IMPLEMENTATION OF AN INTERACTIVE GRAPHICS LANGUAGE

Bert Pieké and Günther Schrack
Department of Electrical Engineering
University of British Columbia
Vancouver 8, B.C.

ABSTRACT

The design and implementation of the interactive graphics language IGL is described. This language not only allows the definition and display of line drawings but also has full facilities for manipulating, naming, identifying and interacting with such drawings. The language has been implemented as an extension to Fortran IV using the XPL compiler generator system. The experience gained so far in the use of the language has already proven a number of advantages over present-day graphics systems. The language is readily learned by users with previous high-level language experience. As no extensive testing and documentation is necessary due to the readability of the program, the time required for the completion of a project is greatly reduced.

RESUMÉ

On décrit le développement et la réalisation du langage interactif de représentation graphique IGL. Ce langage permet non seulement la définition et la représentation de dessins à l'aide de lignes mais aussi la manipulation, la nomination, l'identification et la modification des dessins. Ce langage est une extension du Fortran IV utilisant le système du générateur-compilateur XPL. L'expérience que l'on a acquise jusqu'à présent dans l'utilisation du langage a déjà démontré quelques avantages sur des systèmes de la représentation graphique d'aujourd'hui. L'utilisateur qui a déjà de l'expérience dans les langages élevés peut facilement l'acquerir. La programme ne nécessitant pas d'essais étendus ou de documentation (du fait qu'il est très clair), le temps à consacrer pour faire un travail est très réduit.

IMPLEMENTATION OF AN INTERACTIVE GRAPHICS LANGUAGE

Bert Pieké and Günther Schrack
Department of Electrical Engineering
University of British Columbia
Vancouver 8, B.C.

Introduction

Over the last years, an increasing need has been felt in the Department of the authors for an interactive graphics facility to permit computer-aided design. After an initial attempt by writing a program in assembler language to handle drawings of electronic circuits on the graphics console of a PDP-9 computer, it became obvious that the design of graphics systems should occur on a more general level and take into consideration the following specifications:

- It should be possible to write graphics programs in a high-level language thus speeding up the design, the implementation, and the testing and to allow writing programs for a wide variety of applications.
- The system programs themselves must be well documentable and easy to change, particularly in a university environment where a regular turnover of students using and updating an existing system occurs.
- The convenience of existing algorithmic high-level languages should be available.

Programs for interactive computer graphics have traditionally been expensive ventures tieing up large computers to service the display and requiring a great amount of systems knowledge of the applications programmer, partly due to the fact that the languages available for the implementation were originally designed for other purposes. Practical systems were written in assembler and experimental systems in algorithmic languages. A number of special purpose languages have recently appeared in the literature, but none seem to have found wide acceptance.

This paper describes the design and implementation of such a language and the experience gained with it.

The Language IGL

The acronym IGL stands for Interactive Graphics Language, a computer language for handling line drawings. This language not only allows the definition and display of line drawings but also has full facilities for manipulating, naming, identifying and interacting with such drawings. Through the use of windowing techniques, parts of the drawing can be magnified for closer inspection or scaled down for greater display density. Commands are further provided for saving and restoring drawings on secondary storage.

The IGL Compiler

Creating a complete, versatile computer language may seem to be an even greater task than to implement a graphics system without the help of such a language. This does not have to be so, however. A number of algorithmic languages exist that include most features needed. Such features are, e.g., branching, conditional and arithmetical statements and a complete I/O handling facility. Languages as for example Fortran and Algol that are widely known and used lend themselves excellently to act as host languages for special purpose languages. The special purpose language can then be written as an extension of the host language and need only include those constants, operators, functions and commands that are not already available.

These ideas are incorporated in the language IGL. Thus an applications program written in IGL appears to be a mixture of statements for graphical manipulations and host-language statements. The distinction between the two is made on a card-by-card basis, the graphical statements having a special character (*) in the first column of each card. The compilation is executed in two stages: first a compilation from IGL into host-language, secondly a compilation of the host-language into machine language.

Because of its widespread use and support, Fortran IV was chosen both as host language for IGL and as programming language for the semantic routines. This one-language approach provides almost complete portability from one computer installation to another.

The IGL compiler was written using the XPL compiler generator system [5]. This is a well documented and easy to use program package written in the XPL dialect of PL/I. To build a compiler using the XPL system, the language syntax in Backus Naur notation (BNF) and the corresponding semantics in XPL must be supplied. These two user components are processed as follows. The syntax is read in, printed, analyzed and a parser is punched out by the XPL program ANALYZER. ANALYZER will check that the grammar is unambiguous and will attempt to modify the grammar if that is not the case. The produced parser is in a format that allows it to be directly inserted into a compiler framework called SKELETON. SKELETON itself is written in XPL and has two open slots, one for accepting the parser from ANALYZER and one for accepting the semantics. When these two components have been added, SKELETON can be compiled by the XPL compiler producing for the input grammar a compiler as an object program.

The semantics are written as an XPL procedure named SYNTHESIZE. This procedure is called in SKELETON each time a grammar rule is applied by the parser for reducing the input string. The number of the particular rule applied is passed along as an argument. In SYNTHESIZE the output language is generated. If the output is in a high level language, the output statements will usually consist of calls to semantic routines. If the output is in assembly language, the actions can be generated directly. For passing names from the input to the output language, SYNTHESIZE has access to the contents of the parsing stacks. Pointers are kept by SKELETON to show the current state of these stacks.

This automation of the compiler-writing process allows changes to the language to be carried out easily when needed. The syntax and the corresponding semantics can be updated independently of one another. Compilers generated by the XPL system provide good error checking facilities and have proven to be quite efficient.

Syntax

The graphical aspects of the syntax of IGL are based on the work of F. Nake [6]. In addition to existing types of constants and variables in the host language (e.g. integer, real, etc.), the new type IMAGE is introduced. Image constants are either elements from the set {BLANK, DOT, LINE, SQUARE, TRIANGLE, CIRCLE, HALFCIRCLE} or strings of literals (keyboard characters). Image variables are defined with the aid of the image assignment operation (:=) by image expressions which are strings of image constants and/or variables joined by diadic image operators. Image constants and variables are initially defined on the unit square. To each image variable a set of attributes is attached defining coordinates, scale, and angle of rotation of the picture which the variable represents. These attributes can be redefined with the use of unary image operators, hence affine transformations such as translation, scaling, rotation and mirroring can be applied.

For illustration, the following image assignment statements define an image variable to represent the symbol "resistor".

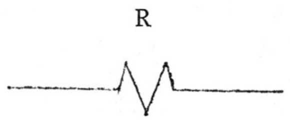
```
  *     HR:= LINE FROM 0,.5 TO .4,.5 + LINE FROM .4,.5 TO .425,1
  *     + LINE FROM .425,1 to .5,0;
  *     R:= HR + HR VSYM .5;
```

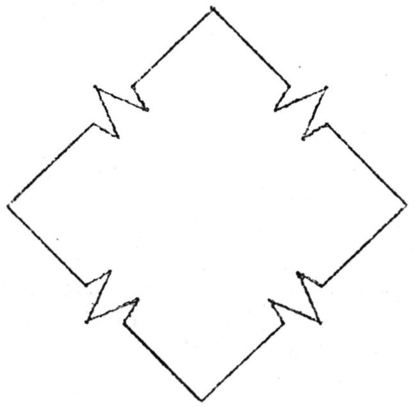
The right hand sides of both statements employ the diadic image operator +(superposition). The first statement defines a temporary image variable HR by superimposing the image constant LINE three times, each time modified with the image operator FROM ... TO ... in the desired manner. The second statement defines the variable R as a superposition of HR and the reflection of HR on the vertical at x = .5, thus completing the symbol "resistor".

The numeric operands of image operators may be not only numeric constants but can be numeric variables of the host language as well. Therefore, complicated pictures can be defined with image assignment statements imbedded in statements of the host language, e.g. in loops. For example, the following program portion:

```
      S = .25*SQRT(2.)
      SC = 2.*S
      ALPHA = PI/4.
*     BRIDGE:=BLANK;
      DO 100 I = 1, 4
      X1 = .5 + S * SIN(ALPHA)
      Y1 = .5 - S * COS(ALPHA)
*     BRIDGE:= BRIDGE + R AT X1,Y1 SCALE SC,.1  ANGLE ALPHA;
100   ALPHA = ALPHA + PI/2.
*     DISPLAY BRIDGE;
```

would display a resistor bridge:





The facilities described above allow the creation and naming of items called image variables. For most applications, however, an additional facility is needed to group into logical groupings, items that do not necessarily look alike. As an example consider an electronic circuit diagram. At first glance it may seem to be composed of a limited set of identical symbols. Apart from being in different places in a circuit, two resistors, e.g., may seem to be duplicates of each other. A closer examination, however, shows that the leads of the first have different lengths than the leads of the second and that the labels differ from one another. Such details as the length of the leads and the label are properties of the pictures representing the individual resistors and must be taken into account. A second category of variables, subscripted image variables, is therefore defined in the syntax. Subscripted variables names are names of items that are not affine transformations of one another, but that are logically members of the same set. All network elements in a circuit can thus be named, e.g., ELEMENT(I) regardless of whether they are resistors, capacitors etc. Alternatively, if this distinction is important, all resistors in a circuit can be named RESISTOR(I) regardless of the length of their leads or the letters in their label.

Example:

```
*     RESISTOR(3): = R AT X,Y SCALE S1 + LEAD FROM X1, Y1 TO X2, Y2 + LEAD
           FROM X3, Y3 TO X4, Y4 + 'R' AT X5, Y5 + VALUE(3) AT X6, Y6;
*     CIRCUIT:= CIRCUIT + RESISTOR(3);
```

Interaction

Describing the interactive process means defining the response of the system to each input. The response is not only dependent on the type of input, but also on the state of the system at the time the input occurred. W.H. Newman [7] has proposed to treat the system as a finite-state automaton where the response is determined by the state of the program as well as by the action. The actions are inputs to the automaton, which cause it to change its state; reactions are the outputs.

The interaction is then best described in the from of a state diagram, which is used as a guide when writing the program. To facilitate this process, the syntax of IGL allows a close correspondence between the state diagram notation and the formulation in the program. This aspect of the IGL-system is based on the work of P. Boullier et al. [1]. The program is divided into states which correspond to the states in the diagram. For each input (e.g. a hit on a menu symbol on the screen) a decision is made within the current state of the program as to which state is to be executed next. For an illustration, see Figure 1.

The division into states also provides a physical segmentation of the program which can be used for paging or overlaying at computer installations with insufficient core storage for the entire program. Control statements are defined in the language to allow variables to be displayed on the screen, to turn on the cursor or cross-hairs (which can be positioned by the user with the help of some graphical input device such as a joy-stick, tracker ball or lightpen) and to detect an interrupt from the user, signalling that he has chosen an item on the screen. A special variable: IHIT is used in the program to indicate the subscript of the item last identified on the screen. This subscript allows the programmer to refer to items that are pointed at by the user when he is interacting with the program. For example the sequence:

```
*     DISPLAY 'IDENTIFY RESISTOR TO BE DELETED' AT .2,.9;
*     CURSOR ON; WAIT FOR INTERRUPT;
*     FOR HIT ON RESISTOR: CIRCUIT:=CIRCUIT-RESISTOR(IHIT)
```

would prompt the user to select the resistor to be deleted and would execute the deletion.

Semantics and Data Structure

The semantic routines are the routines that execute at run-time and perform the actions specified by the statements written in the program. The control and interaction routines affect the sequence of the execution and the communication with the graphics terminal, whereas the assignment routines operate on a data structure that represents the current state of the pictorial information. The display is derived algorithmically from this structure and no separate display file is needed. The correspondance between the variable names used in the program and the locations in the data structure is kept through a hash-coded directory where all names are stored together with pointers to the structure. (For a description of hash-coding see [4]). The data structure is implemented as a linked list [3] with "brother"-pointers linking items that are connected by the (+) operator and "son"-pointers linking downwards through the structure to sub items and primitives that are used in the definition of items. No "father"-pointers are kept linking upwards through the structure, so it would not be possible to say e.g. in which items the primitive LINE is referenced. This means that the cursor identification on the screen does not follow the usual pattern of determining which line is closest to the location of the cursor and then determining to which item this line belongs. Instead, the syntax of the identification statement: FOR HIT ON <variablename>: <statementlist> END; allows a scan of all items with the name <variablename>, which are linked by additional "buddy"-pointers, to determine whether any of those items were within a tolerance region (specified by the x and y-scale of the item) around the cursor location.

Using IGL

The flexibility and ease of use make the language IGL ideal for experimenting with new graphics techniques and ideas. To gain experience with the system, several applications were programmed by different users. First projects were a program for drawing electronic circuits and a load flow program for the analysis of power systems [2]. In two 12 week systems lab projects, groups of fourth year students in this department implemented complete interactive systems; one for interactive project scheduling using CPM methods and one for interactive nonlinear circuit analysis, [8]. Both systems run on a minicomputer in this department and interface to analysis programs on the IBM 360 installation in the Computer Centre. A voice grade data link is used for transmitting data from one computer to another. The experience gained so far in the use of the language has already proven a number of advantages over present-day graphics systems.

Due to the definition of the syntax and the semantics, changes of these are easily incorporated as experience is gained in the use of the language. The system has been found far easier to use than the graphical subroutine package found on most computer installations. The language is readily learned by users with previous high-level language experience. As no extensive testing and documentation is necessary due to the readability of the program, the time required for the completion of a project is greatly reduced. Furthermore, the availability of a new powerful tool stimulates the imagination of the user to tackle problems previously considered out of reach.

System configuration

Two implementations are used at this university. One runs under the MTS timesharing system on an IBM 360/67 with Calcomp and line-printer plotters. This system is used mainly for debugging new programs. The interactive system runs under DOS on a Data General 20k Supernova with a Tektronix 4010 graphics terminal. Graphics programs can be run on either system without modification.

Figure 1

```
*    STATE 1:
C
C --- DEFINE ELEMENTS AND MENUSYMBOLS
C
     ...
     ...
C --- CAPACITOR
*    HALFC=LEAD + LINE FROM .4,0 TO .4,1;
*    C=HALFC + HALFC VSYM .5 + MARKER;
     ...
     ...
C --- MENUCOMMANDS
*    DELETE='DELETE'; SAVE='SAVE'; ROTATE='ROTATE';
     ...
     ...
*    GOTO STATE 2;
*    END STATE 1;


*    STATE 3:
C
C --- STATE 3 : PICKING FROM THE MENU
C
*    DISPLAY 'CHOOSE FROM MENU' AT .1,.9;
*    CURSOR ON;
*    WAIT FOR INTERRUPT;
*    FOR HIT ON MENU:
*    FOR HIT ON DELETE : GOTO STATE 4; END;
*    FOR HIT ON ROTATE: GOTO STATE 5; END;
*    FOR HIT ON SAVE: GOTO STATE 6; END;
     ...
     ...
*    END;
     ...
     ...
*    END STATE 3;


*    STATE 5:
C
C --- STATE 5 : ROTATING AN ELEMENT
C
*    DISPLAY 'PICK ELEMENT TO BE REVERSED' AT .1,.2;
*    CURSOR ON; WAIT FOR INTERRUPT;
*    FOR HIT ON MENU:  GOTO STATE 3; END;
*    FOR HIT ON ELEMENT:
*    AN = ANGLE(ELEMENT(IHIT));
       PI=3.141593
       AN=AN+PI
*    ANGLE(ELEMENT(IHIT)) = AN;
*    ERASE SCREEN; DISPLAY CIRCUIT;
*    END; GOTO STATE 3;
*    END STATE 5;
```

References

[1]. P. Boullier et al., "METAVISU, A general purpose graphic system", in F. Nake, A. Rosenfeld, Eds., Graphic Languages. Amsterdam: North-Holland Publ. Co., 1972.

[2]. B.A. Dixon, "Interactive Graphical Load Flow", submitted to: International Electrical, Electronics Conference and Exposition, Toronto, Oct. 1973.

[3]. D.E. Knuth, The Art of Computer Programming, Vol. 1, Addison-Wesley, 1969.

[4]. W.D. Maurer, "An Improved Hash Code for Scatter Storage", Comm. ACM 11 (Jan. 1968).

[5]. W.M. McKeeman et al., A Compiler Generator. Prentice-Hall, 1970.

[6]. F. Nake, "A proposed language for the definition of arbitrary two-dimensional signs", in O.I. Grusser, R. Klinke Eds., Zeichenerkennung in biologischen and technischen Systemen. Berlin: Springer, 1971.

[7]. W.M. Newman, "A system for interactive graphical programming", AFIPS Conf. Proc. SJCC 32, 1968.

[8]. B. Pieké and G.F. Schrack, "Interactive Circuit Analysis Using a High-Level Graphics Language", 16th Midwest Symposium on Circuit Theory, Waterloo, 1973.