has two components. The first component is a picture PICT1 with attributes ATTLIST1.

The second component is another picture PICT2 with attributes ATTLIST2. The two components are referred to as PICT.1 and PICT.2 respectively.

If the statement

PICT.2 = PICT3(ATTLIST3)

is now executed, the second component of PICT is replaced by the picture PICT3 with attributes ATTLIST3.

If the first component of PICT is to be deleted, it can be achieved by executing

PICT.1 = NULL

leaving PICT3 to be the first and only component of PICT.

It should be noted that executing the statement

PICT.2 = NULL

does not have the same effect as executing

PICT2 = NULL

In the first case, the second component is simply deleted. In the latter case, PICT2 becomes an empty picture but PICT still retains PICT2 as its component.

Some other uses of the picture description scheme are exemplified below under different categories:

(I) Definition of Pictures

    (a) Construction of pictures from primitives:

        SQUARE = V(1,0) + V(1,1) + V(0,1) + V(0,0)

        TRIANGLE = L(3,'10,0; 5,10; 0,0 ')

Notice that in this example the picture SQUARE is constructed from four components, all of the primitive type V, whereas the picture TRIANGLE is constructed from only one component of the primitive type L, although it is made up of three vectors.

(b) Construction of pictures by copying a picture component:

    P1 = PICT.1(X1,Y1)

    Thus if the most recent definition of PICT is

    PICT = SQUARE(XS,YS) + TRIANGLE(XT,YT)

    then P1 has the same definition as if

    P1 = SQUARE(X1,Y1)

(II) Modification of pictures

(a) New definition of pictures:

    PICT = TRIANGLE(X3,Y3)

    would replace the picture PICT by the new definition and its previous definition is lost.

(b) Addition of a picture component to a picture:

    PICT = PICT + TRIANGLE(XNEW,YNEW)

    would cause a new component to be added to the picture PICT. Notice also that this does not increase the depth of the picture tree, but does increase the number of components of the picture PICT.

(c) Modification of a component of a picture:

PICT.1 = SQUARE(X5,Y5)

would replace the first component of the picture PICT by the picture SQUARE.

(III) <u>Deletion of pictures</u>

    (a)   Total deletion:

        PICT = NULL

    (b) Partial deletion:

        PICT.1 = NULL

(IV) <u>Transformation</u>

    (a)   Scaling of a picture:

        PICT = SQUARE(XX,YY)*SCALE(SIZEX,SIZEY)

    (b) Multiple transformation:

        PP = TRIANGLE(XP,YP)*ROTATE(THETA)*SCALE(SX,SY)

So far, a number of examples of the use of the picture description scheme have been shown. By combining the picture description scheme with the capabilities of high-level languages, more complex pictures can be described. Further examples will be given in the next two chapters.

3.2.2. Picture Output

Pictures are output by the DRAW statement. The general
format of the statement is:

DRAW (<output device>, <pict var> (<attri3>))

The first parameter specifies the device on which the
picture will be output. The second parameter specifies the
picture, with its attributes, which is to be output.

Here, the problem of picture generation is not a complex
one, since the data structure is output-device independent
and the primitives are hardware-related. By evaluating the
picture tree in a top-down fashion, it is relatively simple
to generate the code for output on any graphical device.

However, to be able to generate efficient and hardware-
independent display files need some detailed consideration.
There are in general two types of display-file construction.
The first type is the structured one, which is also the most
conventional. It has the advantage of easy code generation
and modification, and pictures can be easily identified. If
many copies of the same picture are required, there would
also be a considerable saving in display-file space.
Fig. 3.1 shows a structured display file. The disadvantage
of this type of display-file construction is that when the
picture to be displayed requires a tranformation that cannot
be performed by the available hardware, another version of

the display file, the transformed display file, has to be used. Under this circumstance, there will be a duplication of effort and use of storage.

The second display file uses a structureless or sequential type of construction. Newman's display procedure[29] is an example of this kind. It has the advantage of being less hardware-dependent than the structured type since practically all displays can use such display files. Fig. 3.2 shows the sequential display file for the same picture as for Fig. 3.1. However, modifications of such display files usually require regenerating the entire file. Although Newman has used picture frames[29] to reduce the regeneration required, it is still not as convenient as the use of structured display files. In addition, identification of picture parts can be a problem.

$$A = B + C$$

$$B = D + E$$

$$C = D + F$$
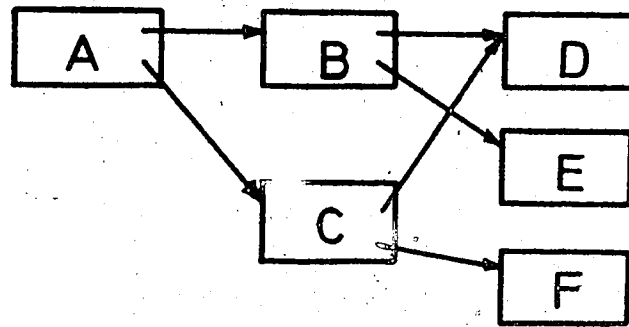
(a) Picture Description   (b) Structured Display File

Fig. 3.1. Structured Display File
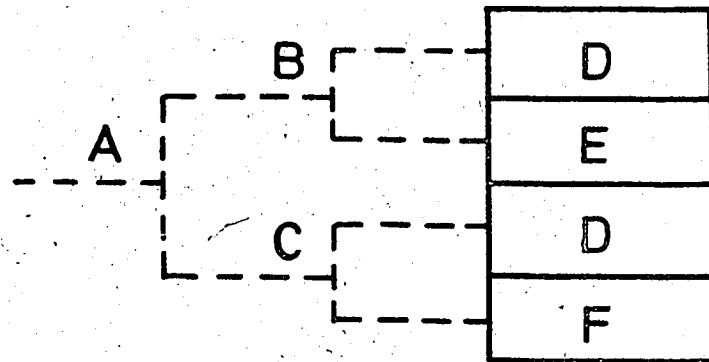
Fig. 3.2. Sequential Display File

A New Display File Construction Scheme

A scheme has been implemented here which ensures uniform construction of display files independent of the hardware available. A mixed display-file construction is used. The scheme basically uses a structured display-file construction scheme, for the reasons outlined above. Furthermore, the code that has to be generated in this case is small when a minor change is made to the display file, which is especially important for graphical systems with remote display terminals.

The scheme works as follows: when a picture is to be output, the DRAW routine determines whether the picture can be constructed as a purely structured display file. If this is possible, it will be so generated. Otherwise, if at any point in evaluating the picture tree, a transformation is required which is beyond the system's hardware capability, a sequential display file is generated for that subpicture.

To illustrate this point more clearly, consider the statement

A = B + C * ROTATE(ANGLE)

If the system has hardware rotation, the solution is straightforward. A structured display file as shown in Fig. 3.3 is generated:
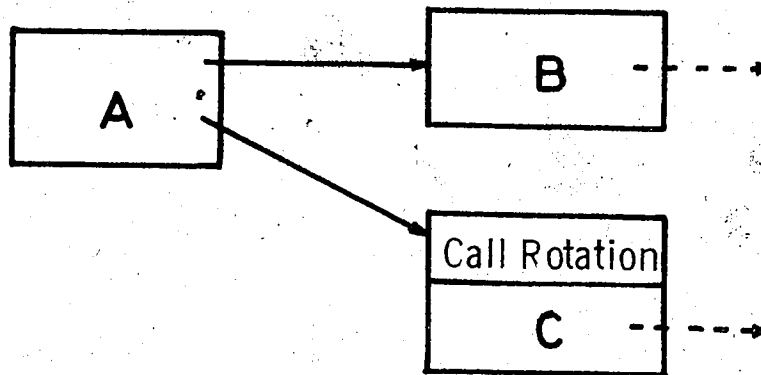
Fig. 3.3. Display File for System
with Rotation Hardware
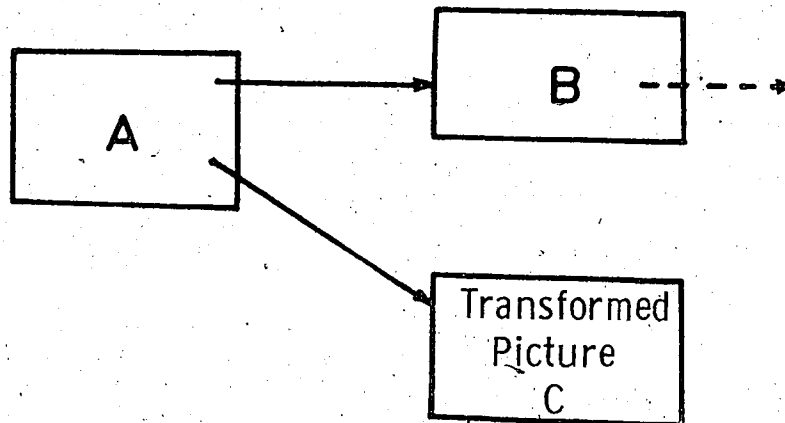
Fig. 3.4. Display File for System
without Rotation Hardware

On the other hand, if nc hardware rotation is available, the picture C, which may have subpictures and further subpictures, is transformed by software to give the rotation effect before the entire transformed sequential display file for C is generated, as shown in Fig. 3.4. The picture C is first converted from its tree-like description to a sequential description, in the form of a picture vector of the following format:

$$\begin{bmatrix} ATTLIST1 & X1 & Y1 \\ ATTLIST2 & X2 & Y2 \\ \cdots & \cdots & \cdots \\ TTLISTN & XN & YN \end{bmatrix}$$

where   X and Y   e the X and Y coordinates, and

ATTLIST is the attribute list for each element of the picture vector.

The picture, now in the form of a vector, can be easily transformed by software. Notice that the transformed picture has lost all its structure. The entire transformed picture C is considered to be a single component that has no further subpictures.

One slight drawback of this approach is that it can become quite clumsy if too many such transformations are required. Whenever such a tranformation is executed, all the subpictures have to be converted to give the picture

vector. To a certain extent, the efficiency of the system is dependent on how many such transformations are required. Thus, a system with more hardware transformation facilities is bound to be, as expected, more efficient.

Nevertheless, this method of display file construction ensures that the display file is constructed in an efficient manner. It takes full advantage of both types of display file construction. It is general and hardware independent. Any change in hardware will not affect the philosophy of the design. Depending on the requirement, purely structured or sequential construction is also applicable.

In order that the user can describe his pictures in real-world coordinates (or page coordinates), the user is allowed to specify the range of page coordinates as attribute parameters of the DRAW statement so that the entire page can be displayed on the screen. It is the responsibility of the DRAW routine to convert the page coordinates to the screen coordinates[30] in the display-file code-generation process. Since the conversion is not carried out until the actual display file is generated, the DRAW routine remains largely display-hardware independent; only the display-file code-generation routine has to be modified for different displays.

In contrast to the conventional use of address stacks to keep track of the branch of the picture tree that is being

displayed at any instant, the identifications(ID) of the picture components are stacked. This provides a more efficient means of identifying pictures picked by the light pen on the screen. The stack at the time of the interrupt will reveal exactly which picture component is picked, and provides a simple method of identifying different copies of the same picture displayed on the screen.

Thus for the picture A defined as

A = B + B

the code that is generated will be executed in the sequence shown in Fig. 3.5. More details of how the stack is treated are discussed in the next section.

```
STARTDIS        SET ID=1 AND PUT ON STACK

                CALL DISPLAY PROCEDURE A

                POP UP ID STACK

                GO TO STARTDIS

A               SET ID=1 AND PUT ON STACK

                CALL DISPLAY PROCEDURE B

                POP UP ID STACK

                SET ID=2 AND PUT ON STACK

                CALL DISPLAY PROCEDURE B

                POP UP ID STACK

                RETURN

B               . . . . . .
```

FIG. 3.5.   CODE EXECUTION SEQUENCE FOR THE PICTURE A=B+B

3.2.3. Decoding Input from Display Terminal

In an interactive graphics system, it is necessary to handle input and interrupts from the display terminal. These can be 'light pen interrupts, or input from the keyboard, function keys, joysticks, or even picture input via light pen, drawing tablets, etc. A facility 'is incorporated in the language for handling such interrupts. The general. format of the DECODE statement is: DECODE(<intno>,<dev>,<inform>,<pictname>,<x>,<y>,<tf>) where

<intno> specifies the interrupt number that is being decoded,

<dev> indicates the device from which the interrupt is expected. It is specified in the form of a character string denoting the device, e.g. LPEN, KEYBRD, etc.,

<inform> is a location in which device-dependent data may be stored,

<pictname> is a picture name which will be assigned to the picture that is being input,

<x> and <y> are variables in which the X and Y coordinates where the interrupt occurred are to be stored, and

<tf> is a true/false variable which is set to indicate whether the interrupt indeed comes from the device specified in <dev>.

With the exception of light pen hits on pictures
displayed on the screen, all inputs from the display
terminal can be considered as picture input, whether it be a
picture input via drawing tablets, text typed in via
keyboard, or sampled values from analog devices, etc. A
picture is thus defined, in a way analogous to a picture
definition, and the data structure is correspondingly
updated. The picture specified as <pictname> is the one
that is updated. Other immediately useful information is
communicated to the host language via the variables
<inform>, <x> and <y>. It should be noted that all
information is provided only if <tf> has been set to true,
that is, when the interrupt is as expected.

For light pen hits on pictures displayed on the screen, a
rather different decoding process is required. In fact, the
<pictname> parameter is replaced by a <id vector> parameter.
This is a vector of N+1 elements, where N is the depth of
the picture branch that is identified. Since the value of N
depends on the picture component that is being picked, the
<id vector> is of variable length. N is stored as the first
element of this vector and the remaining N elements contain
the actual identification.

Suppose that the identification vector is as shown in
Fig. 3.6. It implies that the picture that is picked by the
light pen is M levels deep. At the first level, it is the

a-th component, at the second, the b-th component, and so
on. Structurally, this is as shown in Fig. 3.7. Notice
that this identification vector uniquely identifies every
single branch of the picture tree. By proper investigation
of the identification vector at the correct level, it can be
determined whether it is indeed the branch of interest. It
is not always necessary to check all M levels: so long as a
satisfactory distinction can be made, it is not necessary to
interrogate further down.

For a more concrete example, consider the picture tree
shown in Fig. 3.8. If the first "B" is picked by the light
pen, the vector will be (2,1,1) since it is 2 levels deep.
If the second "B" is picked, then the vector is (2,1,3),
while if the second "I"(from left) is picked, then the
vector is (4,1,2,2,2). Since there is a stack which saves
these identifications when the picture is displayed, the
identification vector is easily obtained by converting the
stack to the proper format.

The DECODE statement thus serves the purpose of
identifying light pen hits and other interrupts from the
display terminal. It also serves to provide some
communication between the host and the graphics language.
More examples of this facility are given in Section 5.4.

M + 1 locations
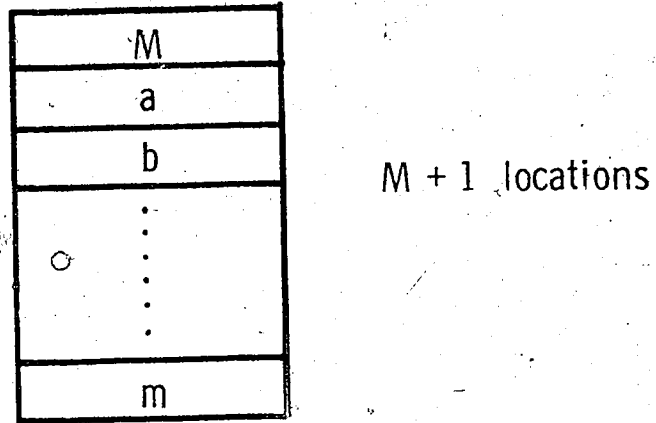
Fig. 3.6. Identification Vector
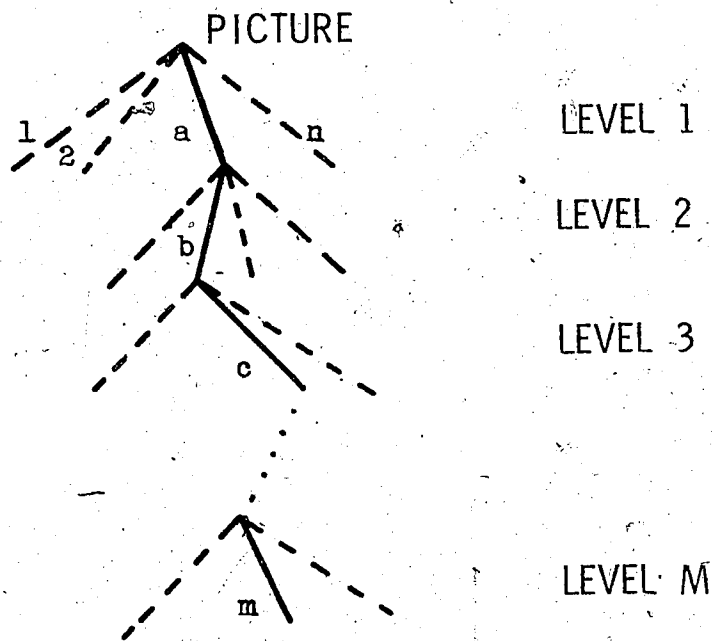
Fig. 3.7. The Picture Tree

$$A = B + C + B + D$$
$$C = E + F + G$$
$$F = I + I + I$$
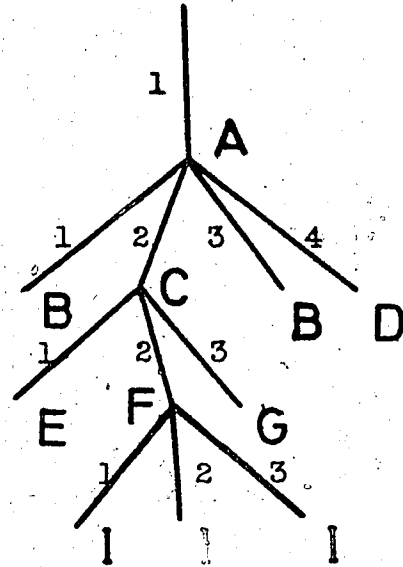
(a) Picture Description

(b) Picture Tree
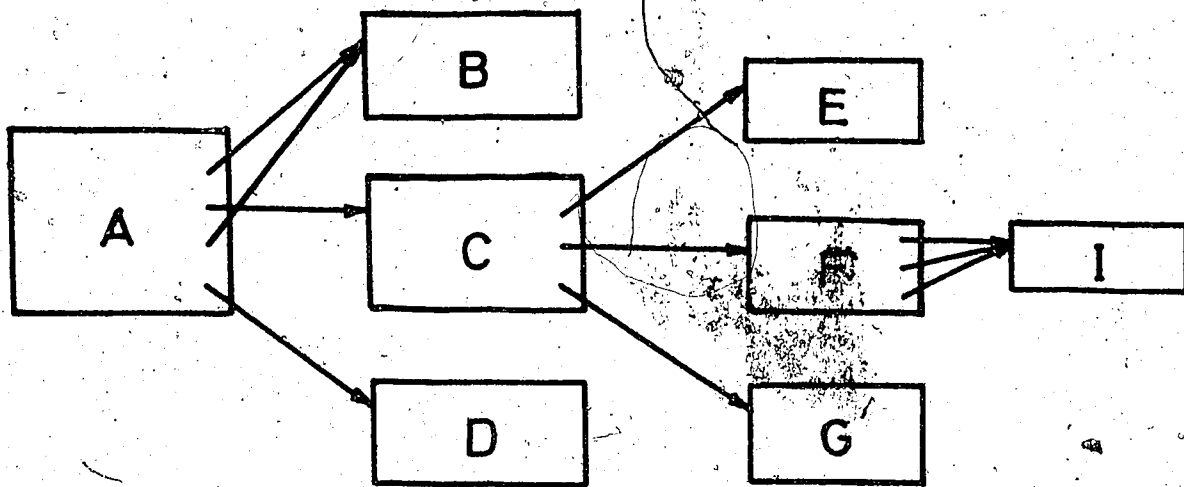
(c) Internal Representation (Schematic)

Fig. 3.8. An Example

3.2.4. Picture Information Retrieval

A facility for retrieving information about the pictures defined is often useful. Positional information and other attribute values are required when one wants to investigate such problems as connectivity of pictures. This compensates for the fact that only the '+' operator is available for grouping components into higher-level pictures. The RETRIEVE statement is designed to provide such a facility. The general format of the statement is:

RETRIEVE(<pict var>,<informa>, <var>)

where

<pict var> is the picture about which information is required;

<informa> is a character string which specifies the type of information that is required, e.g. NUMELEM, X, Y, etc., and

<var> is the variable for storing the returned information. In general it is a vector of variable size.

For example, the statement

RETRIEVE(PICT, NUMELEM, LOC)

checks the number of components that the picture PICT has, and the value is returned in location LOC. More examples can be found in Section 5.5.

This facility thus serves as a means of communication between the host language and the picture data structure. It also saves the user the trouble of maintaining picture information in the host language.

3.2.5. Function Declaration

Although a number of transformation functions are provided in the system (see Section 5.6), it is desirable that user-defined functions can be incorporated as well. A function declaration facility is therefore provided in the language. It should be noted that this statement has undergone drastic changes since the language was proposed. Rather than using it as a macro definition facility in the graphics language itself as it was first intended[31], it is now used for declaring transformation functions that the user has written in the high-level language instead. The rationale for the change is that a macro facility in the present graphics language alone is rather limited, since it does not have any arithmetic capability. Besides, it is easier to provide transformation functions in the high-level language.

The general format of the statement is

FUNCTION <fct name>(<para>)

where

<fct name> is the name of the user-defined function, and

<para> is a list of formal parameters.

The function declaration should be made before it is used in the program. The function must be incorporated by the user in the form of a subroutine, and has to be loaded at run time. As explained before, a picture vector is generated when a transformation is required. Since the picture vector is merely an N-dimensional vector, proper manipulation of this vector will give the transformation required.

For example, if FORTRAN is the host language, and the picture vector is a N x 3 vector where the X and Y coordinates occupy the second and third columns of the vector respectively, a function SCALE for the scaling of the picture can be written as shown below:

```
      SUBROUTINE SCALE(PV,N,SX,SY)
COMMENT PV REPRESENTS THE PICTURE VECTOR,
C       PV(1,I) IS THE ATTRIBUTE, PV(2,I) AND PV(3,I)
C       ARE THE X AND Y COORDINATES
C       N IS THE DIMENSION OF THE VECTOR,
C       SX IS THE X SCALE, AND
C       SY IS THE Y SCALE.
      INTEGER PV(3,N),N,SX,SY
      DO 100 I=1,N
      PV(2,I) = PV(2,I) * SX
      PV(3,I) = PV(3,I) * SY
```

```
100  CC   NUE

     RETURN

.    END
```

Notice that the first two parameters of user-defined functions are always reserved for the picture vector and its dimension, and need not be included in the parameter list of the FUNCTION declaration.

The function declaration facility should be valuable for those users who require unusual transformations of their pictures. The uniformity of the picture vector allows simple incorporation of the functions.

3.3. Discussion

A new graphics language has been defined in this chapter, and a formal description of its syntax is presented in Appendix I. It provides a graphics facility which is to be used with other high-level languages. The language is simple but sufficient for many interesting graphics applications. The picture description scheme is elegant and powerful. Good facilities for interactive graphics are also provided.

Chapter IV

A Universal Graphics Facility

4.1. The Notion of a Universal Graphics Facility

The development of computer graphics in the last decade has not been as fruitful as predicted[1]; not only is the hardware expensive but also the software is inadequate. Fortunately, the cost of graphics hardware has been coming down for the last five years[22], and a 10:1 cost reduction has been projected for the next five to ten years[13]. Graphics software, on the other hand, is intrinsically complex and requires a lot of programming effort[32]. A major obstacle is that graphics software is usually available in a restricted environment. Very often, the graphics facilities are accessible only via a particular host language, which is highly undesirable. Computer users are generally reluctant to use a new facility if they have to change to a new programming environment. Besides, the language in which the graphics facilities are available may not be suitable for handling their problems.

In most graphical applications, graphics operations are only one component in the total solution of a problem[32]. Invariably, capabilities other than picture description and generation are required. However, the areas of graphical applications are so extensive that the capabilities required

cannot be determined easily. Ideally, one should be able to use graphics in whatever language that is most convenient to express the problem as a whole. This is probably the language that the programmer uses most commonly. For example, a data-structure language is most suitable for applications in computer-aided design, whereas an algebraic language is most suitable for applications where complex data tranformations are required.

The above considerations brought about the concept of a Universal Graphics Facility, that is, a graphics facility which can be accessed from a number of high-level languages. Fig. 4.1 depicts an overall view of the concept. The advantages of this approach include:

1. Only a single graphics package has to be built for use among different high-level languages.

2. The capabilities of different high-level languages allow wide areas of applications in graphics.

3. Users can use the graphical facilities in the language of their own choice.

High-Level Languages   Universal Graphics Facility

FORTRAN

ALGOL

Graphics Software

Graphical I/O devices

COBOL

Host / Graphics Interface
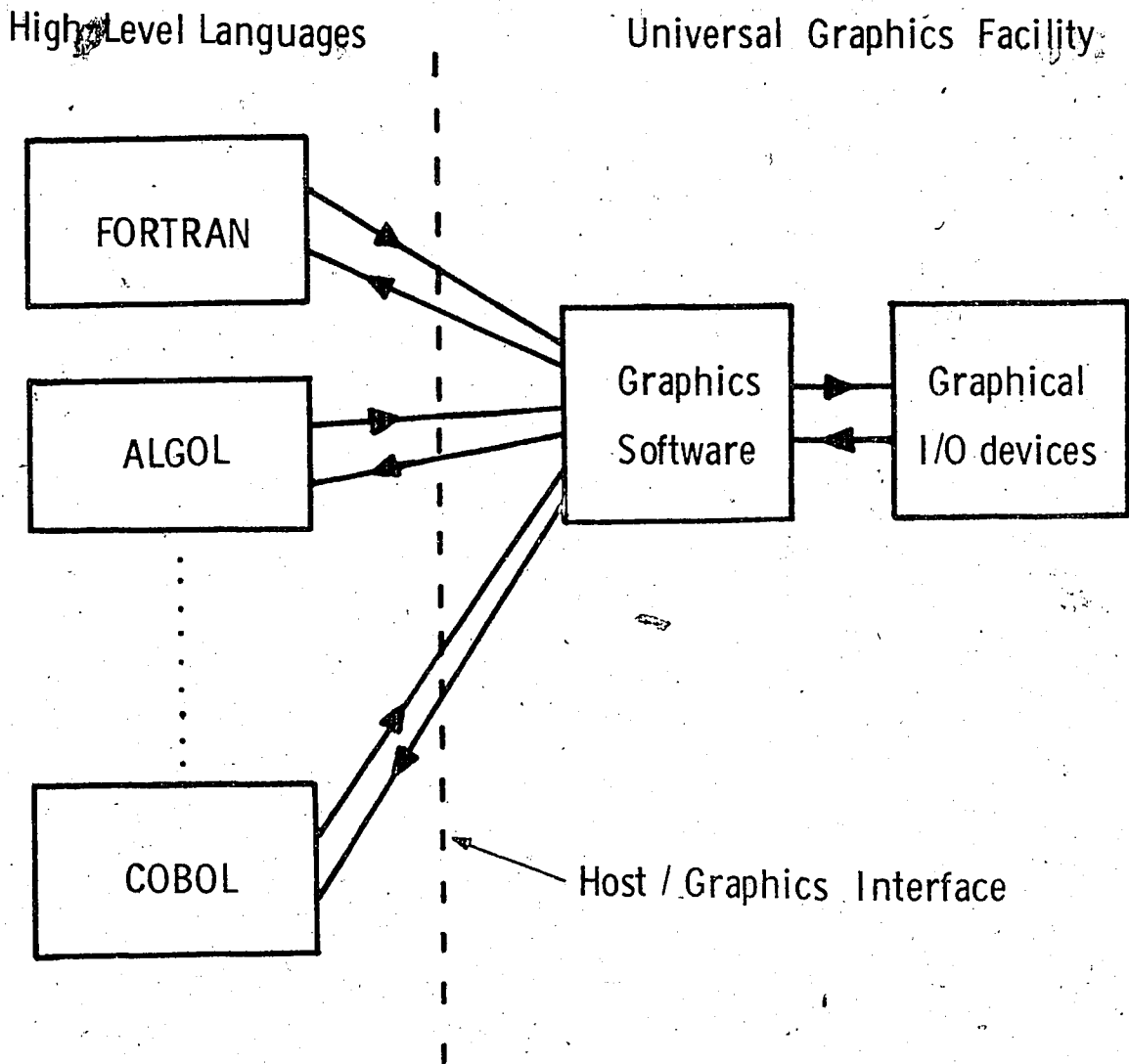
Fig. 4.1. Notion of a Universal Graphics Facility

It should be noted that the Universal Graphics Facility
bears close resemblance to software packages for handling
ordinary computer input/output using the READ and WRITE
operations. However, the objective here is to implement the
graphics software in such a way that it can be used in
"association" with a number of high-level languages. The
graphics software is neither provided as a package of
subroutines that can be called directly from the host
language, nor is the host modified so that graphics
operations are included. Rather, a technique is developed
which allows facilities of both languages to be used within
a single program, with each language retaining its
integrity. The graphics software is so implemented that it
does not depend on any particular high-level language, and a
means of communication between the two languages is
available to provide meaningful "symbiosis" of the host and
the graphics languages.

4.2. <u>Symbiosis</u> <u>of</u> <u>Host</u> <u>and</u> <u>Graphics</u> <u>Languages</u>

To establish a more "habitable environment" for the graphics users, it is important that the graphics operations be easily accessible so that they appear to be essentially extended operations of the host language. The philosophy developed here is to allow free mixing of host and graphics statements. In this way, users can program in both languages at the same time. The fact that one can program in both languages in a continuous stream gives the impression that the graphics facilities are part of the host language itself. By suitably mixing statements of the two languages, facilities of both languages can be used to their best advantage. This approach differs from the usual embedding methods in that the host language does not have to be modified in any way; it is a simple way of providing graphics operations to the high-level languages.

Before such mixed statements can be executed, they must be compiled. A preprocessor is used to separate the two types of statements and establish proper linkages between the two languages at execution time. However, ambiguities between the syntaxes of the two languages may exist. In order to facilitate the recognition process, the graphics statements are flagged so that they can be easily distinguished from the host statements.

The preprocessor scans the mixed statements and detects blocks of consecutive graphics statements. Each graphics statement is checked to ensure that it is syntactically correct. Each block of graphics statements is then replaced by an appropriate call to the host/graphics interface. This interface is also generated by the preprocessor and is instrumental in providing the proper linkages between the host language and the graphics software during execution. The graphics statements are converted to calls to the graphics software within the host/graphics interface. A schematic description of the preprocessor is shown in Fig. 4.2. To ensure that the interface can cope with different linkage conventions, the interface should be at machine or assembly language level.

Modified Program

in Host Language

Input of Mixed

Host and Graphics

Statements

Host Statements
Flagged Graphics Statements
Host Statements
Flagged Graphics Statements
Host Statements

Host Statements
Call to Routine GR0001
Host Statements
Call to Routine GR0002
Host Statements

Host / Graphics Interface

Interface Routine GR0001
Interface Routine GR0002

Before

Preprocessing

After

Preprocessing

Fig. 4 . The Preprocessor

The mod   d host program, on the other hand, now only consists of statements in the host language and can be translated into object code by the host compiler. Without loss of generality, the linkage from the host language to the interface can be established by introducing the variables as parar   of the modified calls. This scheme is quite general and can be used for most high-level languages. However, simpler means of communication is often possible and can be used as well. For example, in FORTRAN, by restricting the variables to be declared in COMMON blocks, the address of the variables can be established without relying on the parameter list. The generality of the preprocessor is not affected since the linkage convention and structure of different host languages are different, and the scanning and interface sections of the preprocessor differ in any case.

No matter how the linkage from the host is achieved, the host/graphics interface must accomplish the following tasks before it can be used as the linkage between the host and the graphics software:

1. On entry to the interface, the operating environment of the host must be saved.

2. The addresses of the variables must be properly established and saved in a variable table used by the graphics software.

3. Calls to the graphics software must be generated in

accordance with the graphics statements used. References to variables should now be reduced to references to the variable table.

4. Before returning, appropriate results must be passed back to the host.

5. The previous operating environment must be restored.

The variable table is used so that addresses of variables have to be evaluated only on entry to the interface. Each entry in the variable table contains the name of the variable, its data type and its address (or means of calculating the address). Reference to the variables after entering the interface are made only via the variable table. A uniform way of linking to the graphics software independent of the host used is thus provided.

Case studies of how the graphics software can be used with different high-level languages are presented below. The hosts used are FORTRAN, PL/I and ALGOL W, as implemented on the IBM 360 machines. The technique used can be similarly applied to other hosts as well. In all cases, the preprocessor remains the same except for the scanning and the interface sections.

Case Study 1.   FORTRAN as host

The use of '#' in column 1 serves as a simple identification of the graphics statements. By simply inspecting the first column of each card, the graphics statements are distinguished. The interface can be generated easily since IBM/360 FORTRAN uses a standard IBM linkage convention[15]. In fact, if the variables used in the graphics statements are restricted to COMMON blocks, as in the current implementation, the address of a variable can be determined without introducing it as a parameter. Details of implementation are described in Chapter 5.

Case study 2.   PL/I as host

In PL/I, the identification of graphics statements is best introduced in the form of special character(s) that cannot be legally used as the starting character of any PL/I statement. The use of '#' is one possible identification. It would also be helpful to terminate the graphics statements with semicolons so that other PL/I statements can follow in the same line. To scan for the graphics statements, it is necessary to look for the semicolons and the keywords 'THEN' and 'ELSE', since any meaningful graphics statement can only be preceded by the semicolon or one of those keywords. However, the scanning process can be much simplified if the restrictions are imposed that '#' must be specified in column 1 and that each graphics

statement begins on a new line. This does not impose much hardship on the PL/I user, and requires much less work from the preprocessor.

As far as the interface is concerned, variables can be introduced as parameters of the modified calls. However, in spite of the fact that PL/I, as implemented in the IBM SYSTEM/360, also uses register 1 to point to the parameter list, the linkage convention of PL/I is more complicated than FORTRAN. Dope vectors are used for strings and arrays and any reference to these data types must be appropriately taken care of within the interface. Details of the format of the dope vectors can be found in the PL/I manual[16], and are not described here. To illustrate how the linkages are established, consider the mixed PL/I and graphics program shown in Fig. 4.3. The corresponding modified program after preprocessing is shown in Fig. 4.4. The procedure GR0001 is the interface that is generated by the preprocessor to provide appropriate linkages to the graphics software. Fig. 4.5 shows the sequence of actions within the procedure GR0001.

```
PL1:   PROCEDURE OPTIONS(MAIN);

       DCL(A,B,C,D) FIXED;

       A = 0 ;

       B = 10 ;

       C = 20 ;

       D = 30 ;

#      PICT = V(A,B) + V(C,D) ;

       . . . . . . . .

       END;
```

FIG. 4.3. MIXED PL/I AND GRAPHICS PROGRAM

```
PL1:   PROCEDURE OPTIONS(MAIN);

       DCL GR0001 ENTRY(FIXED, FIXED, FIXED, FIXED);

       DCL(A,B,C,D) FIXED;

       A = 0 ;

       B = 10 ;

       C = 20 ;

       D = 30 ;

       CALL GR0001(A,B,C,D) ;

       . . . . . . . .

       END;
```

FIG. 4.4. MODIFIED PROGRAM IN PL/I

```
GR0001    CSECT

          SAVE REGISTERS

          SET UP ADDRESSES OF VARIABLES A,B,C,D IN THE

               VARIABLE TABLE USING PL/I CONVENTIONS

          CALL PICTSET,(PICT)

          CALL ADDVEC,(A,B)

          CALL ADDVEC,(C,D)

          CALL ENDPICT

          RESTORE REGISTERS

          RETURN

          END
```

FIG. 4.5.   SEQUENCE OF ACTIONS WITHIN THE INTERFACE
            ROUTINE GR0001.

Case Study 3. <u>ALGOL</u> <u>W</u> <u>as</u> <u>host</u>

The same flagging and scanning procedures as for PL/I can
be used for ALGOL W since both languages have similar
statement formats. However, some differences do exist. For
example, the keywords 'BEGIN' and 'DO' can also precede
graphics statements and must be taken care of by the scanner
as well. Another point is that if the graphics statement is
preceded by the keyword 'THEN' and followed by 'ELSE', the
semicolon used to delimit the graphics statement must be
eliminated. Again, if the same restrictions as for PL/I
host are applied here, the scanning process can be easily
achieved.

As far as the interfacing section is concerned, the
linkage convention of ALGOL W is substantially different
from PL/I. Besides, ALGOL W can only access externally
compiled routines which have been declared as 'FORTRAN' or
'ALGOL' type[37]. Nevertheless, the interface can be
similarly constructed. If the interface routine is declared
as 'ALGOL', ALGOL W linkage conventions have to be used
within the interface. Similarly, if the interface routine
is declared to be 'FORTRAN', then FORTRAN linkage
conventions have to be followed.

For a similar example as in Case Study 2, the
corresponding programs before and after preprocessing are
shown in Fig. 4.6 and Fig. 4.7 respectively. The interface

remains practically the same as in Fig. 4.5, except that the

procedure for setting up the variables now follows FORTRAN

linkage conventions.

From the above case studies, it can be seen that if the

linkage convention of the host is known, the provision of

the graphics facility to the host language poses no serious

problem. By suitably flagging the graphics statements, the

mixed statements can be readily preprocessed to provide the

interface.

```
BEGIN INTEGER A,B,C,D;

        A := 0 ;

        B := 10 ;

        C := 20 ;

        D := 30 ;

#       TRIAN = V(20,0) + V(10,20) + V(0,0) ;

#       PICT = TRIAN(A,B) + TRIAN(C,D) ;

        . . . . . . .

END.
```

FIG. 4.6.   MIXED ALGOL W AND GRAPHICS PROGRAM

```
BEGIN   PROCEDURE GR0001(INTEGER VALUE DA,DB,DC,DD);

        FORTRAN "GR0001";

        INTEGER A,B,C,D;

        A := 0 ;

        B := 10 ;

        C := 20 ;

        D := 30 ;

        GR0001(A,B,C,D);

        . . . . . . .

END.
```

FIG. 4.7.   MODIFIED PROGRAM IN ALOGL W

4.3. Examples of Use

One advantage of using graphics statements in conjunction with high-level language statements is that the power of the picture description scheme is greatly enhanced. The description also becomes more user-oriented. This is best demonstrated by examples.

Example 1. Plotting of Bar Graphs using FORTRAN

Plotting of bar graphs can be achieved in the graphics language alone if the explicit coordinates are known. However, the description becomes very lengthy if many points have to be plotted. Fig. 4.8 shows how it can be easily achieved using mixed FORTRAN and graphics statements. The program reads in the Y ordinates and the bar graph is plotted as shown in Fig. 4.9. Notice that the way the bar graph is defined allows any portion of the graph to be modified using suitable graphics statements.

```
        IMPLICIT INTEGER (A-Z)
  #      BGRAPH = XAXIS + YAXIS + GRAPH
  #      GRAPH = NULL
  #      XAXIS = V(U,0)
  #      YAXIS = V(0,V)
         READ (5,1) A, M
         OLDY = 0
         DX = A
         DO 10 I=1,M
         READ (5,1) Y
  1      FORMAT (1X,I10)
         DY = Y - OLDY
  #      GRAPH = GRAPH + V(0,DY,REL) + V(DX,0,REL)
         OLDY = Y
 10      CONTINUE
  #      DRAW (1,BGRAPH)
         STOP
         END
```

FIG. 4.8. PLOTTING OF BAR GRAPH IN FORTRAN
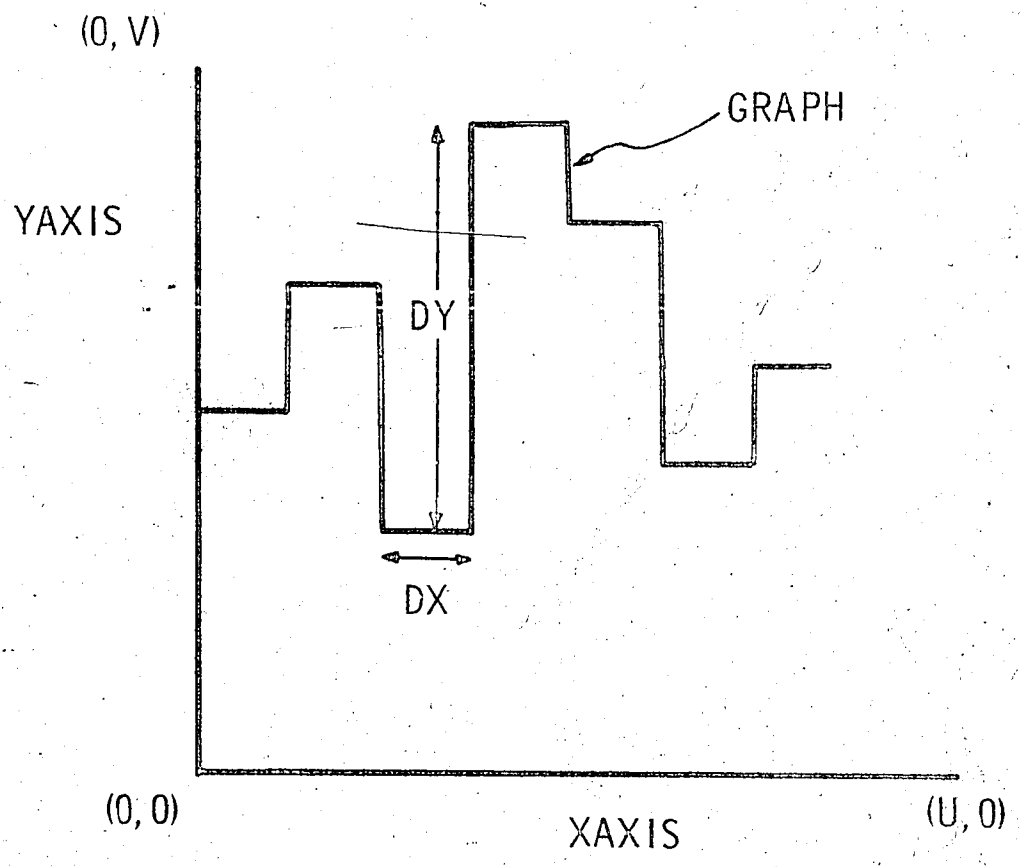
Fig. 4.9. Plotting of Bar Graph

Example 2. <u>Plotting</u> <u>of</u> <u>a</u> <u>Page</u> <u>of</u> <u>Text</u>

A program written in ALGOL W as shown in Fig. 4.10 can be used to plot a page of text in the format as shown in Fig. 4.11. Notice that a line of text is used as the primitive here. There are M lines in a page and 60 characters in a line.

```
     BEGIN INTEGER X,Y,M,D,A,B; STRING(60) TEXT;
          READ(M,D,A,B) ;
           X := A ;
           Y := B ;
#          PAGE = NULL ;
           FOR I:=1 UNTIL M DO
           BEGIN
             READ(TEXT) ;
#            PAGE = PAGE + T(X,Y,60,TEXT) ;
             Y := Y - D
           END;
#          DRAW(1,PAGE) ;
     END.
```

FIG. 4.10. PLOTTING A PAGE OF TEXT IN ALGOL W
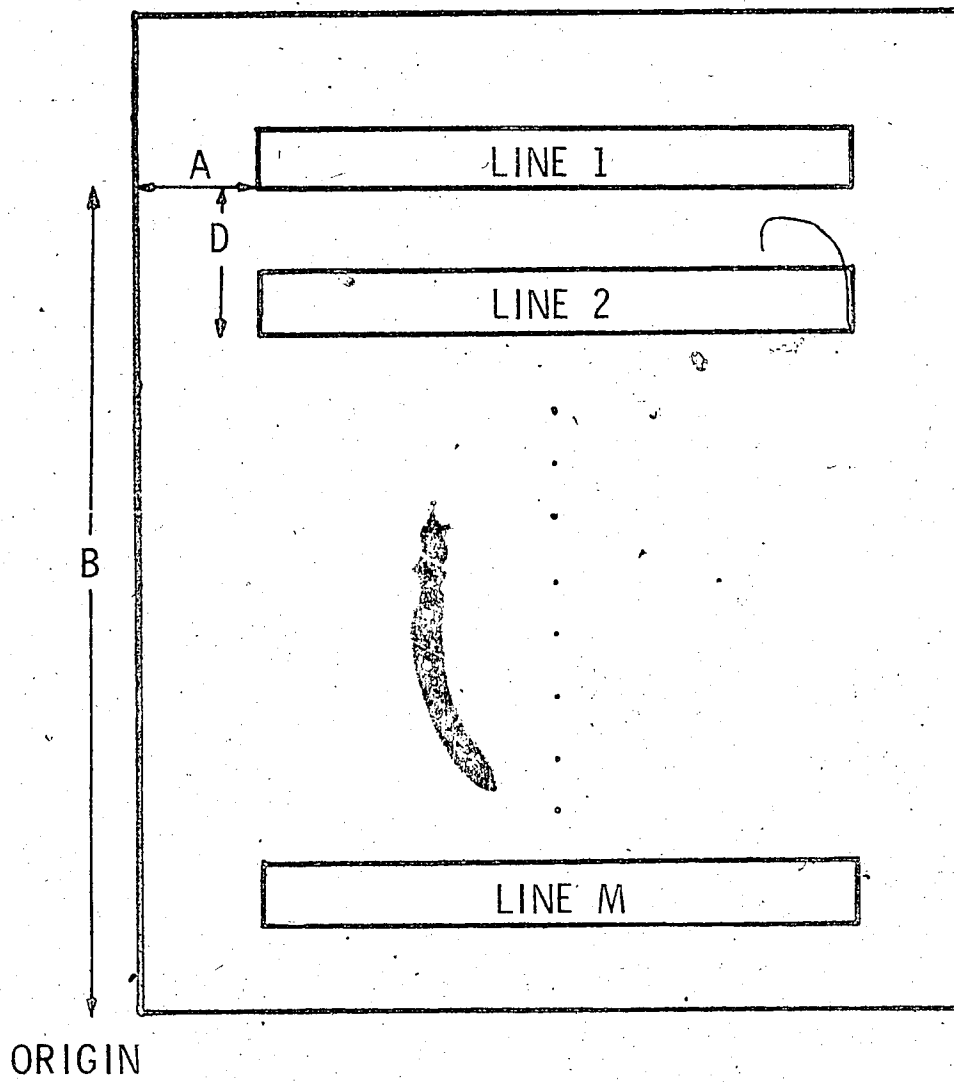
Fig. 4.11. Plotting of a Page of Text

Example 3. Generation of Cobweb in PL/I

A cobweb can be generated by the mixed program of PL/I
and graphics statements shown in Fig. 4.12. The resulting
picture is shown in Fig. 4.13.   Notice that conditional
execution of graphics statement is used here.

```
PROCEDURE OPTIONS(MAIN);
        DCL  (X,Y,DX,A,B)  FIXED;
        GET(A,B) ;
        X = A ;
        DX = B ;
        DO I=1 TO 1000 ;
          IF MOD(I,2) =0 THEN
            PICT = PICT + V(Y,0,REL) + V(0,Y,REL) ;
          ELSE
            PICT = PICT + V(X,0,REL) + V(0,X,REL) ;
          X = X + DX ;
          Y = -X ;
        END;
        DRAW(1,PICT) ;
    END ;
```

FIG. 4.12. PLOTTING OF COBWEB IN PL/I

ORIGIN
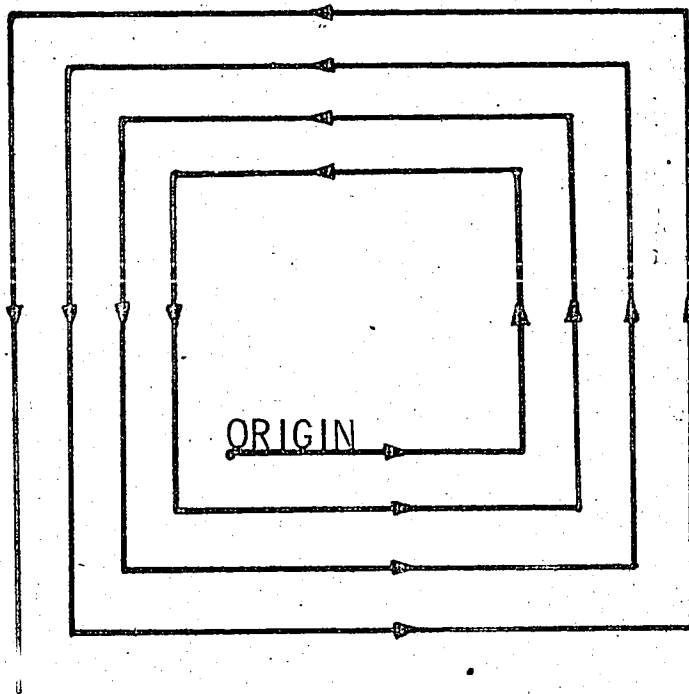
Fig. 4.13. Plotting of Cobweb

4.4. Discussion

The concept of a Universal Graphics Facility has been introduced in this chapter. A technique has also been proposed which can be used to provide such a facility. The examples given demonstrate that symbiosis between the host and the graphics language provides an easy means of describing pictures.

The significance of the approach lies in the generality of the graphics facility. Only a single graphics package has to be built for use among different host languages. It is not necessary to modify the host languages at all. It results in considerable savings in effort in the of the graphics facility.

It should be noted that this approach can be generalized to other problem-oriented languages (of which a graphics language is an example). It is an effective way to achieve software compatibility[10]. With this provision, facilities of one language can be used in another.

Chapter V

Implementation Details

5.1. The Environment

The GRAphics Facility (GRAF) is implemented on a graphical sub-system connected to an IBM 360/67 operating under Michigan Terminal System (MTS) at the University of Alberta. The graphics sub-system consists of a CDC Graphical Remote Interactive Display (GRID)[3] which is interfaced to the IBM 360/67 via a 40.8 kilobits/sec half-duplex telecommunications line. The GRID is composed of a cathode ray tube display with 1024x1024 addressiable units over a screen size of 12x12 inches, limited arithmetic capability (e.g. there is no multiplication other than by 10, no division or indexing), 12K store of 12 bits words (in banks of 4K words each), 10 function keys, 4 status keys (allowing 16 status settings), light pen and keyboard. The display repertoire permits only the plotting of points, vectors and characters.

Because of limited processing capability, the GRID relies heavily on the 360 for data and structure processing. It is in essence used as a remote display terminal and is used only for refreshing the display, handling user interaction at the display terminal and communicating with the 360. The Multibank Display Supervisor[39], which was developed for

GRIDSUB -- a package of FORTRAN routines for interactive
graphical display programming[17], was slightly modified to
accommodate the present system. However, the behaviour of
the supervisor remains basically the same. It allows the
user to compose messages interactively using the light pen,
keyboard and function keys. Facilities for drawing vectors
and points on the screen using the light pen are also
available.

GRAF is currently implemented with FORTRAN as the host
language. Both the graphic software and the preprocessor
are programmed in 360 assembler language. At present,
variables are restricted to integers and integer arrays of
one dimension. However, the routines for handling the
variables are quite detached from the rest of the software
and can be extended to take care of other types of variables
if required. The first version of the language is now ready
for use.

5.2. The Picture Data structure

A picture name table, in conjunction with a linked list
structure, is used to model the pictures. The data
structure is updated as the picture definition statements
are executed. Fig. 5.1 shows an overall view of the
modelling scheme.

Picture
Name　　　Pointer

PICTNAME

Picture Name Table

Linked List Structure
representing the
picture components

NULL
Pointer

First
Field

Second
Field

Third
Field

LINK
pointer

Picture
Information

UP TO 7
Transformations

Function
entry point

List of
parameters
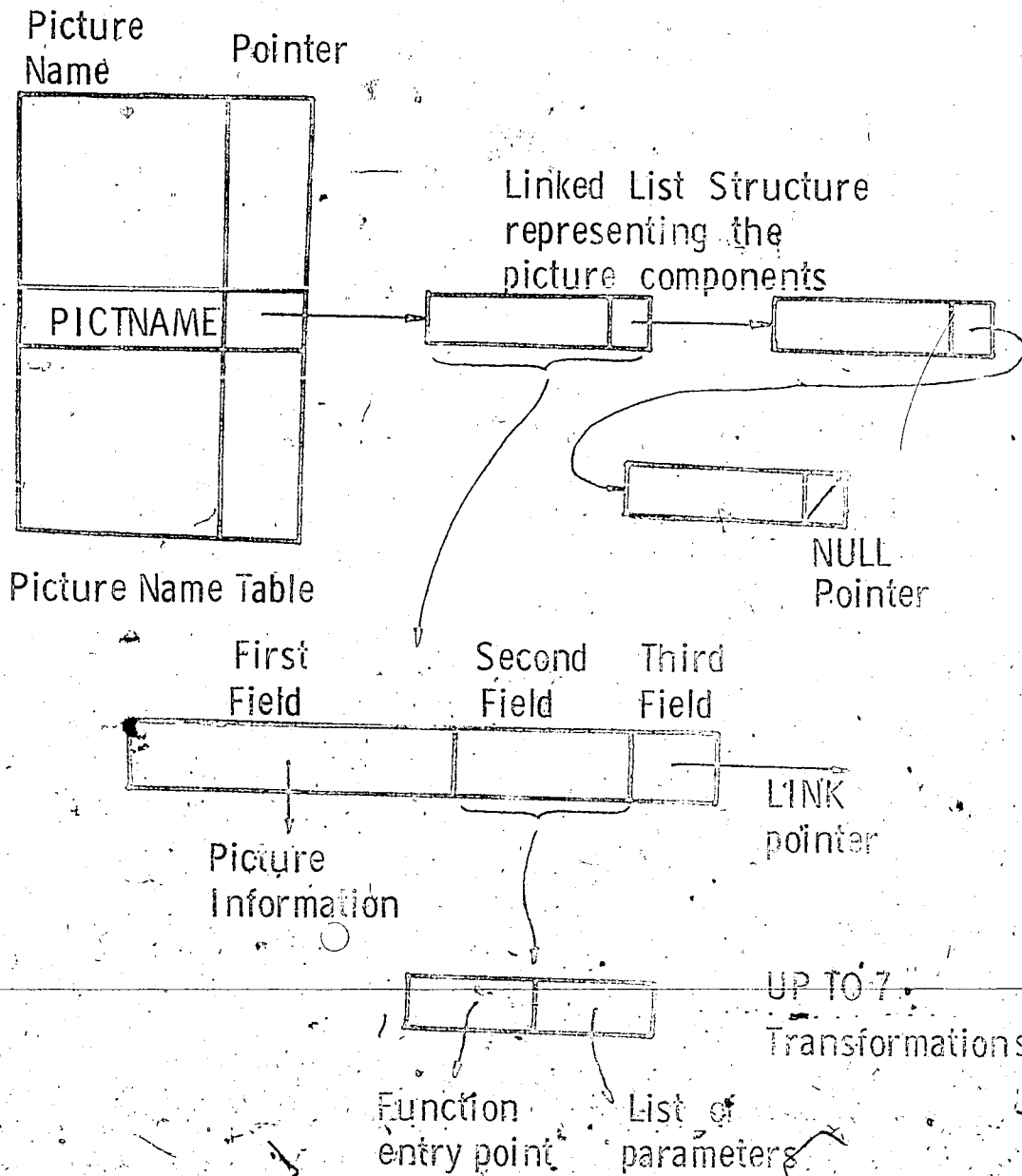
Fig. 5.1. The Picture Data Structure

Each entry in the picture name table occupies 3 words
(all references to "word" are to a 360 word of 32 bits
unless stated otherwise). The first two words are used for
the picture name; hence, a maximum of 8 characters is
allowed for each picture name. The third word is a pointer
to the first component of the picture (which can be NULL).
The picture names are hashed into the table during
preprocessing time. Obviously, a picture name is defined
only if it appears on the left-hand side of a picture
definition statement. Initially, all valid picture names
are assigned the value NULL.

The picture components are represented by the list
elements. Each list element describes one component and
consists of three fields. The first field contains the
following information: the attribute values, the coordinates
of the component, the size of the list element (i.e. the
number of words used in forming the element), the number of
transformations specified, and whether the component refers
to a subpicture or a primitive. The size and format of this
field vary for different primitives and a maximum of 5 words
is used. Fig. 5.2 shows the format of the first field for
the different primitives, and a detailed description of the
attribute word is given in Fig. 5.3.

1. PICTURE   Picture name table

Attributes		X	Y

2. POINT   not used   8 bits

Attributes		1	X	Y

3. VECTOR   not used

Attributes		2	X	Y

4. SYMBOL   not used

Attributes	S	4	X	Y
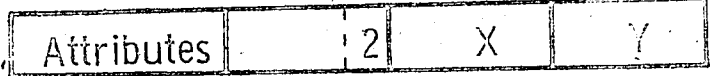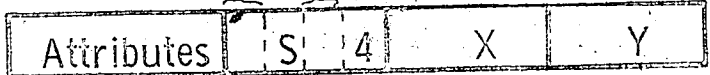
symbol

5. LINE   not used

Attributes	N	L		

no. of vectors   pointers to X and Y coordinates

X_1
Y_1
⋮
X_N
Y_N

6. TEXT   not used
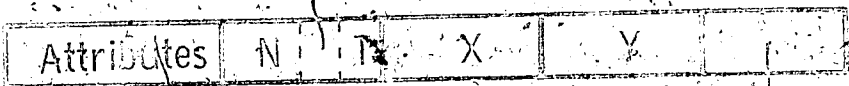
Attributes	N	T	X	Y	

no. of symbols   pointer to text

N.B. L = 10 or 11 for LINE type 1 or type 2
T = 12 or 13 for TEXT type 1 or type 2

Fig. 5.2. Format of the First Field for different Component Types

| 31 | 19   16 15 | 0 |
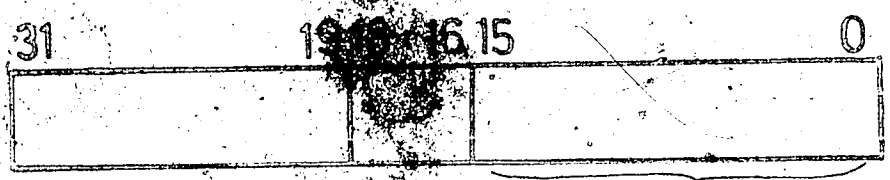
number of 360
words used for
this component

no. of
transformations

Attributes

The attributes only
apply if appropriate

BITS	MEANING
1 - 0	Primitive / Picture
3 - 2	Absolute / Relative Positioning
5 - 4	Light Pen / Not Light Pen Detectable
7 - 6	Blink / Non blink
9 - 8	Blanked / Non-blanked
11 - 10	Solid / Dashed Vector
13 - 12	Small / Large Symbol
15 - 14	Vertical / Horizontal Symbol (for L primitive, Absolute / Relative Positioning for <POS2>)

Fig. 5.3. The Attribute Word

The second field, whose length is variable, is used to
store the information pertaining to each of the
transformations specified. Currently, a maximum of 7
transformations is allowed, since three bits only are used
for the number of transformations in the first field (see
Fig. 5.3). If no transformation has been specified, then
this field is omitted. Two words are used for each of the
transformations. The first word is a pointer to the entry
point of the function that performs the transformation, and
the second word is a pointer to the list of parameters (see
Fig. 5.1). The transformations are entered in the order as
they were specified in the picture definition statement.

The third field is a pointer which provides the linkage
to the next picture component. If it is the last component
of a picture, then this field is assigned the value NULL.

It can be seen that the picture data structure closely
reflects that description in the picture definition
statement, and thus facilitates operations such as
modification of the pictures and retrieval of information
from the data structure.

A linked list is also used to keep track of free spaces
in the picture data structure area. Any unused space and
space that is no longer required by the data structure is
returned to the free-space list. Whenever space is required
for a new picture component, the free-space list is searched

for an element that is large enough to accommodate the picture component. The picture component is then moved in and the free-space list is updated. The problem of space fragmentation is not serious here since the variation in size of each element is relatively small; from a minimum of 5 words to a maximum of 19 words, and the elements are five words most of the time. The free-space size in the current implementation is 1024 words initially, and automatically extends up to ten times the initial size.

Another table is used to keep track of pictures that have been modified, but not yet updated in the display file. Each entry in the table is a pointer to the picture name table for the picture involved. This table is necessary since not all pictures that are defined are stored in the display file. More discussion on the display file will be made in the next section.

5.3. <u>Display File Management and Code Generation</u>

display file, which is a replica of the GRID core, is maintained in the 360. Whenever there is a change in the current displayed picture, the display file is updated and modifications are transmitted via the link to the GRID. A display file table is used to keep track of the current locations of picture routines in the display file.

The 3 banks of GRID core are used for different purposes.

Bank 0 is used for the display supervisor and handling of interrupts. With the exception of a small section (66 GRID words) in each bank which is used for the handling of interrupts, Bank 1 and Bank 2 are used as the display file area. The architecture of GRID is not suitable for structured display file construction since it does not allow simple transfer to the other banks while in display mode. Hence a straightforward calling sequence for picture subroutines (as demonstrated in Chapter 3) is not possible.

To overcome this problem, a set of directory routines is maintained in Bank 1. These directory routines serve the purpose of switching banks and branching to the correct entry points. Picture routines that need to call other picture routines are placed in Bank 1, while those picture routines that consist of display instructions only (e.g. pictures defined by primitives) can be placed in Bank 2. Fig. 5.4 shows the layout of the display files in Bank 1 and Bank 2. For each routine currently displayed (either in Bank 1 or Bank 2), there is a corresponding entry in the directory. A total of 100 entries are allowed for both banks. The directory routines are entered in the same sequence as they appear in the display file table.

BANK 1

BANK 2

0_8

77_8

2740_8

7777_8

Interrupt
Handling

Directory
Routines

Display

File

Area

0_8

77_8

7777_8

Interrupt
Handling

Display

File

Area
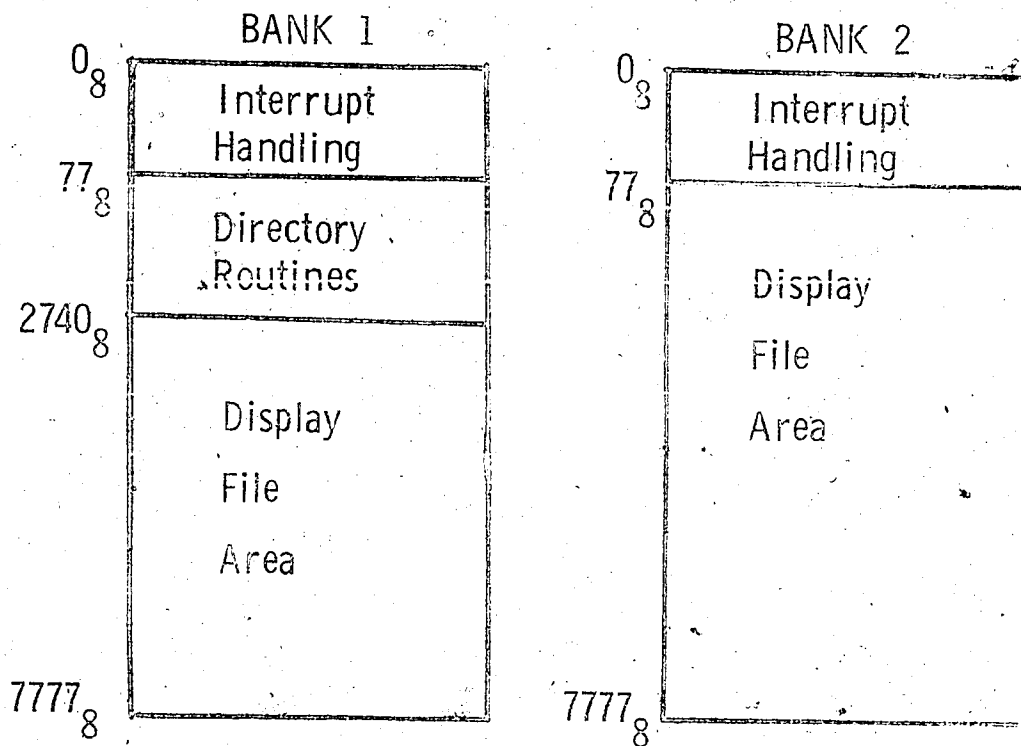
Fig. 5.4. Layout of Display File

Whenever a branch to a picture routine is to be made, a
jump to the directory routine is first performed. From
there, a branch to the entry point of the picture routine,
which may be in Bank 1 or Bank 2, is made. Although the
directory routines take up quite a bit of space (slightly
more than one-third of Bank 1), they allow both Bank 1 and
Bank 2 to be used effectively as the display file area.
Also, the code-generation process is much simplified since
the absolute address of where to branch to can now be
determined easily: in fact the display code for the picture
can be generated before its subpicture are generated and
inserted in the display file.

The Display File Table

The display file table consists of 100 entries, each of 4
words. The first word is a pointer to the picture name
table indicating the picture that is being displayed. The
second word is used to indicate whether the entry is
significant (i.e. if the copy of the picture is required for
the current display), the bank in which the routine is
located, and the starting and ending addresses (as
displacements from the top of the display file) of the
routine that is generated for the picture in the 360 display
file. The third word is reserved for the attributes of the
picture in order to distuishing copies of the same picture
that have different attributes. The fourth word is a

pointer which forms a linked list with other picture routines that are associated with this picture. The linkage is required for those picture components which do not have an explicit picture name, essentially the transformed pictures. An example is given in Fig. 5.5, which also shows the layout of the display file table. The linkage pointer is used mainly for garbage-collection purposes. When the picture is no longer required in the display file, all the entries in the linked list can also be deleted.

The present 100 entries for the pictures in the display file table should be sufficient for most applications, since only pictures currently displayed need be entered into the display file table.

Code Generation

The display file table together with the directory routines render code generation a relatively simple process. The table used to keep track of pictures that have been recently modified also helps to reduce picture regeneration time; only those pictures that are found in this table need be regenerated.

PICT = A + B * ROTATE(ANGLE)

Pointer to
Picture Name
Table

		PICT	A			

Attributes

Linkage
Pointer

100 entries

	not used		Starting Address	Ending Address
31 30	26 25 24 23		12 11	0
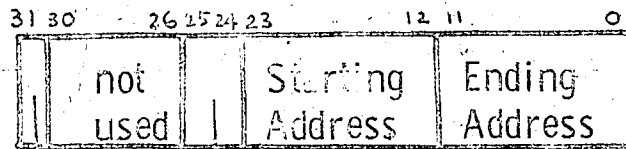
Bank Number

Significant
Bit

Fig. 5.5. Illustration of the Display File Table
and the Linkage Pointer

When a picture is to be output for display, its picture tree is evaluated to generate the display code. Since the entry points of picture routines are determined only by the location of their respective directory routine, the code for display can be generated starting from the top of the picture tree.

Appropriate code is generated for each of the components of the picture. If any component is a subpicture, the display file table is first checked to see if that subpicture exists in the display file. If not, the _ture is entered into the first available entry in the di__ ay file table. The "significant bit", which is used to indicate whether _____ __ required for the current display, is se_____ ____ture is also put into a display queue for la___

On the o____ _____ _____ subpicture exists in the display fil____ ____ _ble for modified pictures is checked to se_____ __ subpicture has been modified since last being displayed. If it is found in the table, then the subpicture has to be regenerated and is put in the display queue. Otherwise, the code for that subpicture is already present and does not have to be regenerated.

Since there is a direct correspondence between the display file table entry and the directory routine, the entry point of the directory routine can be calculated once

a picture is placed in the display file table. As a result, the absolute entry-point address can be determined even before the subpicture is generated.

After the code for a picture has been generated it is inserted into the display file and the display file table is updated. The process is repeated for all the pictures that have been placed in the display queue, thus ensures that all subpictures will be generated. Also, after a picture has been generated in the display file, it is deleted from the table for modified pictures.

It should be noted that all the display code generated is in incremental plotting mode. For pictures that require transformations, the picture vector is generated in the format as described in Chapter 3. Three words are used for each element: the first word for its primitive type and its attributes, the second and third words for the X and Y coordinates respectively. Again, incremental plotting mode is used for all the coordinate values.

Display File Management and Garbage Collection

To insert the generated codes into the display file, it is first decided whether it should be inserted into Bank 1 or Bank 2. Then it is checked whether the bank has sufficient space left. If it does, the code is inserted into the first available space. Thus a contiguous chunk of

code is generated in each bank, which will then be transmitted to the GRID after all the pictures in the display queue have been processed. This is more efficient than transmitting one block of code at a time.

A list of pointers (DSNUM) is used to keep track of the sequence of useful code in the display file, to facilitate garbage collection. Code generated is not purged from the display file until space has been used up and more space is required. Compaction of core is then made. Two compaction schemes are used. The first one is called minor compaction. The DSNUM list is checked to see where compaction is possible. All the code that is no longer required is purged, and the remaining blocks of code, independent of whether they are actually used in the current display or not, will be compacted. More free space is thus left at the bottom of the display file and can be used for more new blocks of code.

The second scheme is the major compaction scheme. It is applied when minor compaction does not leave the required amount of space. In such a case, the display-file table is checked for blocks of code residing in the display file but not required for the current display, which is accomplished by checking the DSNUM list and the corresponding significant the display file table. If there are such blocks, entries in the display file table and the DSNUM list

are deleted. After all the entries in the display file table have been checked, a minor compaction is performed again. If space is still not sufficient, the display file overflows and an error message is issued. Figs. 5.6-5.8 show how the minor and major compactions are performed. Fig. 5.6 gives the overall view of the display file, the display-file table and the DSNUM list before compaction. Fig. 5.7 gives the corresponding view after minor compaction and Fig. 5.8 shows the layout after major compaction.
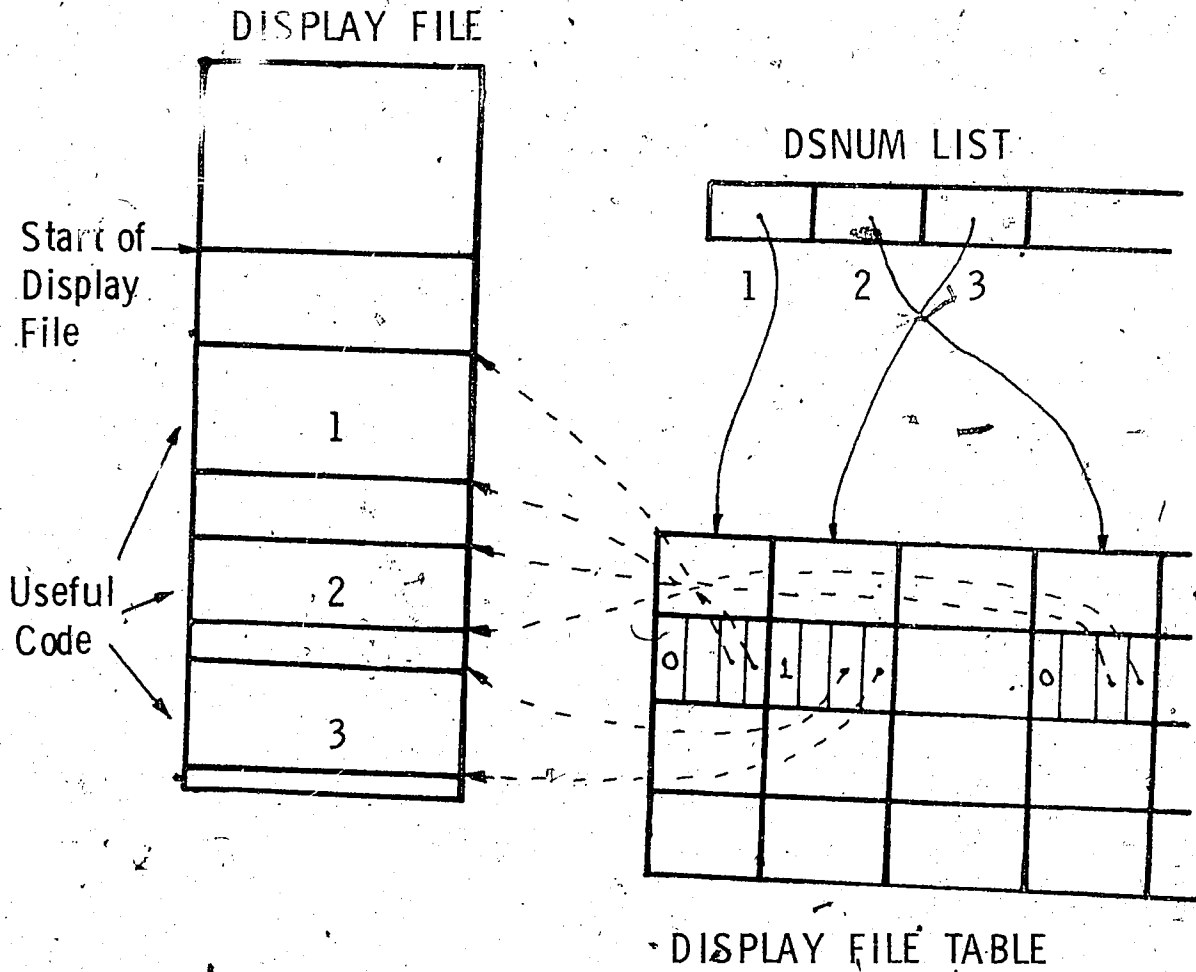
Fig. 5.6. The Display File, the Display File Table
and DSNUM List

DISPLAY FILE

DSNUM LIST

Start of
Display
File

1

2

3

1   2   3

FREE

SPACE

0 | | 1   1 | | 1   0 | |
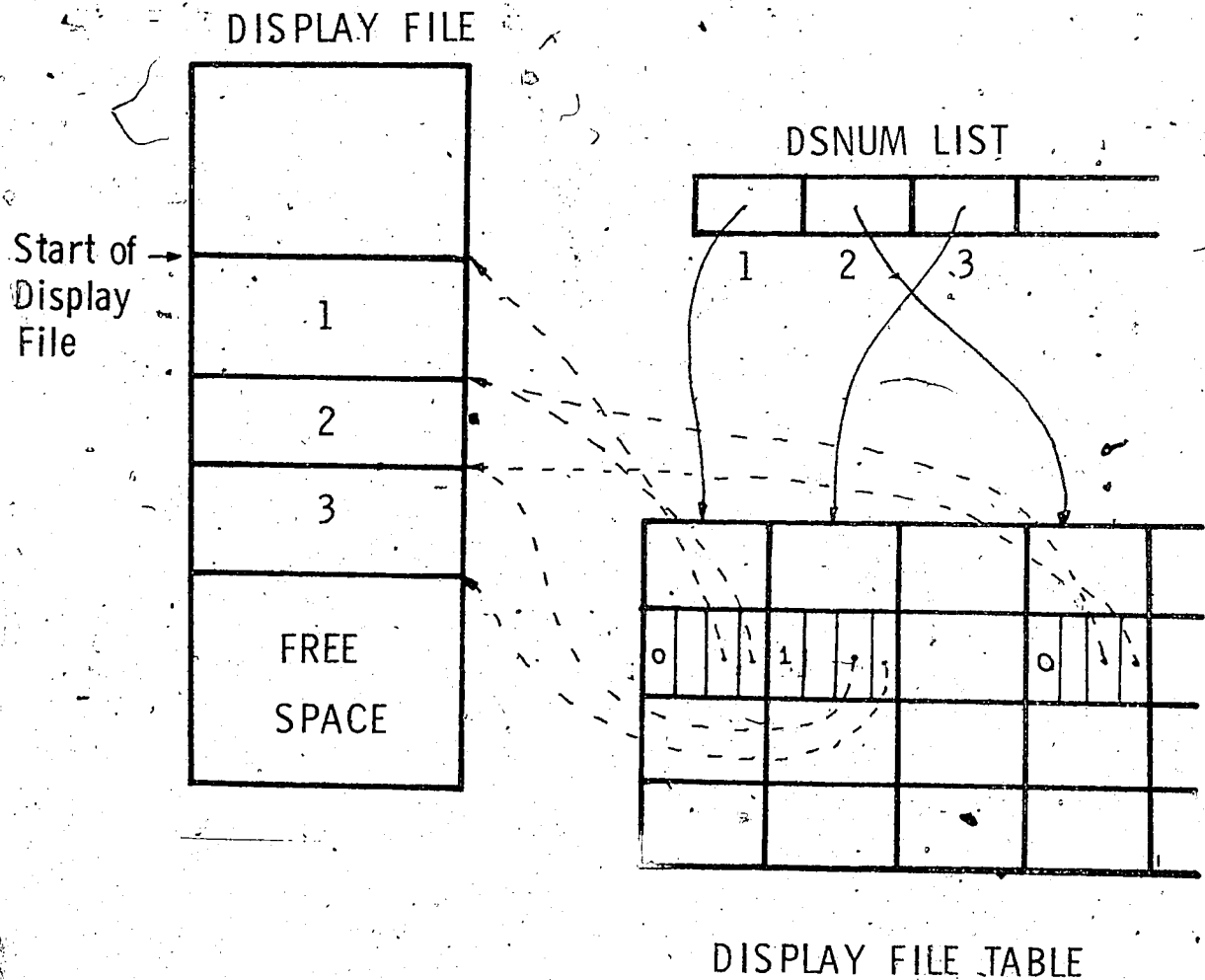
DISPLAY FILE TABLE

Fig. 5.7. The Display File, the Display File Table
and DSNUM List after Minor Compaction
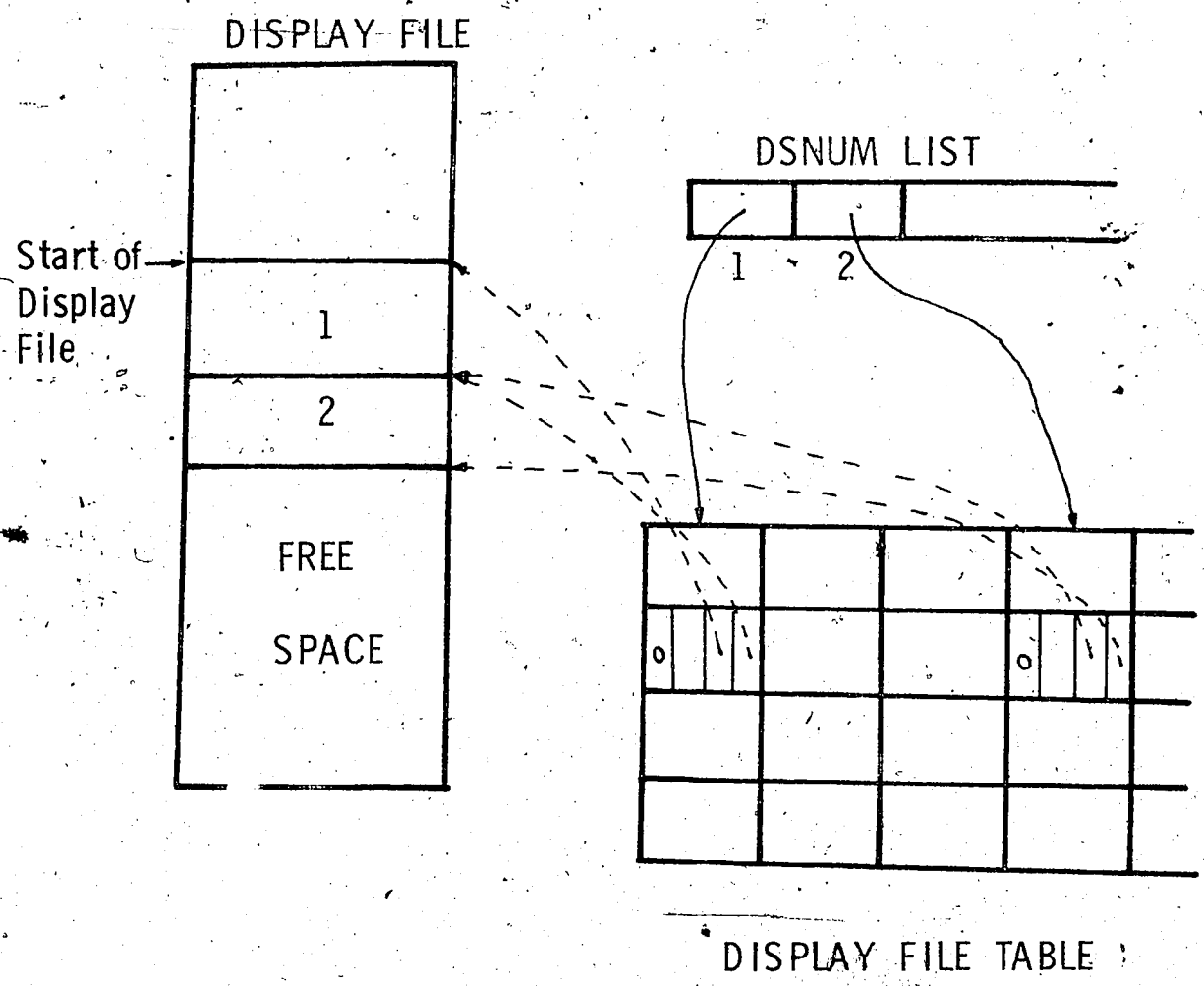
Fig. 5.8. The Display File, the Display File Table
and DSNUM List after Major Compaction

5.4. Decoding of Interrupts

Messages that have been composed by the user at the display terminal are sent to the 360 via the link and are analysed by the DECCDE routine. A table of the sequence and the type of interrupts is first set up. The type of interrupt expected, as specified by the DECODE statement, is then checked against the interrupt number, and the <tf> variable (see Section 3.2.3) is set to 0 or 1 accordingly. A value of 0 implies that the interrupt is as expected. The message caused by this interrupt is examined and the appropriate information is returned. A value of 1 implies that the interrupt is not as expected and no information will be returned.

The legal interrupts include: LPEN, POINT, VECTOR, KEYBRD, FKEY and END. A description of the information returned for the various interrupts follows:

(a) LPEN

This interrupt indicates that an object displayed on the screen has been picked by the light pen. The identification stack of the picture branch that is picked is returned in the <id vector> variable. At present, the deepest level of picture tree allowed is 10 levels. The coordinates where the light pen hit occured are returned in the <x> and <y> variables respectively.

(b) POINT and VECTOR

A series of points/vectors have been input at the display terminal using the light pen. It is taken as a picture component and is assigned to the picture specified by the <pict var> variable. The <x> and <y> variables are assumed to be arrays, and the X and Y coordinates of the points/vectors are saved in the two arrays as well.

(c) KEYBRD

A series of symbols have been input from the keyboard. It is considered as a picture component and is returned in the <pict var> as well. The coordinates of the starting position of the text input are returned in the <x> and <y> variables.

(d) FKEY

This indicates that a status key and a function key have been set. The value returned to the <inform> variable is a function of the status key and function key settings. The value is given by STATUS*10+FKEY where the STATUS value ranges from 1 to 15 (STATUS 0 is reserved for the GRID supervisor) and the FKEY value ranges from 0 to 9. Thus the value returned ranges from 10 to a maximum of 1509. The X and Y coordinates where the FKEY and STATUS value are displayed on the screen are returned in the <x> and <y> variables. No picture is returned.

(e) END

This imples that the SEND key has been pressed to indicate the end of the message. No variable other than <tf> is set.

5.5. Information Retrieval

The RETRIEVE routine essentially searches the picture data structure to retriev the appropriate information. The type of information that can be retrieved includes:

(a) TYPE

A value which indicates the type of the picture component. If the component is a picture, TYPE is 0; otherwise it is the value that represents the type of the primitive, as shown in Fig. 5.2.

(b) X

The X coordinate of the component, with the exception that if the component is of primitive type L, then the entire array of X coordinates is returned.

(c) Y

This works in the same way as X, but for the Y coordinate.

(d) NUMELEM

The number of picture components that the picture has.

(e) NUM

The number of vectors or symbols used in defining the

component. It is only valid if the specified component is of type L or T.

(f) PICTNAME

The picture name of the picture component if the component is a subpicture.

(g) LPDET, BLINK, BLANK, POS

value of the respective attributes (0, 1 or 2) of the specified picture component.

(h) DASH

The value of the DASH attribute. It is only valid if the picture component is of primitive type V or L.

(i) SIZE, ORIENT

The corresponding attribute value. They are valid only if the picture component is of primitive type S or T.

5.6. Function Declaration

Whenever a function declaration is encountered during, preprocessing the function name is entered into the function-name table. At present, the format of the function declaration statement is

FUNCTION <fctname> (<num para>

where <num para> is the number of parameters that is required for the function <fctname>. No checking on the type of the parameters is made. It is assumed that the user specifies them correctly.

A maximum of 128 functions (including system functions) is allowed in the function name table. Each entry has 5 words. The first two words are reserved for the name of the function; hence, a maxmimum of 8 characters is allowed for the function name. The actual limit on the length of the function name is imposed by the host langauge. Thus with FORTRAN as host, the function name cannot be more than six characters. The third word contains the number of parameters that is required by the function. The fourth word is a pointer from which the entry point of the function can be obtained. The fifth word is used to specify whether the function is system- or user-defined. A user-defined function overrides a system-defined function that has the same name.

The system-defined functions that have been implemented include:

(a) ROTATE (<theta>)

This function rotates the picture by an angle <theta> (in degrees) in the counter-clockwise direction.

(b) MOVE (<xdisp> , <ydisp>)

This function translates the pictures by the amount <xdisp> and <ydisp> in the X and Y directions respectively.

(c) SCALE (<xscale> , <yscale>)

This function scales up the picture by the factors <xscale> and <yscale> in the X and Y directions respectively.

(d) RSCALE (<xfact> , <yfact>)

This function reduces the size of the picture by the factors <xfact> and <yfact> in the X and Y directions respectively.

(e) PRIM.

This function merely converts the picture from the structured form to the sequential form consisting of display primitives only. It is useful for reducing the level of the picture tree and thus allowing more to be displayed on the screen.

5.7. The Preprocessor for FORTRAN Host Language

The preprocessor serves the dual purpose of separating the graphics statements from the Fortran statements and translating the graphics statements into the Fortran/graphics interface. As mentioned earlier, the graphics statements are distinguished from FORTRAN statements by the character '#' in column 1. The graphics statements can be placed anywhere between column 6 and column 72. The presence of the sign '-' in column 72 indicates that the next card is a continuation of the graphics statement. Each variable used in a graphics statement must be declared in COMMON (labelled or

unlabelled) blocks, with '#' in column 1. The addresses of
the variables are calculated from their displacements from
the beginning of the blocks.

The current restrictions on the format of th graphics
statement ar_ arbitrarily chosen. There is no technical
difficulty in changing them. For example, the continuation
column could be easily changed to column 2 or 3 so that it
can be entered easily at the termi-1.

The preprocessor works in two passes. In the first pass,
the graphics statements are detected and put into a
temporary file. Blocks of consecutive graphics statements
are modified to give CALLs in FORTRAN. Since it is not
necessary to introduce the variables into the CALLs, the
only parameter in the CALLs is used to provide the correct
entry point in the interface. Validity of the graphics
statements is checked by scanning for the keywords. All
variables (declared in COMMON blocks), picture names (which
appear on the left-hand side of picture definition
statements) and the function names (found in function
declaration statements) are entered into their respective
tables.

If no error is detected in pass 1, the graphics
statements in the temporary file are converted into CALLs to
the graphics software in the second pass. Currently, this
output is in 360 assembly language and has to be assembled

into object code in a separate step. Each graphics statement is converted to give the respective CALLs to the graphics software. Reference to picture names, variables and functions are now made to their respective tables.

Error messages are generated in both passes. Details of error messages can be found in the GRAF user's manual, which is in preparation.

5.8. Use of GRAF

5.8.1. Job Command Language

At present, the program of mixed FORTRAN and graphics statements has to be compiled in three steps before the object code can be executed. The program is first preprocessed to give the interface (in 360 assembler language), the tables used by the interface and graphics software (in object code) and the modified host (in FORTRAN). The interface is then assembled and the modified host compiled. The resulting object code can then be executed. The MTS commands for preprocessing, compilation and execution are as follows:

Step 1. Preprocessing

$RUN NNG.:GROBJ 5=MIXED 6=LIST 3=-HOST 4=-G 7=-INTER 8=X3

where   NNG.:GROBJ is the graphics preprocessor

MIXED is the mixed source program,

LIST   is the device for output listing,

-HOST is the temporary modified host,

-G is a temporary file used by the preprocessor,

-INTER is the temporary interface, and

X3   is the object code for the tables.

Step 2. Assemble the interface into object code

$RUN *ASMG SCARDS=-INTER SPUNCH=X2 0=*SYSMAC

where   X2   is the assembled interface routine.

Step 3. Compi the modified host

$RUN *FORTG SCARDS=-HOST SPUNCH=X1

where X1   is the compiled host.

Step 4. Execution of the graphics program

$RUN X1+X2+X3+NNG.:GRAF 8=NNG.:SUP

where   NNG.:GRAF is the graphics software package and
        NNG.:SUP is the graphics supervisor for GRID.

5.8.2. Programmed Examples

A Simple Demonstration Program

A program for the dis   of squares and triangles on the
screen has been written   GRAF.   It allows the user to
display as many squares and triangles on the screen as he

likes, by first hitting the command word 'SQUARE' or 'TRIANGLE' and then picking a point on the screen using the light pen. It also allows the user to delete all squares or triangles by first hitting the command word 'COMMAND' and 'SQUARE' or 'TRIANGLE' using the light pen respectively. A listing of the program is given in Appendix V.

This is a simple example. It is intended only to demonstrate that GRAF is a simple and compact language. The program has a total of only 49 statements (COMMENT statements not included), while a corresponding program in GRIDSUB requires 75 statements[17]. The main reduction is due to the more powerful picture description statements in GRAF. On the other hand, the total computer time used in preprocessing and compilation is substantially greater than that used for compiling the program in GRIDSUB. The time used for preprocessing is quite acceptable (just over 2 sec.) but the assembly of the interface takes 7.7 sec. This is the area where the GRAF facility needs improvement. By generating the interface directly in object code rather than in assembler language, the efficiency of the preprocessor would be much improved.

Another Example

Basically, this program has the same structure as the last one except that more commands have been added. It

allows the user to create and manipulate squares and triangles of all sizes and orientation by rotating, scaling (up or down) and translating the squares and triangles displayed on the screen. In spite of the limited display-hardware capabilities of the system, the program remains fairly simple. The entire program has a total of 108 statements (FORTRAN and graphics statements included). Some of the commands and how they are implemented using the graphics statements are exemplified below:

1) Command: pick the command word 'ROTATE',

pick any square on the screen,

type in the angle of rotation (in degrees).

GRAF statement:

SQS.N = SQS.N(X,Y) * ROTATE(ANGLE)

where SQS is the entire group of squares displayed,

N is the Nth square displayed,

X and Y are the origin of the Nth square, and

ANGLE is the angle of rotation.

2) Command: pick the command word 'SCALE UP',

pick any square on the screen,

type in the scale.

GRAF statement:

$$SQS.N = SQS.N(X,Y) * SCALE(SC,SC)$$

where   SC is the scale.

3) Command: pick the command word 'SCALE DOWN',

  pick any triangle on the screen,

  type in the scale.

GRAF statement:

$$TRS.N = TRS.N(X,Y) * RSCALE(SC,SC)$$

where   TRS is the entire group of triangles displayed.

4) Command: pick the command word 'MOVE',

  pick any triangle on the screen,

  pick a point on the screen.

GRAF statement:

$$TRS.N = TRS.N(A,B)$$

where   A   and   B   are   the   coordinates   of   the location
  picked.

A Logic-Circuit Application Program

For a more practical example, an application program has been written (see Appendix VI) which allows a user to design, modify and analyse logic circuits at the display terminal.

A menu is displayed on the left-hand side of the screen. The menu consists of a list of commands: ANDGATE, ORGATE, NOTGATE, DELETE, ATTACH, INPUT, OUTPUT, RESTART and ERASE. The logic-circuit components include AND, OR and NOT gates. The AND gate is represented by a semi-circle with two input leads and one output lead. The OR and NOT gates are represented by triangles with input and output leads, and a '+' sign is displayed inside the OR gate to differentiate it from the NOT gate, as shown in Fig. 5.9.

The gates can be displayed on the screen by picking the commands ANDGATE, ORGATE, or NOTGATE and then picking a point on the screen. The input and output leads of the gates can be interconnected using the ATTACH command, by first picking the input lead, and then the output lead. Gates and interconnecting links can be deleted by first picking DELETE on the menu and then the gate or the link that is to be deleted. Inputs allowed are '0' and '1', representing the values 'false' and 'true' respectively. They can be assigned to any input lead by first picking the INPUT command from the menu, then the input lead, and finally typing in the value via the keyboard. The inputs can be nullified using the ERASE command by picking the input lead and the input value displayed. The OUTPUT command is used to generate the output of gates and displayed them cn the screen. Fig. 5.10. shows a completed circuit with the input and output displayed.

```
ST/FK                    STATE

ANDGATE
ORGATE
NOTGATE
DELETE
ATTACH
INPUT
OUTPUT
RESTART
ERASE
```
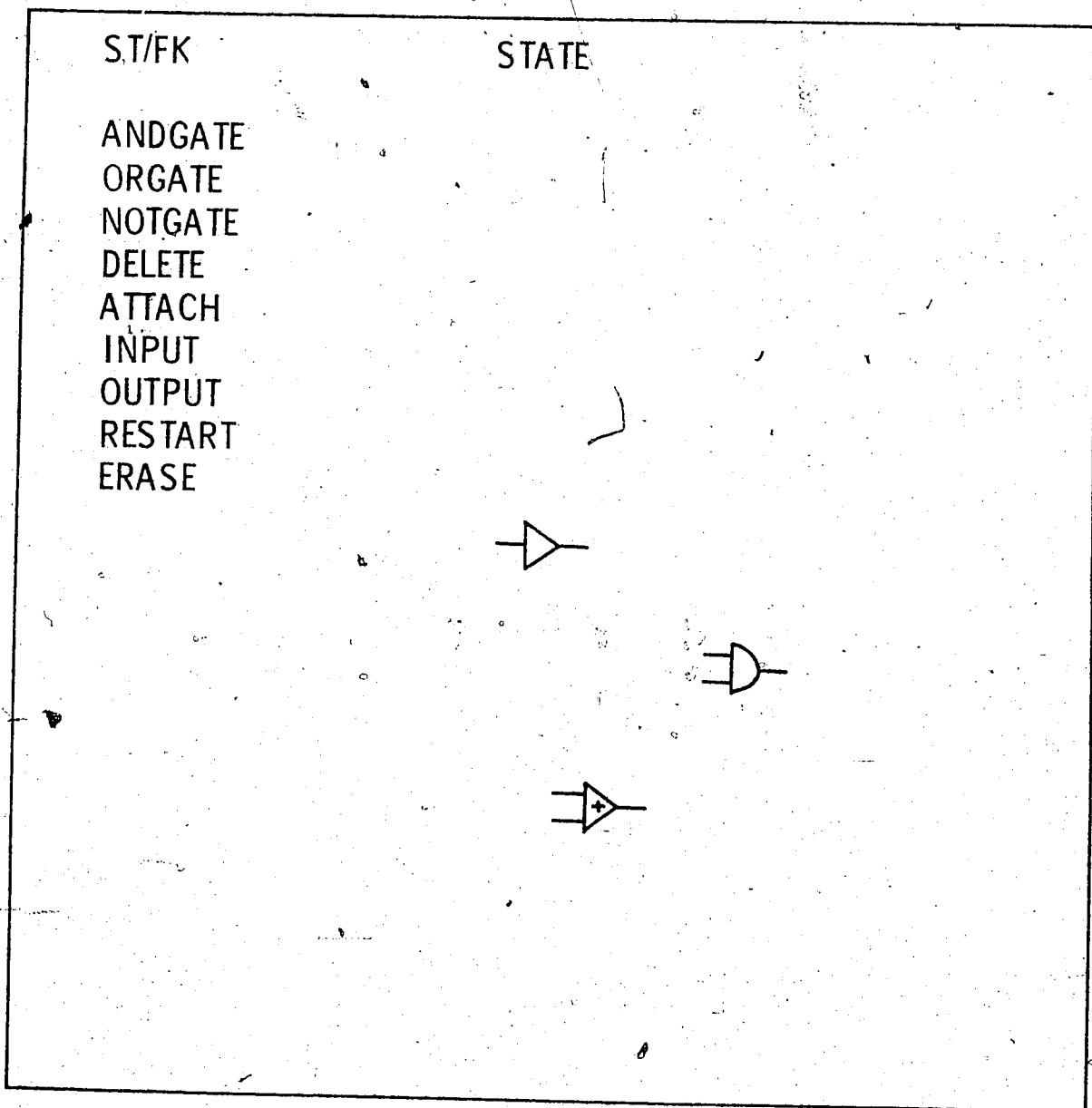
Fig. 5.9. Logic-circuit Design -- AND, OR, NOT Gates

ST/FK   STATE

ANDGATE
ORGATE
NOTGATE
DELETE
ATTACH
INPUT
OUTPUT
RESTART
ERASE
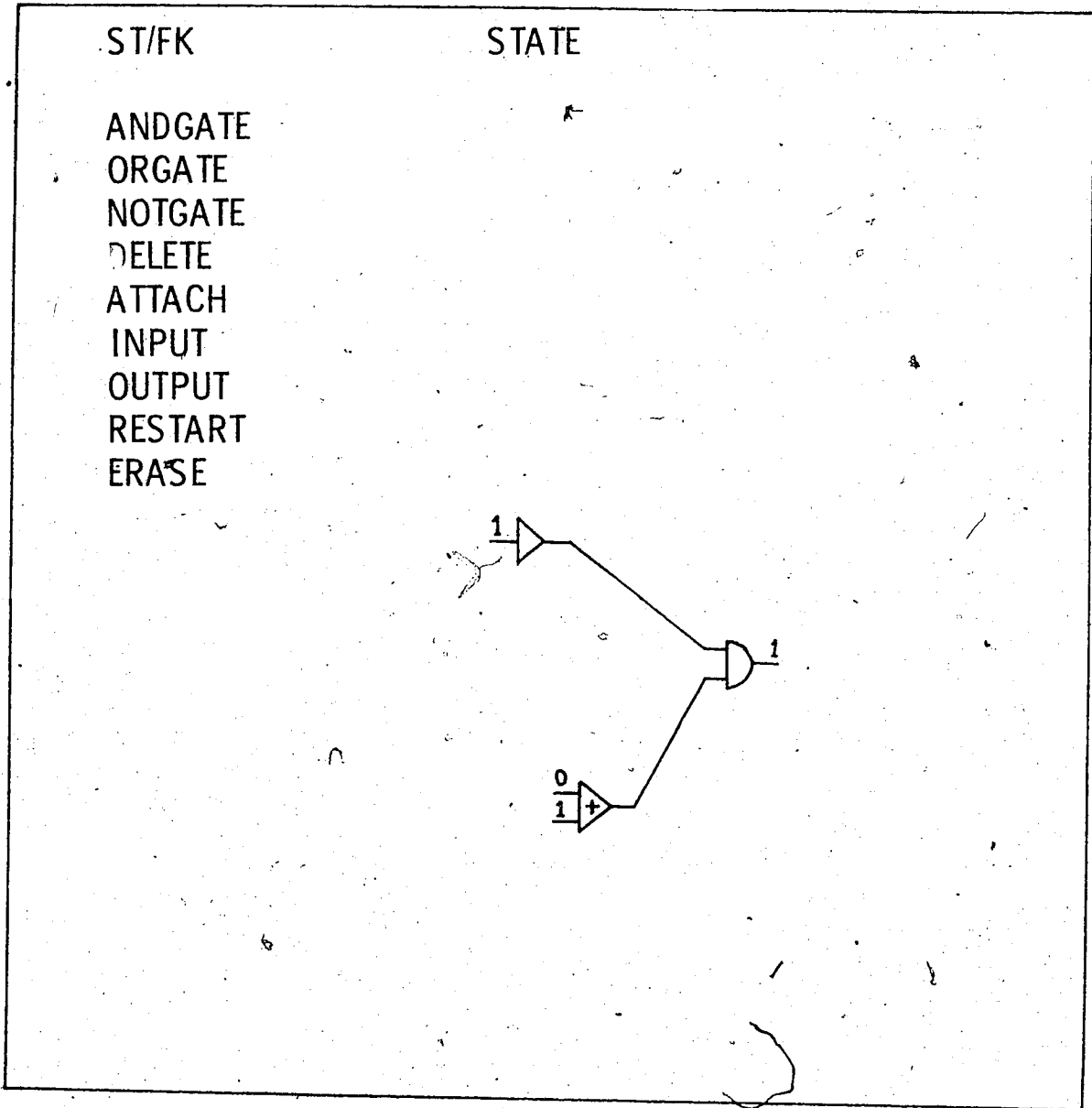
Fig. 5.10. Logic-circuit Design -- A Completed Circuit

The only data structure maintained in the FORTRAN host is a matrix used to indicate how the gates are interconnected: this is essential for the logic-circuit analysis performed in the host language. Every time a change is made to the circuit, the data structure has to be appropriately updated. However, all other picture information is obtained using RETRIEVE and DECODE statements. It can be seen that in spite of the complexity of the application, the size of the program is easily manageable.

Discussion

The implementation of GRAF has been discussed in this chapter. Programming examples are also given. It should be observed that it is easy to program in GRAF. The fact that the size of the programs is small makes debugging of the programs relatively simple. The small number of statements also helps to reduce the chance of typing and logical errors.

Chapter VI

Extension to Picture Analysis

6.1. Picture Generation vs Analysis

So far, the picture description scheme has only been used for the purpose of generating pictures. Several authors[26] [27] [34] have argued that picture description schemes should be applicable to both picture generation and analysis. An immediate question arises: Can this picture description scheme, or a simple extension of it, be used for analysing certain classes of pictures?

One important difference between picture generation and analysis is that an explicit specification of the picture is required for generation purposes. For example, we may be interested in generating a square of sides 3 inches at the centre of the screen. The shape, the size and the location of the object are explicitly given. The description is simply a command to the output device. Based on the description, a replica of the picture is produced. The specification of the description thus has to be explicit, if any definite picture is to be generated.

On the other hand, the aim in picture analysis most frequently is to determine whether a given picture is a fascimile of the description, or whether the picture belongs to a certain class of pictures. Often, the problem posed

would be: Is there a square in the picture? If there is, where is it located and how large is it? One approach which has often been used is to base the process of analysis on the descriptions of classes of pictures, in a manner analogous to that used for syntax analysis of programming languages. In this linguistic approach to picture analysis[32], it is necessary that the descriptions of the pictures contain sufficient information to guide the analyser in recovering pictures that fit the description.

The picture description scheme presented in this thesis has been sucessfully used for picture generation. The fact that it contains a general model within which line-like pictures can be described makes it a possible candidate for picture analysis as well. In the present scheme, the attributes of pictures are described in terms of values, and it is therefore not general enough for analysis of pictures, other than artificial pictures such as those generated by the scheme itself. However, a simple extension of the scheme allows for the description and analysis of a large class of line drawings.

An approach is proposed here for analysing pictures using the extended picture description scheme and is discussed in the following sections. Shaw's PDL[34], which has been used successfully for analysing several meaningful classes of pictures, is used for comparison purposes.

6.2. Picture Analysis using an Extended Picture Description Scheme

6.2.1. General Concepts of the Scheme

The main concept of the picture description scheme is that pictures are represented as tree-like structures. Each picture is defined in terms of its components, which in turn have their own structures (the syntax) and attributes. The components can be subpictures or primitives. The only operator for constructing pictures (in the sense of its syntax) is the '+' operator. The connectivity of the picture components is not obvious unless they are primitives. However, the position attributes do provide the type of information that can be suitably used for analysing pictures.

6.2.2. Extension of the Picture Description Scheme

If the position attributes can only have fixed values, then only one particular picture is described at a time. However, if the syntax of the picture description scheme is extended so that attributes can be represented as variables or a function of variables, the scheme can be used to describe classes of pictures.

For example, the class of squares can be described as:

$$SQUARES = V(X,0) + V(X,X) + V(0,X) + V(0,0)$$

where X is a variable. In fact, it describes squares of all sizes. Similarly, the class of rectangles can be described as:

$$RECTS = V(X,0) + V(X,Y) + V(Y,0) + V(0,0)$$

where X and Y represents the lengths of the sides of the rectangle. For another example, the class of isosceles triangles can be described as:

$$ISOTRI = V(2*X,0) + V(X,Y) + V(0,0)$$

It should be noted that allowing arithmetic expressions in place of the variables will not cause any ambiguities in the syntax of the picture definition statement.

To further enhance the power of the description scheme, the parameters used in transformations are also extended so that they can be specified as variables (or as functions of variables). The picture description scheme now becomes more versatile. Thus, rotated squares can be described as:

$$SQS = SQUARES(A,B) * ROTATE(THETA)$$

where A, B, THETA are all variables and SQUARES is as defined before. This in fact describes any square in two-dimensional space.

6.2.3. An Approach for the Analysis of Pictures

Based on the extended picture description scheme, an approach to picture analysis can be developed. The linguistic approach is still used as the basis of operation, with one further criterion employed. The position attributes are used as constraints which have to be satisfied during the recognition process. Also, the relative positions between the picture components are used as directional guides so that the whereabout of the next component can be located easily, which is similar to the use of "suitable" primitives (which could be blank) in Shaw's PDL.

To determine, say, if there is a member of the class "SQUARES" at a certain location of the picture, the analysis proceeds as follows:

Starting at the location, and using it as the local origin, first determine if there is any line in the +X direction, as indicated by the position attributes of the first component of SQUARES. Having found the line, the value of X is recorded. The minimum value of X where the line is intersected by another line is the first value used. Once the value of X has been set, it imposes a constraint on all other references to the same variable, i.e. X. It then determines whether the next line is perpendicular to the X-axis, as indicated by the second component of SQUARES, and

whether the line is of the same length. In a similar fashion, it is determined whether all the remaining conditions are satisfied. If so, the searc. :ops, and a vertical square of sides X units is found. Otherwise, the search is brought back to the point where the X value is first recorded. A new value of is sought by continued searching in the X direction. The above process is repeated whenever another value of X can be found. When all possible values of X have been used and no SQUARES can be found, the process will report failure.

Thus, not only is the syntax of the description scheme used, the position attributes also provide the information for directing the analysis process. The variables form constraints on the system which have to be satisfied during the recognition process. How well the constraints should be satisfied is determined by the requirement of the analysis. Some relaxation of the constraints is often necessary for real pictures. Using the top-down picture parsing technique[26] as proposed by Shaw, the analysis can be used for structured pictures as well.

For recognition of function-transformed pictures, the process can proceed in a similar fashion. The procedure can become very clumsy and impractical if too many variables are used and the transformation is non-linear. However, if the transformation is linear, the analysis can be similarly

carried out.

For example, to detect a rotated square as defined in the last section, there are at least two constraints to be satisfied, i.e. the angle of rotation THETA and the length of the sides X, assuming that the starting point (A,B) can be located. However, since the square can be rotated to any angle, if only the end points of the square are noted the relations as indicated by the description may not be found. Two approaches can be taken to test whether the relationships are satisfied. Either a rotation transformation can be performed on the end points, or the end points can be used to calculate the angle of rotation and the lengths of the sides. The second approach is probably more suitable in this case.

The recognizer will start searching from the X-axis in a counter-clockwise direction, say, until a line is detected. The slope and length of the line are noted. These two values will be used as the constraints that have to be satisfied in searching for the remaining sides. Thus, if the angle of the first side is found to be 30 degrees, then the second side must be at 120 degrees, and the third and fourth sides must be at 210 and 300 degress respectively.

From the above discussion, it can be seen that the analysis requires only a means of detecting straight lines, and calculating their lengths and slopes. If polar

coordinates are used in the descriptions, the scheme is even simpler.

6.3. A Comparison with Shaw's PDL

Shaw's PDL[32] is essentially a language for describing line drawings which can be represented as connected graphs. Use of blank primitives also allows description of pictures which have disconnected parts. Each primitive has a head and a tail which can be concatenated to other primitives. A number of operators are provided for various head and/or tail concatenations.

In PDL, the basic components are the primitives which, by definition, can be virtually any picture. It is important that such primitives be chosen properly lest the pictures cannot be described, due to the limited possible concatenations. The structure of the description scheme is also tree-like. Although transformation specifications are included in the original definition of the language[24], they have been used to a very limited extent only; e.g. translation of pictures in picture generation (George[9]).

PDL has been used successfully for parsing pictures such as in Spark Chamber Film Analysis[32]. A number of interesting examples of picture generation have also been demonstrated. The main concept used is that if a head or

tail of a picture can be found, it can be used as the reference location for further searches, since all picture parts are connected.

The present Picture Description Scheme (hereafter refer to as PDS in this chapter) differs from PDL in that explicit coordinate specifications are used. Nevertheless, although PDL allows the coordinates of head and tail of pictures to be specified abstractly, the coordinates do exist physically when the primitives are defined. Structurally, the two languages are basically the same. In fact, simulation of PDL can be carried out in PDS and the various concatenations are demonstrated in Fig. 6.1. Part (a) shows the actual drawing, part (b) gives the description in PDL and part (c) gives the corresponding description in PDS. There is little doubt that PDL provides a more compact specification. However, the description in PDS is very general and covers a wide class of pictures. The question of compatibility of primitives does not arise, and it can also be seen that similar pictures (such as case 4 and case 5) are described in a uniform manner in PDS while the description in PDL differs widely.

To give a more realistic example, consider the use of the two schemes to describe the English letter "A". In PDL, this can be described as:

$$"A" = (a + ((b * (a + c)) + c))$$

where 'a', 'b' and 'c' are the primitives as shown in Fig. 6.2.

In PDS, the same picture can be described as (Fig. 6.3):

"A" = V1(0,0) + V2(0,0) + V3(-X,-Y).

where   V1 = V(-2*X,-2*Y)

         V2 = V(2*X,-2*Y)

         V3 = V(2*X,0)

and the origin is taken to be the top of the character.

Alternatively, it can be described in terms of the primitive V alone as:

"A" = V(-2*X,-2*Y) + V(-X,-Y,blanked) + V(X,Y)

        + V(0,0,blanked) + V(2*X,-2*Y)

| Case 1 | $a + b$ | $A(0,0) + B(X,Y)$ |

| Case 2 | $a \times b$ | $A(0,0) + B(0,0)$ |

| Case 3 | $a - b$ | $A(0,0) + B(U,V) *$ ROTATE(-ALP) where ALP=THETA+TAN^{-1}(V/(X-U)) |

| Case 4 | $a * b$ | $A(0,0) +$ $B(0,0) *$ ROTATE(THETA) where THETA = tan$^{-1}(Y_b/X_b)$ |

| Case 5 | $(a+b)*\lambda$ | $A(0,0) +$ $B(X,Y) *$ ROTATE(THETA) |
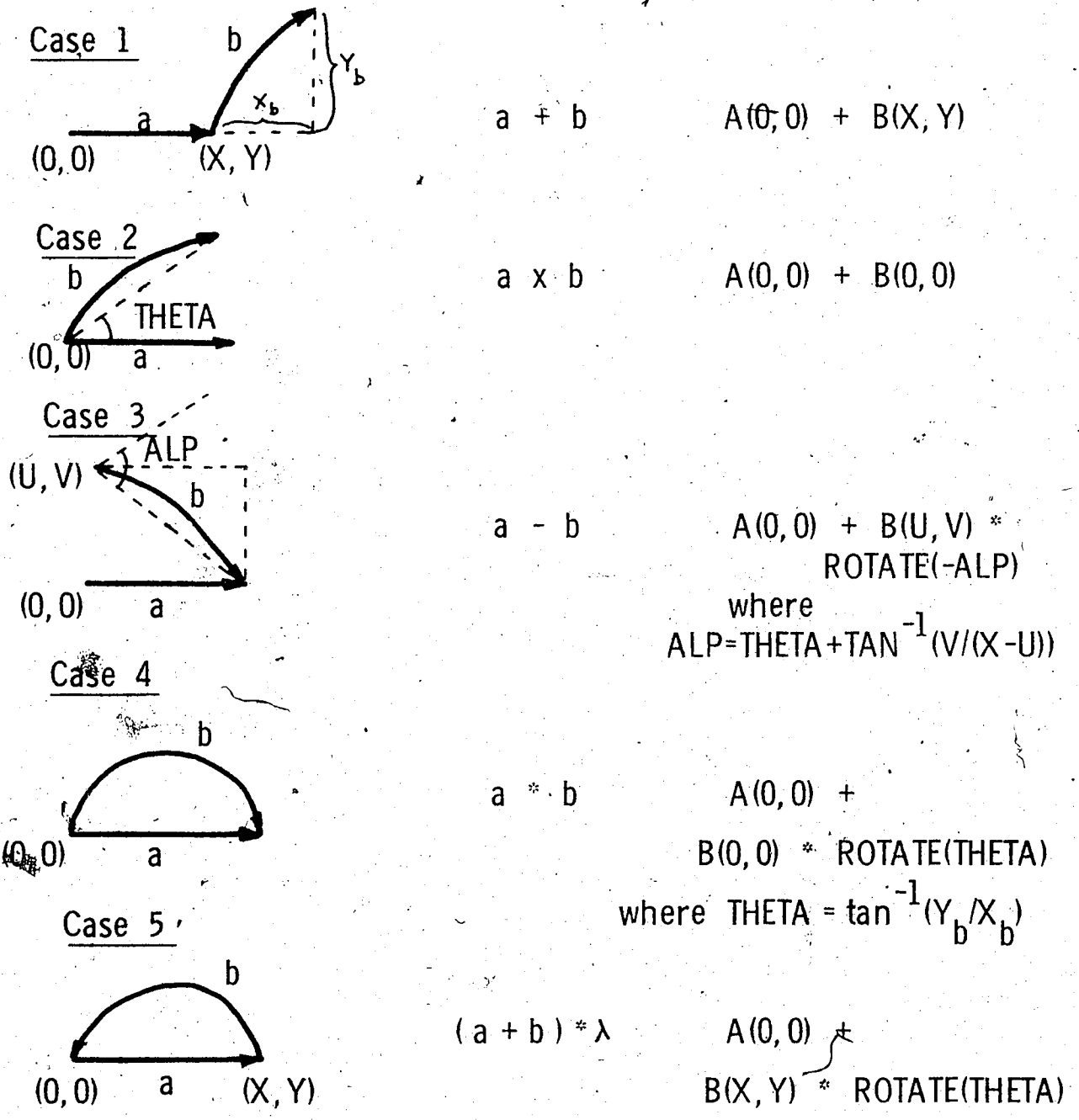
(a) Actual Picture　　　(b) PDL Description　　(c) PDS Description

Fig. 6.1.　Simulation of PDL using PDS
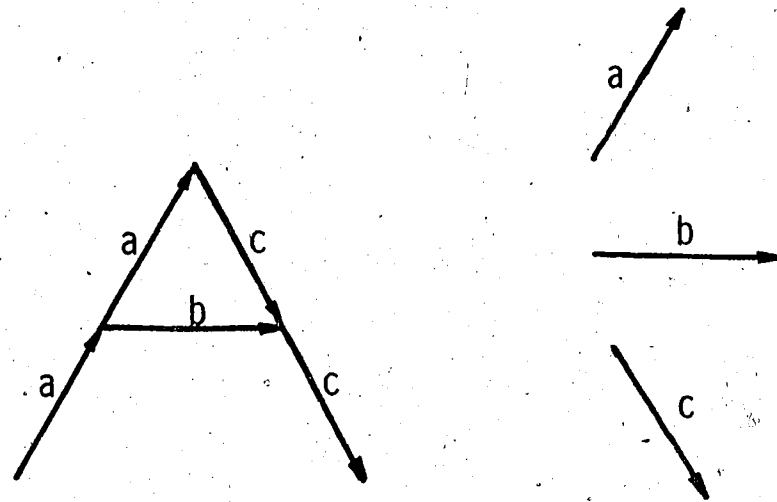
$$"A" = (a + ((b * (a + c)) + c))$$

Fig. 6.2. PDL Description of "A"
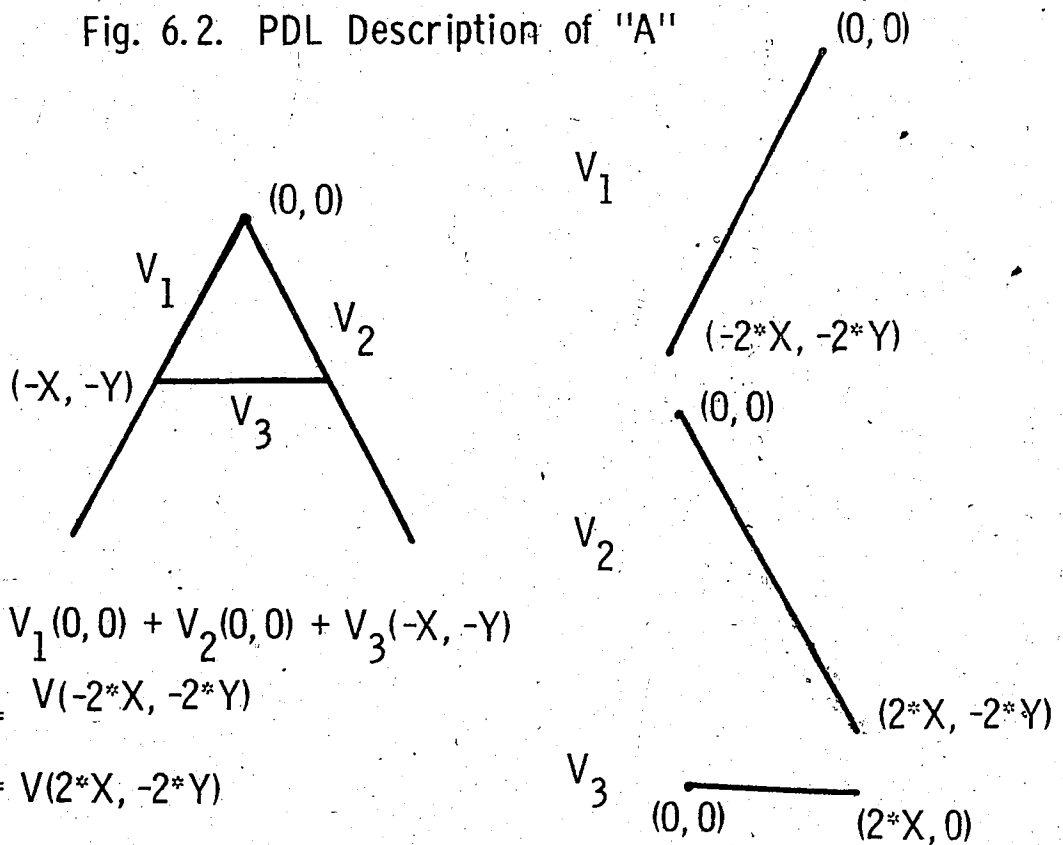
$$"A" = V_1(0,0) + V_2(0,0) + V_3(-X,-Y)$$

$$V_1 = V(-2*X, -2*Y)$$

$$V_2 = V(2*X, -2*Y)$$

$$V_3 = V(2*X, 0)$$

Fig. 6.3. PDS Description of "A"

Although the description in PDL is again more compact, the PDS offers a more user-oriented and general description. In fact, the description is made in the same way as it is usually hand-written. There is no limit on the size of the picture. Also, it does not matter what the values of X and Y are. So long as the constraints are satisfied, the shape "A" will be recognized.

6.4. A PDS Grammar

For the purpose of describing classes of pictures, the description is best expressed as a picture grammar. Since PDS describes pictures using explicit position attributes, picture grammars in PDS can be quite involved. However, for pictures with regularities, the description is often fairly straightforward. For example, the class of R-C filter stages (see Fig. 6.4) can be expressed in a PDS grammar as:

FILTERSTAGES::= FILTER(0,0) | FILTER(0,0) + FILTERSTAGES(X,Y)

FILTER::= RESISTOR(0,0) + CAPACITOR(X,Y) + LINK(U,V)

where FILTER is as shown in Fig. 6.4.

For picture classes that do not have such regularities, the grammar is much more complex, and the choice of picture parts and origins may play an important role. The power and limitations of PDS for picture analysis purposes will depend on how such problems can be handled, and is an area for future research.
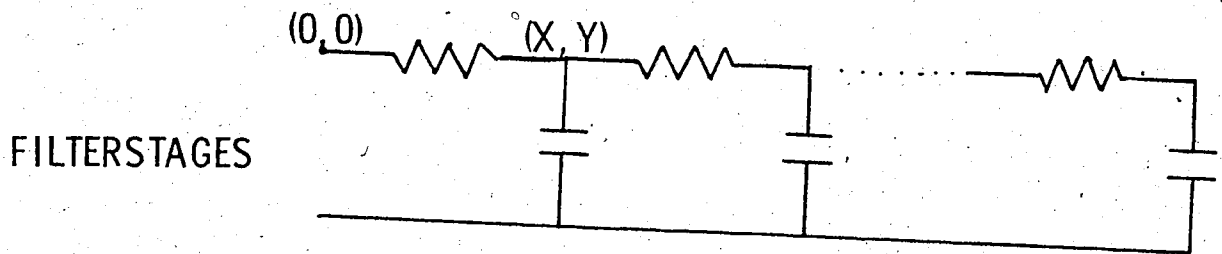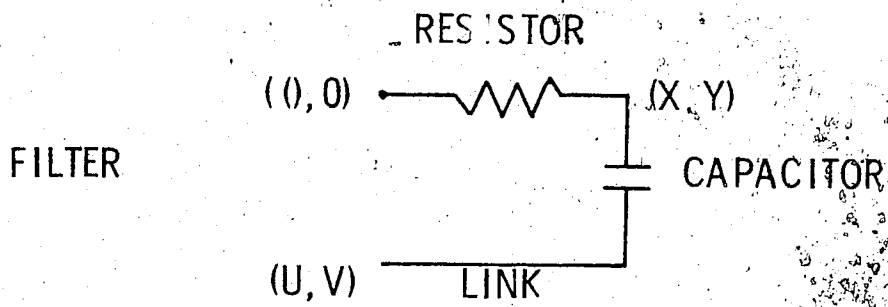
Fig. 6.4. R-C Filter Stages

6.5. <u>Discussion</u>

From the discussion in this chapter, it can be seen that the present PDS can be easily extended to describe classes of pictures. The extended picture description scheme, together with the constraints formed by the position attributes, can be suitably used for picture analysis purposes.

Chapter VII

Conclusion

7.1. Summary of Results

The design and implementation of a graphics facility has been presented in this thesis. Two important results are revealed. First, a convenient and useful graphics language has been designed and implemented. Second, the concept of a Universal Graphics Facility has been presented and partially implemented.

The specifications of the graphics language are simple and modular. Five types of graphics statements are used to handle the different aspects of interactive graphics. The picture description scheme is powerful. Automatic indexing of the picture components provides much flexibility and easy identification. Picture components can be easily modified using the picture component selection feature. The concept of attribute hierarchy allows easy construction of copies of the same picture which have different hardware attributes. Specification of transformation within picture definition statements also provides a more user-oriented description of pictures.

The overall design and implementation of GRAF are aimed at providing a graphics software support system that is as display-hardware independent as possible. The way the

display files are handled provides a canonic transformation of structured display files to sequential display files. The construction is such that the display file software is at a level close to hardware and replaceable by hardware when required. The DECODE, RETRIEVE and FUNCTION statements have also been implemented so that they can be easily extended when new hardware is added to the system or when usage warrants it.

The concept of Universal Graphics Facility provides a more suitable environment for the graphics application programmers. Less effort is required from them since they have to learn only the graphics statements. It is equally beneficial to the graphics-software designer since it is not necessary to modify the host in any way. Only the graphics operations have to be taken care of, independent of the properties of any host language.

7.2. Future Work

GRAF has only been made ready for use recently. Although the author has found it easy to use, its impact on the user remains to be seen. More application programs have to be written before the extent of usefulness of the facility can be determined. Besides, more preprocessors for other host languages have to be implemented before a truly Universal Graphics Facility is fully achieved.

The main drawback of GRAF at present is that the host/graphics interface is produced in assembler language. Assembly of the interface is the most inefficient phase of the system. It is desirable that this interface be produced in machine code directly. Apart from the overhead in preprocessing, GRAF operates with good efficiency in spite of repeated reconstruction in the course of implementation. This can be attributed to the simplicity of the GRAF system. However, there is still room for improvement. Some re-organization of the software routines, and in particular the various tables, would definitely provide better efficiency. Some improvement in the transmission of the directory routines to the GRID would help to provide better response at the display terminal. At present, all the directory routines are transmitted each time the display file is updated. More transformation functions, such as windowing, etc., and retrieve routines would also provide better facilities to the user. Relaxation of the restriction that variables must be of integer type is also desirable.

Finally, the application of the picture description scheme to analyse pictures is an area for future research. The discussion in Chapter 6 indicates that the picture description scheme appears extensible for picture analysis purposes. Large classes of pictures, and in particular pictures with disconnected parts, are suitable candidates for such applications.

References

1. Boehm, B.W., Lamb, V.R., Mobley, R.L., Rieber, J.E., "POGO: Programmer-Oriented Graphics Operation", AFIPS Conference Proceedings, SJCC 1969, pp 321 - 320.

2. CALCOMP, Inc., "Software Reference Manual", California Computer Products Inc., Anaheim, California, 1969.

3. CDC, "GRID Engineering Specification", Data Display Division, CDC, 1968.

4. Deecker, G.F.P., "Computer Graphics and Campus Planning", M. Sc. Thesis, Department Of Computing Science, University Of Alberta, 1970.

5. Dodd, G.G., "APL -- A Language for Associative Data Handling in PL/I", AFIPS Conference Proceedings, FJCC, 1966, pp 677 - 684.

6. Duffin, J.D., "A Language for Line Drawing", Ph.D. Thesis, University of Toronto, Department of Computer Science, May 20, 1970.

7. Feder, J., "PLEX Languages", Information Sciences, Vol. 3, July 1971, pp 225 - 241.

8. Frank, A.J., "B-Line, Bell Line Drawing Language", AFIPS Conference Proceedings, FJCC, 1968, pp 179 - 191.

9. George, J., "Picture Generation Using the Picture Calculus", Stanford Linear Accelerator Centre, Computer Group, GSG 50, 1967.

10. Gosden, J.A., "Software Compatibility: What Was Promised, What We Have, What We Need", AFIPS Conference Proceedings, FJCC, 1968, Vol. 1, pp 81 - 87.

11. Gray, J.C., "Compound Data Structure for Computer Aided Design - A Survey", Proceedings ACM 22nd Nat.

Conf., 1977, pp 355 - 365.

12. GSP, "IBM System/360 Operating System, Graphic Programming Services for Fortran IV", Form C27-6932.

13. Hagan T.G., and Stotz R.H., "The Future of Computer Graphics", AIFPS Conference Proceedings, Vol. 40, SJCC 1972, pp 447-452.

14. Hurwitz, A., Citron, J.P. and Yeaton, J.B., "GRAF: Graphic Additions to Fortran", AFIPS Conference Proceedings, SJCC, 1967, pp 553 - 557.

15. IBM Systems Reference Library, IBM System /360 Operating System, Fortran IV(G And H) Programmer's Guide, Form GC28-6817-2, 1970.

16. IBM Systems Reference Library, IBM System /360 Operating System, PL/I(F) Programmer's Code, Form GC-28-6594-7, Jan. 1971.

17. Jackson, W.C., "Computer Graphics for The Application Programmer", Reference Manual for the IBM 360/GRID Graphics System Software, Department of Computing Science, University of Alberta, Feb. 1972.

18. Kirsch, R.A., "Computer Interpretation of English Text and Picture Patterns", Transaction IEEE, Vol. EC-13, Aug. 1964, pp 363 - 376.

19. Knowlton, K.C., "A Programmer's Description of L⁶", Comm. of ACM, Vol. 9, No. 8, Aug. 1966, pp 616 - 625.

20. Lang, C.A., and Gray, J.C., "ASP -- A Ring Implemented Associative Structure Package", Comm. of ACM, Vol. 11, No. 8, Aug. 1968, pp 550 - 555.

21. Ledley, R.S., "High-Speed Automatic Analysis of Biomedical Picture", Science, Vol. 146, 1964.

22. Machover, C., "Computer Graphics Terminal-A Backward Look", AIFPS Conference Proceedings, Vol. 40, SJCC 1972, pp 439-446.

23. Mezei, L., "SPARTA, A Procedure Oriented Programming Language for the Manipulation of Arbitrary Line Drawings", Proceedings IFIP Congress 1968, Vol. 1, pp 597 - 604.

24. Miller, W.F., and Shaw, A.C., "A Picture Calculus", Emerging Concepts in Computer Graphics, 1967 University of Illinois Conference, edited by Secrest, D. and Nievergelt, J., W.A. Benjamin, Inc., 1968.

25. Miller, W.F., and Shaw, A.C., "Linguistic Methods in Picture Processing -- A Survey", AFIPS Conference Proceedings, FJCC, 1968, Vol. 33, Part 1, pp 279 - 290.

26. Nake, F., "A Proposed Language for the Definition of Arbitrary Twodimensional Signs", Pattern Recognition in Biological and Technical Systems, New York, 1971, pp 396 - 402.

27. Narasimhan, R., "Labelling Schemata and Syntactic Description of Pictures", Information and Control, Vol 7, No. 2, June 1964, pp 151 - 179.

28. Narasimhan, R., "Syntax-Directed Interpretation of Classes of Pictures", Comm. of the ACM, Vol. 9, No. 3, March 1966, pp 166 - 173.

29. Newman, W.M., "Display Procedures", Comm. of the ACM, Vol. 4, No. 10, Oct. 1971, pp 651 - 660.

30. Newman, W.M., and Sproull, R.F., Principles of Interactive Computer Graphics, McGraw Hill, 1973.

31. Ng, N., "A New Graphics Language", University of Alberta Computing Review, Vol. 5, 1972, pp 8 - 19.

32. Penny, J.P., Deecker, G.F.P., Ng, N., "On General-Purpose Software for Interactive Graphics", Proceedings of 5th Austrian Computer Conference, Australia, May 1972, pp 238 - 244.

33. Rovner, P.D., and Feldman, J.A., "The LEAP Language and Data Structure", Proceedings IFIP Congress, Vol. 1, 1968, pp 579 - 585.

34. Shaw, A.C., "The Formal Description and Parsing of Pictures", Ph. D. Dissertation, Department of Computer Science, Stanford University, April 1968.

35. Shaw, A.C., "A Formal Picture Description Scheme as a Basis for Picture Processing Systems", Information and Control, Vol. 14, No. 1, Jan. 1969, pp 9 - 52.

36. Shaw, A.C., "Parsing of Graph Representable Pictures", Journal of ACM, Vol. 17, No. 3, July 1970, pp 453 - 481.

37. Sites, R.L., "ALGOL W Reference Manual", STAN-CS-71-230, Computer Science Dept., Stanford University, Aug. 1971.

38. Smith, D.N., "GPL/I - A PL/I Extension for Computer Graphics", AFIPS Conference Proceedings, SJCC, 1971, pp 511 - 530.

39. Sulkers, P.R., GRID Supervisor Multibank Extension Version 5.0.0, Jan. 1973, Department of Computing Science, University of Alberta.

40. Sutherland, I.E., " SKETCHPAD : A Man-Machine Graphical Communication System", AFIPS Conference Proceedings, SJCC, 1963, pp 329 - 346.

41. Sutherland, W.R., "On-Line Graphical Specification of Computer Procedures", Tech. Rep. 405, Lincoln Lab., MIT, Lexington, Mass., May 1966.

42. Van Dam, A., and Evans, D., "Data Structure

Programming System", Proceedings IFIP Congress 1968,
Vol. 1, pp 557 - 564.

43. Williams, R., "A Survey of Data Structures for
Computer Graphics Systems", Computing Surveys, Vol. 3,
No. 1, March 1971, pp 1 - 21.

44. Williams, R., "A General Purpose Graphical Language",
Tech. Rept. AD-746 688, New York University at
Bronx, N.Y., April 1972.

APPENDIX I Syntax Specification of the Graphics Language

1. The Graphics Language

    <graphics statement> ::= <pictdef statement>

                        | <draw statement>

                        | <decode statement>

                        | <retrieve statement>

                        | <fct decl statement>

2. Picture Definition Statement

    <pictdef statement> ::= <pict var> = <pict exp>

    <pict var> ::= <pictname> | <pict selector>

    <pictname> ::= <alpha> | <pictname> <alphanumeric>

    <pict selector> ::= <pictname> . <selector>

    <selector> ::= <integer> | <integer variable>

    <pict exp> ::= <pict comp> | <pict exp> + <pict comp>

    <pict comp> ::= <pict var> | <valuated pict>

                    | <pict prim> (<attri2>)

                    | NULL

    <valuated pict> ::= <pict var> (<attri1>)

                    | <valuated pict> * <transform>

    <pict prim> ::= P | V | S | L | T

    <attri1> ::= <pos attri> , <display attri>

    <pos attri> ::= <x> , <y> , <pos>

    <display attri> ::= <z> | <display attri> , <z>

    <x> ::= <variable>

```
<y>      ::=    <variable>

<z>      ::=    0  |  1  |  2

<attri2>  ::=  list of attributes for primitives

<transform>  ::=  <fct name> ( <para> )

<fct name>  ::=  <alpha>  |  <fct name> <alphanumeric>

<para>  ::=  <variable>  |  <para> , <variable>

<integer>  ::=  <numeric>  |  <integer> <numeric>

<integer variable>  ::=  <alpha>

              |  <integer variable> <alphanumeric>

<variable>  ::=  value or variable known to host language

<alpha>  ::=  A | B | C | D | E | F | G | H | I | J

             | K | L | M | N | O | P | Q | R | S | T

             | U | V | W | X | Y | Z

<numeric>  ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<alphanumeric>  ::=  <alpha>  |  <numeric>
```

3. Picture Output

```
<draw statement>  ::=   DRAW  ( <output device> ,
              <attri pict> )

<output device>  ::=  <integer>  |  <integer variable>

<attri pict>  ::=  <pict var>  |  <pict var> ( <attri3> )
              |  <attri pict> * <transform>

<attri3>  ::=  <attri1>  |  <attri1> ; <page coord>

<page coord>  ::=  <lower limit> , <upper limit>

<lower limit>  ::=  <x> , <y>

<upper limit>  ::=  <x> , <y>
```

4. Decoding Interrupts

    <decode statement> ::= DECODE (<intno> , <dev> ,

                  <inform> , <pictname> , <x> , <y> , <tf>)

    <intno> ::= <integer> | <integer variable>

    <dev> ::= LPEN | FKEY | KEYBRD | VECTOR | ....

    <inform> ::= <variable>

    <tf> ::= <variable>

    N.B. For LPEN, <pictname> is replaced by <id vector>.

    <id vector> ::= <variable>

5. Retrieval of Information

    <retrieve statement> ::= RETRIEVE (<pict var> ,

                  <informa> , <var>)

    <var> ::= <variable>

    <informa> ::= <alpha> | <informa> <alphanumeric>

6. Function Declaration Statement

    <fct decl statement> ::= FUNCTION <fct name> (<para>)

APPENDIX II   <u>Position</u> <u>Attributes</u> <u>of</u> <u>Picture</u> <u>Components</u>.

The position attributes consist of

    <x> , <y> , <pos>

where

      <x> and <y> specify the X and Y coordinates,

      <pos> specifies whether the coordinates are taken as relative or absolute.

If <pos> = 0 (or blank) or 1, then <x> and <y> are taken as the absolute coordinates with respect to the origin of the picture specified on the left-hand side of the picture-definition statement.

If <pos> = 2, then <x> and <y> are the relative coordinates with respect to the current position of the drawing device.

After plotting a picture, the beam is returned (blanked) to the picture origin. After drawing a primitive, the beam is left at the current position.

The actions taken by the drawing device for different picture components are as follows:

(1) Picture -- the specified picture is drawn at location (<x>,<y>).

(2) Drawing primitives:

    (a) Point -- a point is drawn at location (<x>,<y>).

    (b) Vector -- a vector is drawn from the current position of the drawing device to location (<x>,<y>).

(c) Symbol -- the symbol specified is drawn at location

(<x>,<y>).

(d) Line -- a series of vectors is drawn through the
points specified. For Line type 1, the coordinates of
the points are taken from <vx> and <vy> (see Appendix
III). For Line type 2, the coordinates of the points
are taken sequentially from the string specified.
<pos1> specifies the positioning of the first vector.
<pos2> specifies the positioning of the remaining
vectors.

(e) Text -- a specified sequence of symbols is drawn
starting at location (<x>,<y>). For Text type 1, the
symbols are taken from <sx> (see Appendix III). For
Text type 2, the symbols are taken from the specified
string.

APPENDIX III   List of Attributes as Implemented

1. Display attributes for picture

<lpdet> , <blink> , <blank>

2. P primitive

<x> , <y> , <pos> , <lpdet> , <blink> , <blank>

3. V primitive

<x> , <y> , <pos> , <lpdet> , <blink> , <blank> , <dash>

4. S primitive

<x> , <y> , ' <s> ' , <pos> , <lpdet> , <blink> ,

<blank> , <size> , <orient>

5. L primitive

Type 1

<vx> , <vy> , <num> , <pos1> , <pos2> , <lpdet> ,

<blink> , <blank> , <dash>

Type 2

<num> , ' <x1> , <y1> ; <x2> , <y2> ;...; <xn> , <yn> ' ,

<pos1> , <pos2> , <lpdet> , <blink> , <blank> , <dash>

6. T primitive

Type 1

<x> , <y> , <num> , <sx> , <pos> , <lpdet> , <blink> ,

<blank> , <size> , <orient>

Type 2

<x> , <y> , <num> , ' <text> ' , <pos> , <lpdet> ,

<blink> , <blank> , <size> , <orient>

Note :

(a) <x> , <y> are X and Y coordinates.

(b) <pos> , <pos1> indicate absolute or relative positioning

(c) <pos2> indicates positioning of the vectors with respect
to the first vector of L primitive.

(d) <vx> , <vy> are vectors of X and Y coordinates.

(e) <xi>, <yi> are the actual coordinate values.

(f) <num> is the number of vectors or symbols.

(g) <sx> is the location where the symbols are taken.
Symbols are assumed to be packed four to a 360 word

(h) <text> is the actual text to be plotted.

(i) <s> is the actual symbol plotted.

(j) All other attributes are in <z> format (see Appendix I)
Detailed descriptions of the attribute values are given
in Appendix IV.

APPENDIX IV   The Attribute Values

1.   \<pos\> , \<pos1\> , \<pos2\>

    1 => absolute positioning   (w.r.t. picture origin)

    2 => relative positioning   (w.r.t. last beam position)

2.   \<lpdet\>

    1 => light pen detactable

    2 => not light pen detactable

3.   \<blink\>

    1 => not blinking

    2 => blinking

4.   \<blank\>

    1 => not blanked

    2 => blanked

5.   \<dash\>

    1 => solid plotting of vector

    2 => dashed plotting of vector

6.   \<size\>

    1 => normal size symbols

    2 => large symbols

7.   \<orient\>

    1 => normal orientation for symbols

    2 => symbols plotted at 90 degress

APPENDIX V   A Sample Program

```
C
C   THIS PROGRAM ALLOWS THE USER TO DISPLAY SQUARES AND
C   TRIANGLES ON THE SCREEN.    THREE COMMAND WORDS, 'SQUARE',
C   'TRAANGLE' AND 'DELETE', ARE DISPLAYED INITIALLY ON THE
C   SCREEN.
C   TO DISPLAY A SQUARE(OR TRIANGLE), FIRST PICK 'SQUARE'(OR
C   'TRIANGLE') USING THE LIGHT PEN, THEN PICK A POINT ON THE
C   SCREEN AND FOLLOW BY HITTING THE SEND KEY.   TO DELETE ALL
C   SQUARES(OR TRIANGLES), FIRST HIT 'DELETE', THEN, HIT
C   'SQUARE' (OR 'TRIANGLE') USING THE LIGHT PEN, FOLLOWED BY
C   HITTING THE SEND KEY.   TO END THE PROGRAM, PRESS ANY
C   STATUS AND FUNCTION KEY, FOLLOWED BY THE SEND KEY.
C
        IMPLICIT INTEGER(A-Z)
        COMMON ST(10),M,N,X,Y
C
C   INITIALIZE THE VARIABLES  USED IN THE GRAPHICS STATEMENTS
C
#       COMMON ST(10),M,N,X,Y
C
C   DEFINE THE PICTURES PICT, SQ, TRIAN, COMMAND AND ERROR
C   INITIALIZE THE PICTURES SQS, TRS, DUMMY AND D
C
#       PICT = COMMAND + SQS + TRS + D
#       SQ = L(4,'0,200; 200,200; 200,0; 0,0')
#       TRIAN = L(3,'200,0; 100,200; 0,0')
#       COMMAND = T(5,850,6,'DELETE') + T(5,900,8,'TRIANGLE')
#       + T(5,950,6,'SQUARE')
#       SQS = NULL
#       TRS = NULL
#       DUMMY = NULL
#       D = NULL
#       ERROR = T(500,900,16,'ERROR--TRY AGAIN')
   10   CONTINUE
C
C   DISPLAY PICT ON SCREEN
C
#       DRAW(1,PICT)
#       D = NULL
C
C   CHECK IF THE FIRST INTERRUPT IS FROM FKEY
C
#       DECODE(1, FKEY,M,DUMMY,X,Y,N)
        IF (N.EQ.0) GO TO 100
C
C   CHECK IF FIRST INTERRUPT IS A LIGHT PEN HIT
C
#       DECODE(1,LPEN,M,ST,X,Y,N)
```

```
         IF (N.EQ.0) GO TO 30
   15  CONTINUE
C
C   SET ERROR MESSAGE TO BE DISPLAYED
C
#         D = ERROR(0,0)
         GO TO 10
   30  IF (ST(3).NE.1) GO TO 15
         IF (ST(4).NE.1) GO TO 50
C
C  FIRST INTERRUPT IS A LIGHT PEN HIT ON 'DELETE'
C  CHECK IF SECOND INTERRUPT IS FROM LIGHT PEN
C
#         DECODE(2,LPEN,M,ST,X,Y,N)
         IF(N.NE.0) GO TO 15
         IF(ST(3).NE.1) GO TO 15
         IF (ST(4).EQ.3) GO TO 45
C
C  2ND INTERRUPT IS LIGHT PEN HIT ON 'TRAINGLE'
C  DELETE ALL TRAINGLE DISPLAYED ON THE SCREEN
C
#         TRS = NULL
         GO TO 10
   45  CONTINUE
C
C  2ND INTERRUPT IS LIGHT PEN HIT ON 'SQUARE'
C  DELETE ALL SQUARES DISPLAYED ON THE SCREEN
C
#         SQS = NULL
         GO TO 10
   50  CONTINUE
C
C  CHECK IF 2ND INTERRUPT IS POINT PICK
C
#         DECODE(2,POINT,M,DUMMY,X,Y,N)
         IF(N.NE.0) GO TO 15
         IF(ST(4).EQ.3) GO TO 60
C
C  ADD A TRAINGLE ON THE SCREEN
C
#         TRS = TRS + TRIAN(X,Y)
         GO TO 10
   60  CONTINUE
C
C  ADD A SQUARE ON THE SCREEN
C
#         SQS = SQS + SQ(X,Y)
         GO TO 10
  100  CONTINUE
C
C  STATUS AND FUNCTION KEYS PRESSED
C  SEND FAREWELL MESSAGE
```

```
C
#       PICT = T(500,500,8,'FAREWELL')
#       DRAW(1,PICT)
C
C    ROUTINE EXIT1 RELOADS GRID BOOTSTRAP
C
        CALL EXIT1
        STOP
        END
```

APPENDIX VI   A Logic Circuit Application Program


```
C
C  THIS PROGRAM ALLOWS ONE TO BUILD LOGIC CIRCUITS ON THE SCREEN,
C    MODIFY THE CIRCUIT, AND ANALYSE THE CIRCUIT
C
       IMPLICIT INTEGER (B-Z)
       COMMON/BLK/IN(100,3)
       COMMON ST(11),M,N,X,Y,SEL,VAL,SELM,A,B,NUM,RX(100),RY(100)
C
C   INITIALIZE THE PICTURES TO BE DISPLAYED
C
#      COMMON ST(11),M,N,X,Y,SEL,VAL,SELM,A,B,NUM,RX(100),RY(100)
#      MENU = T(5,950,7,'ANDGATE') + T(5,900,6,'ORGATE')
#         + T(5,850,7,'NOTGATE') + T(5,800,6,'DELETE')
#         + T(5,750,6,'ATTACH') + T(5,700,5,'INPUT')
#         + T(5,650,6,'OUTPUT') + T(5,600,7,'RESTART')
#         + T(5,550,5,'ERASE')
#      LEAD = V(20,0)
#      TRIAN = V(20,20) + V(0,40) + V(0,0)
#      FIGURE = TRIAN(0,0) + S(10,20,'+')
C
C      DEFINE A SEMI-CIRCLE
C
       RADIUS = 20
       ATHETA = 300/RADIUS
       NUM = (180/ATHETA) + 0.5
       ATHETA = 180/NUM
       DO 22 I=1,NUM
       ANGLE = ATHETA * I * 3.1416 / 180
       RX(I) = RADIUS * SIN (ANGLE+0.0)
       RY(I) = RADIUS * (1 - COS(ANGLE+0.0) )
    22 CONTINUE
#      SEMI = L(RX,RY,NUM) + V(0,0)
#      NOT = LEAD(0,0) + LEAD(0,10,0,0,0,2)
#         + TRIAN(20,-20) + LEAD(40,0)
#      OR = LEAD(0,10) + LEAD(0,-10) + FIGURE(20,-20) + LEAD(40,0)
#      AND = LEAD(0,10) + LEAD(0,-10) + SEMI(20,-20) + LEAD(40,0)
#      PICT = MENU + GATES + LEADS + D + INPUT + OUTPUT
    1  CONTINUE
#      INPUT = NULL
#      OUTPUT = NULL
#      DUMMY = NULL
#      D = NULL
#      GATES = NULL
#      LEADS = NULL
C
C   INITIALIZE TABLE FOR ANALYSIS
C
C   IN(I,1) INDICATES THE FIRST INPUT OF I-TH GATE
```

```
C     IN(I,2) INDICATES THE SECOND INPUT OF I-TH GATE
C            (NOT APPLICABLE TO NOT GATE)
C     IN(I,3) INDICATES THE TYPE OF THE GATE
C            1 => AND    2 => OR    3 => NOT
C        INPUT    0 => -1     1 => -2
C
      NUMBER = 0
      DO 5 I=1,100
      DO 5 J=1,3
    5 IN(I,J) = -1000
   10 CONTINUE
#     DRAW (1,PICT)
#     D = NULL
#     DECODE(1,FKEY,M,DUMMY,X,Y,N)
      IF(N.EQ.0) GO TO 90
#     DECODE(1,LPEN,M,ST,X,Y,N)
      IF (N.EQ.0) GO TO 30
   15 CONTINUE
#     D = T(500,900,16,'ERROR--TRY AGAIN')
      GO TO 10
   30 IF (ST(3).NE.1) GO TO 15
      DUM = ST(4)
      GO TO (100,100,100,200,300,400,500,600,700),DUM
      GO TO 15
C
C     IF FIRST LIGHT PEN HIT IS AND , OR ,NOT GATE
C     AND SECOND INTERRUPT IS POINT PICK
C     THE GATE IS DISPLAYED AT THE SELECTED POSITION
C
  100 CONTINUE
#     DECODE(2,POINT,M,DUMMY,X,Y,N)
      IF(N.NE.0) GO TO 15
      NUMBER = NUMBER + 1
      GO TO (110,120,130),DUM
  110 CONTINUE
      IN(NUMBER,3) = 1
#     GATES = GATES + AND(X,Y)
      GO TO 140
  120 CONTINUE
      IN(NUMBER,3) = 2
#     GATES = GATES + OR(X,Y)
      GO TO 140
  130 CONTINUE
      IN(NUMBER,3) = 3
#     GATES = GATES + NOT(X,Y)
  140 CONTINUE
#     OUTPUT = NULL
      GO TO 10
C
C     PROCESS DELETE COMMAND
C
  200 CONTINUE
```

```
 #         DECODE(2,LPEN,M,ST,X,Y,N)
           IF(N.NE.0) GO TO 15
           IF( (ST(3).NE.2) .AND. (ST(3).NE.3) ) GO TO 15
           IF(ST(3).EQ.3) GO TO 250
 C
 C         DELETE THE GATE
 C
           SEL = ST(4)
           II = SEL + 1
           DO 210 I = II, NUMBER
           IN(I-1,1) = IN(I,1)
           IN(I-1,2) = IN(I,2)
           IN(I-1,3) = IN(I,3)
  210      CONTINUE
           IN(NUMBER,1) = -1000
           IN(NUMBER,2) = -1000
           IN(NUMBER,3) = -1000
           NUMBER = NUMBER - 1
           DO 220 I=1,NUMBER
           DO 220 J=1,2
           IF(IN(I,J).EQ.SEL) IN(I,J)=-1000
  220      CONTINUE
 #         GATES.SEL = NULL
           GO TO 140
  250      SEL = ST(4)
           SELM = ST(4) - 1
 #         LEADS.SEL = NULL
 #         LEADS.SELM = NULL
 °         GO TO 140
 C
 C         PROCESS ATTACH COMMAND
 C
  300      CONTINUE
 #         DECODE(2,LPEN,M,ST,X,Y,N)
           IF(N.NE.0) GO TO 15
           IF (ST(3).NE.2) GO TO 15
           IF(ST(5).GT.2) GO TO 15
           INGATE = ST(4)
           INLEAD = ST(5)
           SEL = ST(4)
 #         RETRIEVE(GATES.SEL,X,A)
 #          RETRIEVE(GATES.SEL,Y,B)
           IF(IN(INGATE,3).EQ.3) GOTO 301
           IF(INLEAD.EQ.1) B = B+10
           IF(INLEAD.EQ.2) B = B - 10
  301      CONTINUE
 #         DECODE(3,LPEN,M,ST,X,Y,N)
           IF(N.NE.0) GOTO 15
           IF(ST(3).NE.2) GOTO 15
          ,IF(ST(5).NE.4) GO TO 15
           OUGATE = ST(4)
           OULEAD = ST(5)
```

```
          IF(IN(INGATE,INLEAD).NE.-1000) GO TO 15
          IN(INGATE,INLEAD) = OUGATE
          SEL = ST(4)
#         RETRIEVE(GATES.SEL,X,X)
#         RETRIEVE(GATES.SEL,Y,Y)
          X = X +60
#         LEADS = LEADS + V(A,B,,,,2) + V(X,Y)
          GO TO 140
C
C      PROCESS INPUT COMMAND
C
  400 CONTINUE
#         DECODE(2,LPEN,M,ST,X,Y,N)
          IF(N.NE.0) GO TO 15
          IF(ST(3).NE.2) GO TO 15
          IF(ST(5).GT.2) GO TO 15
          INGATE = ST(4)
          INLEAD = ST(5)
          IF(IN(INGATE,INLEAD).NE.-1000) GO TO 15
          SEL = ST(4)
#         RETRIEVE(GATES.SEL,X,X)
#.          RETRIEVE(GATES.SEL,Y,Y)
          IF(IN(INGATE,3).NE.3) GO TO 420
          Y = Y + 4
          GO TO 430
  420 IF(INLEAD.EQ.1) Y = Y +14
          IF(INLEAD.EQ.2) Y = Y - 6
  430 CONTINUE
#         DECODE(3,KEYBRD,M,DUMMY,A,B,N)
          IF(N.NE.0) GO TO 15
          VAL = INTCVT(M,1,E)
          IF((VAL.NE.0).AND.(VAL.NE.1)) GO TO 15
          IF (VAL.EQ.1) GO TO 450
          IN(INGATE,INLEAD) = -1
#         INPUT = INPUT + S(X,Y,'0')
          GO TO 140
  450 IN(INGATE,INLEAD) = -2
#         INPUT = INPUT + S(X,Y,'1')
          GO TO 140
C
C      PROCESS OUTPUT
C
  500 CONTINUE
#         DECODE(2,LPEN,M,ST,X,Y,N)
          IF(N.NE.0) GO TO 15
          IF(ST(3).NE.2) GO TO 15
          IF(ST(5).NE.4) GO TO 15
          OUGATE = ST(4)
          OULEAD = ST(5)
          SEL = ST(4)
          CALL ANALYS(OUGATE,VALUE)
#         RETRIEVE(GATES.SEL,X,X)
```

```
#       RETRIEVE(GATES.SEL,Y,Y)
        Y = Y + 10
        X = X + 60
        IF(VALUE.EQ.-2) GO TO 550
#       OUTPUT = OUTPUT + S(X,Y,'0')
        GO TO 10
  550 CONTINUE
#       OUTPUT = OUTPUT + S(X,Y,'1')
        GO TO 10
C
C     PROCESS RESTART COMMAND
C
  600 CONTINUE
#       DECODE(2,END,M,DUMMY,X,Y,N)
        IF(N.NE.0) GO TO 15
        GO TO 1
C
C     PROCESS ERASE INPUT COMMAND
C
  700 CONTINUE
#       DECODE(2,LPEN,M,ST,X,Y,N)
        IF(N.NE.0) GO TO 15
        IF(ST(3).NE.2) GO TO 15
        INGATE = ST(4)
        LD = ST(5)
        IF(LD.GT.2) GO TO 15
#       DECODE(3,LPEN,M,ST,X,Y,N)
        IF(N.NE.0) GO TO 15
        IF(ST(3).NE.5) GO T O 15
        SEL = ST(5)
        IN(INGATE,LD) = -1000
#       INPUT.SEL = NULL
        GO TO 140
   90 CONTINUE
#       PICT = T(500,500,8,'FAREWELL')
#       DRAW(1,PICT)
        CALL EXIT1
        STOP
        END
        SUBROUTINE ANALYS(INPUT,NOUT)
        IMPLICIT INTEGER(A-Z)
        COMMON/BLK/IN(100,3)
        DIMENSION STACK(100)
C
C     THIS SUBROUTINE DOES THE LOGIC ANALYSIS
C
        PT = 1
        STACK(PT) = INPUT
   11 IF((STACK(PT).LT.0).AND.(PT.EQ.1)) GO TO 1999
        IF(STACK(PT).GT.0) GO TO 199
        IF(STACK(PT-1).LT.0) GO TO 299
        GATE = STACK(PT-1)
```

```
      IF(IN(GATE,3).NE.3) GO TO 100
      IF(PT.GE.3) GO TO 20
 15   VAL = STACK(PT)
      PT = PT - 1
      IF(VAL.EQ.-1) STACK(PT) = -2
      IF(VAL.EQ.-2) STACK(PT) = -1
 20   IF(PT.LT.3)GO TO 11
      GATE = STACK(PT-1)
      GATE1 = STACK(PT-2)
      IF(GATE1.LT.0) GO TO 15
      IF ( (IN(GATE,3).EQ.3) .AND. (IN(GATE1,3).NE.3) ) GO TO 100
      GO TO 11
100   STACK(PT-1) = STACK(PT)
      STACK(PT) = GATE
      GO TO 11
299   IF(STACK(PT-2).GT.0) GO TO 310
300   WRITE(6,301)
301   FORMAT(1X,'ERROR')
      RETURN
310   GATE = STACK(PT-2)
      IF(IN(GATE,3).EQ.1) GOTO 350
      IF((STACK(PT).EQ.-2).OR.(STACK(PT-1).EQ.-2)) GOTO 320
305   STACK(PT-2) = -1
      GO TO 330
320   STACK(PT-2) = -2
330   PT = PT - 2
      GO TO 11
350   IF((STACK(PT).EQ.-2).AND.(STACK(PT-1).EQ.-2)) GO TO 320
      GOTO 305
199   GATE =STACK(PT)
      IF(IN(GATE,3).NE.3) GO TO 200
      PT = PT +1
      STACK(PT) = IN(GATE,1)
      GO TO 11
200   STACK(PT+1) = IN(GATE,1)
      STACK(PT+2) = IN(GATE,2)
      PT = PT + 2
      GO TO 11
1999  NOUT = STACK(PT)
      RETURN
      END
```