MC SYLLABUS 35

COLLOQUIUM COMPUTER GRAPHICS

P.J.W. TEN HAGEN (RED.)

MATHEMATISCH CENTRUM AMSTERDAM 1978

AMS(MOS) subject classification scheme (1970): 68.00

ACM-Computing Reviews-categories: 8.2

ISBN 90 6196 142 4

INHOUD

Inhoud. 1

Voorwoord VI

I. AN INTRODUCTION TO COMPUTER GRAPHICS by P. KLINT 1

Ø. Introduction. 1
1. Devices for computer graphics 1
1.1. Output devices. 2
1.1.1. Plotters. 2
1.1.2. Cathode ray tube (CRT) displays 2
1.1.3. Numerically controlled milling machines 3
1.1.4. Plasma panel. 3
1.2. Input devices 4
1.2.1. Tablet. 4
1.2.2. Light pen 4
1.2.3. Buttons 4
2. Graphics System organization. 4
3. A model for graphics i/o functions. 10
3.1. Motivation. 10
3.2. Model description 11
3.3. Choice of primitives. 14
3.3.1. Output. 15
3.3.2. Input 16
3.3.2.1. Input primitives. 16
3.3.2.2. Implementation of more complex input operations 17
3.3.2.3. Incorporation of input primitives in the DDT. 18
3.3.3. Picture manipulation. 19
4. Bibliography. 20

II. GPGS
GENERAL PURPOSE GRAPHIC SYSTEM by L.C. CARUTHERS 22

1.0. Introduction. 22
2.0. Design Considerations 23
2.1. What kind of subroutine package?. 23
2.2. Device independence 24

2.3. Real devices as seen by GPGS. 26
3.0. Graphic i/o 26
3.1. Picture making. 26
3.2. Picture segments and picture elements 27
3.3. Setting viewing conditions. 28
3.4. Picture element processing pipeline 29
3.5. Picture segment manipulations 29
3.6. Input from console tools. 30
3.7. Additional facilities 31
4.0. Conclusions 31
4.1. Acceptability of design 32
4.2. Access to devices 32
4.3. Suitability for applications. 33
References. 33
Appendix A: Sample program. 35
Appendix B: Console tool information returned 37
Appendix C: Table of device driver facilities 38

III. PHILDIG

PHILIPS DEVICE INDEPENDENT GRAPHICS door C. NIESSEN & J.W. ERO . 39

0. Inleiding 39
1. De noodzaak van PHILDIG 39
2. Globale opzet van PHILDIG 40
3. Administratieve routines. 44
4. Omlijsting routines 44
5. Generatie routines. 44
6. Structuur routines. 47
7. Transformatie routines. 50
8. Modificatie routines. 50
9. Invoer routines 52
10. Fouten afhandeling. 54
11. Implementaties. 54

IV. THE INTERMEDIATE LANGUAGE FOR PICTURES by P.J.W. TEN HAGEN . . . 57

1. Introduction. 57
2. The basic structure of the ILP. 59

3. The interpretation of attributes and pictures 62
4. Picture elements. 66
4.1. Coordinate type 66
4.2. Text. 68
4.3. Curve 70
4.4. Library 70
5. Basic attributes. 71
5.1. Control 71
5.2. Transformations 72
5.3. Penfunctions. 73
5.4. Style 73
5.5. Detection 75
6. Conclusion. 77
References. 78

V. SATELLITE GRAPHICS by J. VAN DEN BOS 79

0. Abstract. 79
1. Introduction. 80
2. Programming the satellite-lost system 81
3. CAGES 83
4. ICOPS 86
5. A multi microcomputers system for graphics. 88
6. Conclusion. 91
References. 91

VI. COMPUTER AIDED DESIGN OF MECHANICAL COMPONENTS door H. RANKERS . 95

0. Inleiding 95
1. De Fourrier-representatie van een periodieke functie. . 97
2. Het karakteriseren van mechanismen. 97
3. De type-synthese. 98
4. De dimensie-synthese. 98
5. Het overschot aan informatie. 98
6. Voorbeeld voor de synthese. 100
7. Uitbreiding van het systeem 100
8. Waarom benutten wij de beeldbuis? 105
9. Geen beslissingsalgorithme. 105
10. Demonstratie. 105

11. Interactieve werkwijze 105
12. Literatuurlijst 106

VII. INTERAKTIEF ONTWERPEN VAN MULTIVARIABELE REGELSYSTEMEN door A.J.J. VAN DER WEIDEN. P. VALK & O.H. BOSGRA 107

1. Inleiding 107
2. Inverse Nyquist array ontwerpmethode. 111
2.1. Grafische Interpretatie 114
2.2. Ontwerpmethodiek. 116
3. Programmatuur 117
4. Ontwerpvoorbeeld. 119
5. Konklusies. 123
Referenties 124

VIII. LOGICAL DESIGN AND DATA ANALYSIS WITH GPGS by L.C. CARUTHERS . . 125

0. Introduction. 125
1. STARFIR 127
2. PRISMA-PLOT 129
3. DIGDRA, Semi-automatic digital circuit construction . . 131
4. CONCLUSIONS 136
5. ACKNOWLEDGEMENTS. 137

IX. THE "TROTS SOFTWARE PACKAGE FOR THREE-DIMENSIONAL RECONSTRUCTION" by A.H. VEEN 138

1. Introduction. 138
2. The process of reconstruction 139
2.1. Preparing the visual data 140
2.2. From visual to computer data. 140
2.3. Transfer to the time sharing computer 141
2.4. Visual editing. 144
2.5. Three-dimensional reconstruction. 147
3. The hardware. 150
4. The TROTS software. 150
4.1. TRACER: Gathering the data. 151
4.2. PAPER: Transfer to the PDP-10 153
4.3. EDIT: Visual data manipulation. 153

4.4. STRIP: Converting to strip format 155
4.5. FIG 3D: Three-dimensional transformation and hidden line suppression. 158
4.6. UTILITIES: MERGE, FIXIT and FILL. 160
4.7. Experimental programs: CROSS and DRESS. 160
5. The supporting graphics package 162
6. Towards a real-time software hidden line remover. . . . 162
7. Evaluation. 166
References. 167

X. SHAPE PROCESSING FOR MECHANICAL COMPONENTS by I.C. BRAID 168

1. A mechanical component observed 168
2. Modelling the component's surface shape 170
3. Building up a shape model 171
4. A new shape design system 172
5. General intersections 176
6. Geometric extensibility 178
7. Communicating with the system 179
8. Conclusion. 179
References. 179

XI. ALGOL 68 G GRAPHIC EXTENSION OF ALGOL 68 by P.J.W. TEN HAGEN . . 180

1. Introduction. 180
2. Layered structure 181
3. The host language 182
4. The basic layer 183
4.1. Geometry. 184
4.2. Transformations and subspace. 187
4.3. Attributes. 188
4.4. Pictures. 189
4.5. Example 190

VOORWOORD

Het colloquium Computer Graphics werd gehouden in het academische jaar 1976/1977 op het Mathematisch Centrum.

Het werd gezamenlijk georganiseerd door de computer graphics groep van de Universiteit van Nijmegen, de computer graphics groep van het rekencentrum van de Technische Hogeschool te Delft en de Afdeling Informatica van het Mathematisch Centrum.

De serie voordrachten geeft onder meer een onverzicht van recente activiteiten op dit gebied in Nederland. Alle bijdragen hebben betrekking op eigen werk van de auteurs.

P.J.W. ten Hagen.

AN INTRODUCTION TO COMPUTER GRAPHICS

P. KLINT

0. Introduction

Several diciplines can be distinguished in the specialism called "picture processing" or "more dimensional data processing", like pattern recognition, computer aided design, computational geometry, semantic picture analysis and graphic languages. In this lecture we will not even be able to mention all these areas. Instead we restrict ourselves to one particular area: computer graphics (CG) and computer graphics system (CGS) organization. It is felt that this example is representative and can give some insight in the problems encountered in general.

In chapter 1 a short survey is given of graphics devices. Dissimilarities between various devices are stressed. Chapter 2 deals with graphics system organization and discusses some design goals and trade-offs. Chapter 3 presents a model for graphics i/o functions. This model is still under development and will be used in the Mathematical Centre computer graphics system.

1. Devices for computer graphics

This chapter is not intended as a primer on graphics devices. Reviews of the detailed working of such devices can be found in the literature (NEWMANN & SPROULL [1], ENCARNACAO [2]). The intention of this chapter is rather to stress the dissimilarities between various devices, since it are these dissimilarities that plague (and have plagued) many CG systems.

1.1. Output devices

1.1.1. Plotters

The digital plotter is the classical CG output device. Several types have evolved (drum and flat bed plotters) but their main characteristics are:

- incremental operation (i.e. operations are relative to the current pen position)
- a fixed number of drawing directions (typical 8 to 16)
- pen up, pen down and select pen (colour) operations
- it is not possible to erase lines, that are already drawn.

A recent development is the Ink Jet Plotter (SMEDS[3]). Its principle of operation is based on the deflection of charged ink particles. Three ink jets of electrically charged ink droplets in one of the primary colours are fired on a piece of paper. The intensity of each stream is modulated by means of an electric field, which transforms the ink stream into a spray. This spray hits a diaphragm and thus the amount of ink that reaches the paper depends on the electric field which is applied. In this way full colour pictures of reasonable quality can be produced.

1.1.2. Cathode ray tube (CRT) displays

These displays use the same technology as television tubes: an electrostatically or electromagnetically controlled electron beam writes on a phosphorescent screen. The places on the screen that are bombarded by the electron beam, glow for a short while (ca. 1/5 sec.) and produce a visible image. To obtain a stable image, the screen must be refreshed, that is the electron beam must write the same pattern on the screen repeatedly. Mainly two techniques are used to refresh the screen:

- the data from which the screen is refreshed are kept in an external memory and a special display processor sends these data to the CRT continuously (refresh displays).
- the CRT itself contains a fine wire grid that "remembers" the image written on the screen.

Typical operations that both types of displays can perform are:

- draw a vector (relative or absolute, visible or invisible).
- set intensity level.
- display text.

Since most refresh displays contain a separate display processor, their instruction repertoire can be very sophisticated:

- draw lines which blink with a chosen frequency.
- draw dot-dash lines according to a chosen pattern.
- jump, subroutine jump, subroutine return (useful operations for the implementation of segmented display files, see chapter 2.).
- translate, rotate, scale.

Since the image is refreshed at a high rate (ca. 30 times per second), it is easy to delete parts of the image and to produce moving pictures.

The less powerful (and thus cheaper) storage display has as main disadvantage that a long time is required to erase the whole screen and produce a new image. In chapter 2. we will see which problems this presents in an interactive environment.

1.1.3. Numerically controlled milling machines

These machines are used for cutting solid objects, made of metal or other material. The objects produced by a milling machine can be considered as an ultimate goal in graphic output, since the object itself is produced and not some picture of it. Very special problems are encountered when programming these machines. For example, the connectedness of the objects produced must be taken into account.

1.1.4. Plasma panel

The plasma panel (BITZER & SLOTTOW [4]) consists of a rectangular array of luminous gas discharge cells. By means of an addressing mechanism that resembles the one used in normal core memories, each individual cell can be made to glow or be extincted. This device achieves an almost perfect symmetry between input and output operations, since the gas cells can also be excited by the application of an external voltage to individual cells.

1.2. Input devices

1.2.1. Tablet

A tablet consists of a flat surface on which can be drawn by means of a pen or stylus. The tablet transmits the coordinates of the stylus position (or the change in stylus position) to its controlling computer. Any point on the tablet can be selected in this manner.

1.2.2. Light pen

The light pen is a pointing device and is always used in conjunction with a refresh display. Its operation is roughly as follows. Though its name might suggest the opposite, a light pen accepts light by means of a photo cell. If the light pen is used to point at an item on the screen, this photo cell "sees" light when that item is redrawn during the refresh cycle. At that moment the display processor is halted and an interrupt is requested in the controlling computer. The item being pointed at can now easily be identified by means of the contents of the display processors address register. Only intensified points can be selected by the light pen.

1.2.3. Buttons

Buttons or function keys are switches that are used to input an integer value in a fixed interval (say [0, 15]). In most applications a special meaning is attached to each button, such as the selection of a specific program action.

2. Graphics System organization

Basic concepts and terminology will be introduced in this chapter. An outline is given of the organization underlying most CG systems. Some design goals and tradeoffs are discussed (see for instance NEWMANN & SPROULL[5]).

Figure 2.1. gives an oversimplified view of the processes and data that are essential for the operation of an interactive graphics application program.

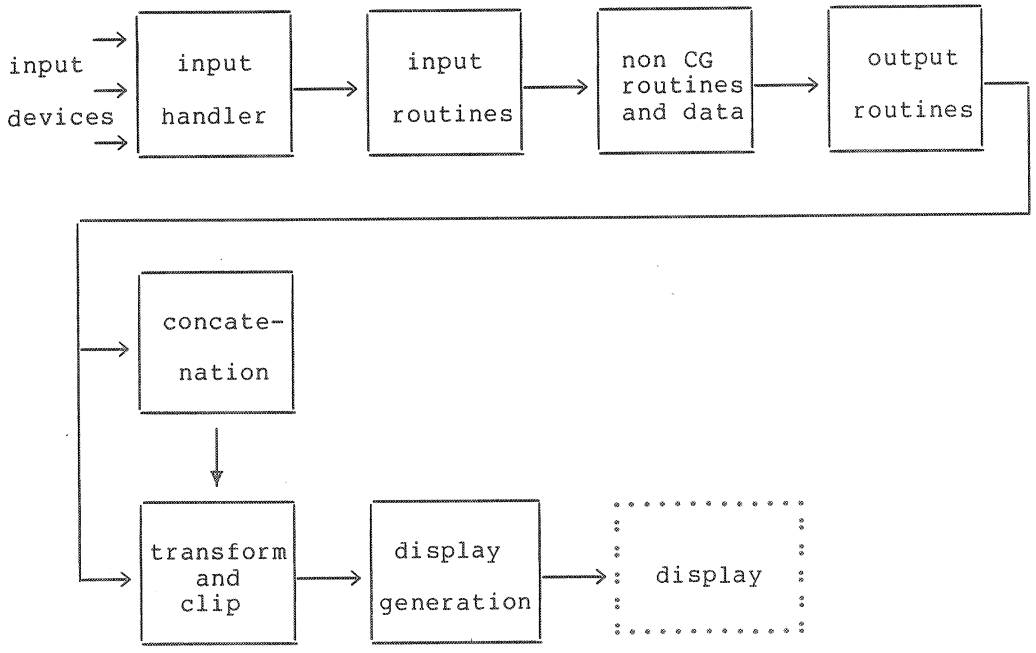


Figure 2.1. A simplified view of a CG system.

Input devices like keyboard, tablet and light pen provide the data and control information for the application program. An input handler services each input device and performs device specific operations.

The input routines receive their information from the input handler, possibly update some internal administration, and pass the input data to the application program in a uniform manner.

The output routines define the picture to be displayed. In some CG systems only sequentially ordered sets of simple pictorial elements like lines, points and characters can be displayed with various intensities, shapes (italicity) and styles (dot-dash lines). In other systems these primitive elements can also be collected in subpictures, which can be referenced from other subpictures. In this way complex hierarchycal pictures can be build.

In all but the most simple systems pictures can be transformed, i.e. scaled, rotated, translated and so forth. In most systems transformations are represented by matrices. The details of the representation vary from system to system

and will not be discussed. In general the output routines create pictures of arbitrary size in user coordinates. It is also a task of the transformation routines to convert these user coordinates to screen coordinates.

Sometimes a window can be defined in the user coordinate system that effectively removes all picture elements (or parts thereof) that lie outside this window. This process of removing undesired picture parts is called clipping. A viewport may be defined which specifies where the contents of the window should be positioned on the screen.

The task of the concatenation routines is to combine transformations whenever possible. It is of vital importance to replace many successive transformations (as occur with nested subpictures) by one overall transformation, before each picture element is transformed.

Display generation involves the creation and transmission of device specific order code, that visualizes (in some sense) the objects defined by the graphical output routines. The following discussion is centered around the question how the characteristics of display devices influence this display generation process.

As long as the graphical output routines are used just once, the mechanism of display generation given above remains sufficient. But as soon as modifications are being made to pictures that are already generated, the need arises for a less naive mechanism. We will now consider two such mechanisms. It will turn out that both provide only partial solutions and that under certain circumstances both mechanisms are required.

The first of these mechanisms, the viewing algorithm, is both very attractive and difficult to implement. The way in which the graphical output routines are called depends entirely on the data base associated with the application program and the application program itself. In a sense, the resulting picture can be considered as a view of the contents of the application programs data base. To obtain the effect of giving a continous view of the contents of this data base, the output routines must be arranged to be executed repeatedly at a sufficiently high frequency. This arrangement is called a viewing algorithm.

As simple as this method seems, in practice enormous difficulties are encountered with the generation of output at such a rate that a flicker free image of good quality is guaranteed. One solution to this problem is to restrict the class of transformations or pictures to which the viewing algorithm is applied. An other solution, to build a Structured Picture Description as result of the graphical output routines before the actual display generation, will

now be considered in more detail.

The Structured Picture Description (SPD) concept is based on the observation that in most cases modifications only affect minor parts of a picture, while the major part of it remains unchanged. Modifications can be described as edit operations on the SPD. To generate a picture a trace routine must traverse the SPD (figure 2.2.).

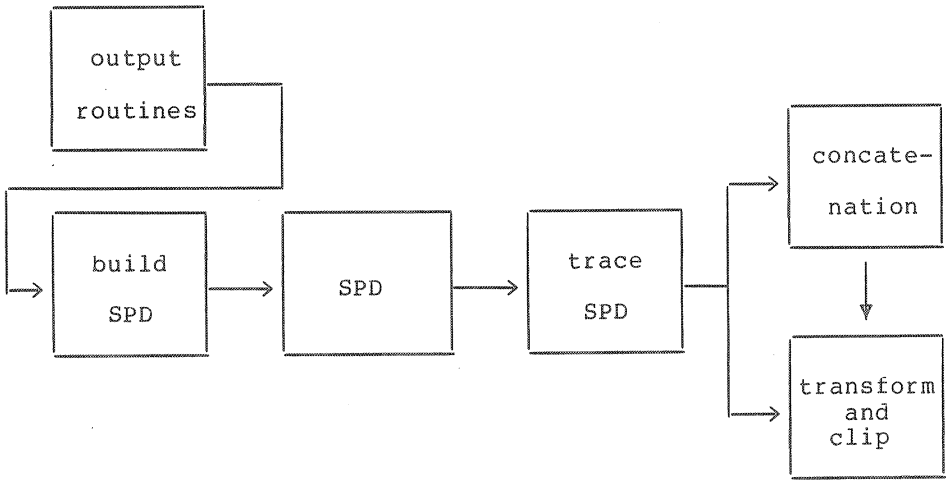


Figure 2.2. CG system with structured picture description.

The use of an SPD increases the overall complexity of the system since processes to build and trace the SPD are now required.

The second mechanism for the generation of the graphical output is the plotter analogy. In this model all output operations control a fictitious pen or beam. As is the case with real plotters, a picture store (the paper in the plotter) is inherent to this model. For refresh displays a file or data area with order code for that display constitutes the picture store. In other words, complete pictures can be erased but it is impossible to selectively erase parts of a picture.

To allow for selective erasing, the picture store can be replaced by a transformed display file, that contains the result of the transformation process (figure 2.3.). The transformed display file may be subdivided in logically distinct, named segments or records, that may be created, modified or deleted separately.

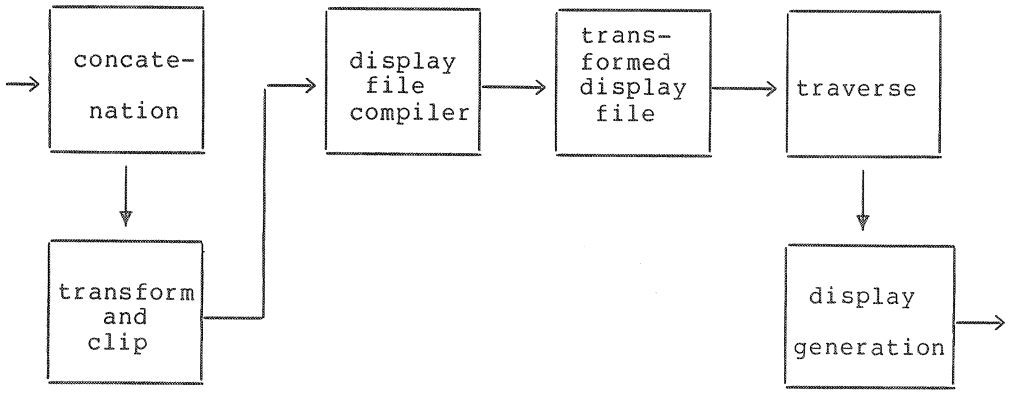


Figure 2.3. CG system with transformed display file.

As can be seen in figure 2.3. a display file compiler is required to generate the transformed display file, and a traverse process is required to generate actual order code from the transformed display file.

Both models presented, the structured picture description and the transformed display file have their disadvantages. For displays without hardwired transformation capability the SPD cannot be used directly, but a transformed display file must be constructed first. This introduces the problem when and how often this transformed display file must be regenerated from a modified SPD. A comparable regeneration problem exists with transformed display files and storage displays, since the display is not refreshed from the transformed display file. Transformed display files have as an other disadvantage that they form a low level representation of the original picture. They are completely determined by the characteristics of the output device.

What we have identified here, is the very fundamental choice of intermediate representation problem, that occurs in all software systems of any complexity. This problem arizes when:

(1) The output (or input) data are not completely determined. For example, the precise format of the object code generated by a portable compiler, is not known to the designer of that compiler. The variety of potential i/o devices presents an analogous uncertainty in the CG case. Therefore an intermediate representation is required from which the output data can be generated (or to which the input data can be converted).

(2) The problem to be solved is too complex and must be divided in subproblems (the scans of a compiler). Solutions to subproblems must be able to communicate with each other in the form of an intermediate representation.

Sometimes the intermediate representations required by (1) and (2) above are coincident.

It is useful to compare the design of a device independent CG system with the design of device independent software in general. One technique will be considered in detail, namely the "levels of language" or "hierarchy of abstract machines" approach.

Given a program P written in language L; how can P be made portable? We can either choose for L an ubiquitous programming language (FORTRAN IV) or design a new language L, especially suited to write P in. This last alternative has the disadvantage that L must be implemented for each transfer of P to an other computer. To facilitate the implementation of L, it has been proposed to provide a hierarchy {L[n], ..., L[Ø]} of new languages such that L = L[n], and L[i] can be translated to L[i-1] (n > i > Ø). A new implementor of P can now choose which L[i] is most suited for him.

The first alternative can not be used in the context of a CG system, since there does not exist a language that is generally accepted by all graphics devices. Remember that in the CG case, P is the result of the graphical output routines!

If we restrict our attention to the second alternative we discover that we have already encountered two elements from such a hierarchy in the CG case: the high level structured picture description and the low level transformed display file.

In the remainder of this chapter we will outline why it is not desirable, in the specific case of a CG system, to have several intermediate representations.

In a portable software system only one path from machine independent to machine dependent representations has to be selected at a time. This implies that, once L[i] is chosen, the other elements of the hierarchy can be short cut. Moreover, such a selection occurs only when the portable software is moved to another computer installation.

In a CG system more than one path can be selected at a time, since a user program can access several graphics devices simultaneously. Because it is not known in advance which devices will be used, no element of the intermediate

representation hierarchy that is specific to some device can be discarded.

In contrast with most portable software systems, there exists in a CG system a way back from graphics device to intermediate representation. This way back may consist of input actions or higher level picture editing operations. If more than one intermediate representation exists two situations are possible: either this backward transformation becomes very inefficient because input information must be converted to several intermediate representations in succession, or the overall complexity of the CGS increases because all possible backward transformations are incorporated in the system. In the latter case, the undesirable situation arizes that different versions of a picture represented by different elements of the representation hierarchy exist.

The fact that no "cookbook" solution can be given for the design of a CGS, explains why this is still a subject of current research.

3. A model for graphics i/o functions

3.1. Motivation

The preceeding chapters show clearly the differences between various graphics devices and the influence these differences have on graphics system design. Given this large range of hardware features, one is faced with three problems:

- to provide the user of the CG system with a (small) conceptually consistent set of primitives
- to allow the exploitation of exotic hardware features
- to minimize the effort required to incorporate a new device in the system

The first and third requirement can be met by providing as primitives the "least common denominator" of all graphics devices known. But as a consequence many hardware features can not be supported by the system. On the other hand one might try to include primitives for all kinds of hardware features. The result will be a huge, unstructured set of device dependent primitives.

It is worthwhile to consider a model for graphics i/o functions that combines the advantages of the solutions just proposed, but has less drawbacks. This model is mainly concerned with the organization of the interface between graphics system (or more precisely an SPD) and graphics devices. Most details of this interface itself can not be discussed here. Examples of other efforts to obtain machine independent graphics systems are described in BLINN & GOODRICH[6] or SWITZKY[7].

3.2. Model description

To explain the model, we must first define a kernel of picture primitives. We require that the device driver for each drawing device is able to perform some meaningful action for each primitive in the kernel. If we consider output primitives, this kernel might consist of:

MOVETO (absolute, invisible beam displacement)

MOVEBY (relative, invisible beam displacement)

DRAWTO (absolute, visible beam displacement)

DRAWBY (relative, visible beam displacement)

POINTTO (absolute, invisible beam displacement with endpoints intensified)

POINTBY (relative, invisible beam displacement with endpoints intensified)

TEXT (display characters)

We will see that this set of primitives has a useful interpretation even for gray-scale devices. The list of possible non kernel primitives is probably not finite, but some likely candidates are:

draw polygon

draw curve

draw surface

draw object with hidden lines removed

draw object projected along a certain axis

All these primitives are higher level, device independent drawing operations. Note that the list is ordered (as far as possible) according to increasing complexity of the prim-

itives. In the sequel primitives will be identified by their position in this list. In following sections input primitives (3.3.2.) and picture manipulation primitives (3.3.3.) will be described.

By means of a device description table (DDT) the relations between primitives and hardware features of a specific device can be established. A DDT consists of:

- a list (KERNEL) of addresses of routines that implement the primitives in the kernel for this device
- a list (ACTIONS) of addresses of routines that implement kernel and other primitives for this device
- an integer (NENTRIES) defining the number of entries in the ACTIONS list

The kernel primitives appear twice in the DDT, since ACTIONS contains also the addresses of kernel primitive routines.

In principle the ACTIONS list corresponds with the list of primitives in the system, but the ACTIONS list may be shorter as we will see. If primitive number P is available as hardware feature, ACTIONS[P] contains the address of a routine in the device driver, that generates code to perform the action corresponding with primitive P. If P is not available as hardware primitive and P is a legal index in the ACTIONS list there are two possibilities:

1. ACTIONS[P] contains the address of a standard system routine that implements primitive P in software by means of more elementary primitives (primitives with a lower index).
2. ACTIONS[P] contains the address of a device specific routine. This routine can use hardware features of the same or higher level than that of primitive P.

Otherwise standard system routine number P is used to implement primitive P. Thus the selection mechanism is as follows:

```
if P < NENTRIES then
        execute ACTIONS[P]
else
        execute standard routine[P]
fi
```

The inclusion of NENTRIES in the DDT is more important

than it may seem at first sight. We indicated already that the list of primitives is not finite, or will at least grow during the life of a graphics system. If a new primitive is added to the list only the DDTs of devices that give hardware support for the new primitive need to be modified. All other DDTs remain unchanged thanks to the selection mechanism described above. In a sense NENTRIES can be considered as a complexity measure for the device under consideration. For primitives of complexity less than NENTRIES the device can sometimes give hardware support. For primitives of complexity greater than NENTRIES the device can never give hardware support.

Operations that control the state of a drawing machine can also be considered as candidate primitives. We will investigate how the method works for a specific example: the generation of dot-dash lines. Two primitives are involved: DRAWTO and SET_LINESTYLE. For the moment we forget about DRAWBY.

If a device supports dot-dash lines in hardware, the task of SET_LINESTYLE is straightforward, namely send control information to the drawing device to initiate dot-dash line generation.

If a device does not support dot-dash lines in hardware, the situation is more complicated. The desired effect can be achieved if SET_LINESTYLE changes the entry ACTIONS[DRAWTO] to let it contain the address of a standard system routine STYLED_DRAWTO. STYLED_DRAWTO generates lines in the style defined by SET_LINESTYLE and uses the primitives in the KERNEL of the DDT for this purpose. It will by now be clear why kernel primitives appear twice in the DDT. This approach has the disadvantage that DDTs can not be shared among several user programs. Figures 3.1. and 3.2. depict the whole scheme.

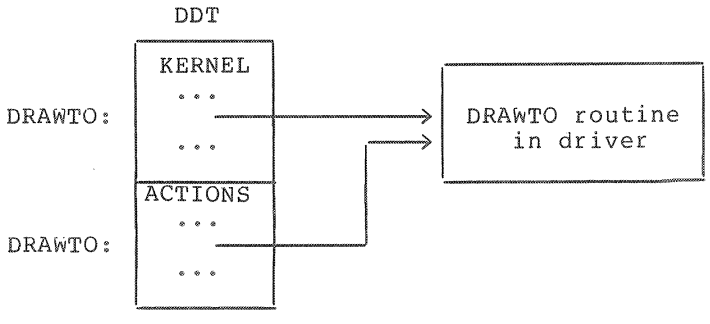


Figure 3.1. DDT before SET_LINESTYLE call.

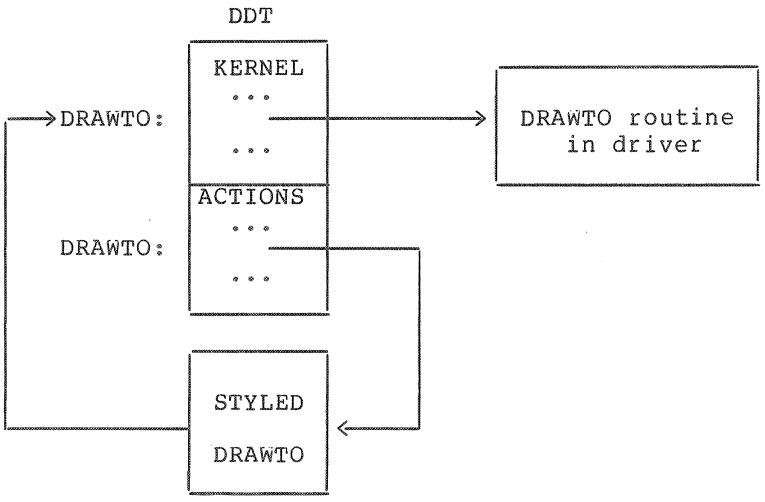


Figure 3.2. DDT after SET_LINESTYLE call.

Two final remarks conclude this discussion. In the first place it must be noted that the interaction between several primitives may be very subtile. For this reason it might be advantageous to divide the ACTIONS list in classes of mutually not dependent primitives. In the second place one must bear in mind that the model presented here does meet many, but not all off the requirements stated in the beginning of this chapter. Since a certain primitive and hardware primitive may resemble very much, it may not be possible (or only at extra cost) to implement the primitive using that hardware feature because of minor differences in their description method. The different manners to define a circle form an example.

3.3. Choice of primitives

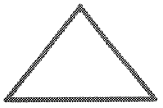
The DDT is used to interface structured picture description (SPD) and graphics i/o devices. In other words the DDT provides the semantic primitives required by the SPD traversal process. From this it follows that at least the SPD primitives must be part of DDT or standard system routines. Some more specific examples will clarify this point.

3.3.1. Output

In the previous section the output primitives MOVETO, MOVEBY, DRAWTO and DRAWBY were introduced. It is obvious that these primitives have a useful interpretation for line drawing machines. We will now give an interpretation for gray-scale devices (NEWMANN & SPROULL [8]).

The result of the drawing operations:

```
MOVETO(0,0);
DRAWTO(20,40);
DRAWTO(40,0);
DRAWTO(0,0);
```



under the standard interpretation in conventional CG systems is as indicated.

To allow the creation of opaque or filled objects, the beginning and end of the boundary of such a filled object must be marked. The primitives BEGINFILL and ENDFILL serve this purpose. ENDFILL and MOVETO always close any filled object in progress, so an additional DRAWTO is not necessary. In this manner a filled triangle is defined by:

```
BEGINFILL
        MOVETO(0,0);
        DRAWTO(20,40);
        DRAWTO(40,0);
ENDFILL
```



Additional MOVETO - DRAWTO sequences inside a BEGINFILL - ENDFILL pair can be used to specify additional boundaries, such as holes:

```
BEGINFILL
        MOVETO(0,0);
        DRAWTO(20,40);
        DRAWTO(40,0);
        MOVETO(10,10);
        DRAWTO(20,30);
        DRAWTO(30,10);
ENDFILL
```



3.3.2. Input

3.3.2.1. Input primitives

The choice of graphic input primitives is difficult. Graphic input functions offer a large, but unstructured collection of programming techniques and hardware facilities. In addition most input functions require some kind of feedback (echo) and are thus dependent on the output primitives.

In general each input primitive is able to cause an event of one or more types. All these events are collected in a socalled event queue. At the application program level only one type of input statement exists: GET_EVENT. GET_EVENT returns the next element from the event queue (if any). In this manner the application program does not have to specify from which device input is expected.

An input primitive can only cause an event if it is enabled, i.e. the user program is prepared to consider events caused by this particular input primitive. Otherwise the primitive is said to be disabled and can not cause events. This construction prevents the continuous attention required by input devices that cause events at a fixed rate.

We discuss now first a set of primitives that can be used to describe most existing input techniques (see for example FOLEY & WALLACE [9] or WALLACE [10]). Next some specific examples are given.

Input primitives can be divided in selectors and valuators. Selectors are used to select an element from some user or program defined universe. The two selectors: pick and button have different universes. Valuators are used either to determine values in some vector space (potentiometer, locate) or to read alphanumeric data (text).

The selector pick chooses between user defined objects, and can be used to select particular lines, arcs, subpictures and the like. The result of the pick operation is a reference to the object being pointed at. All objects have as part of their definition a detectability attribute that determines whether the object can be selected by a pick operation. The classical pick device is the light pen.

The selector button chooses between program defined objects and can be used to select a particular program action. The result of a button selection is an integer. The interpretation of this integer may vary from program to program.

Valuators are simpler, but not less useful input primitives. The valuator text delivers as value a string of characters input from the graphics device by means of an al-

phanumeric keyboard, on-line character recognition or any other method whatsoever. A text event is generated as soon as a character from a user specified set is encountered. Thus an event may be generated for each input character (user specified set is equal to whole character set) or only when a specific character is encountered (example: user specified set contains only the newline character).

The valuator locate delivers as value coordinates in screen space and can be used to select an arbitrary position on the screen. The result of a locate operation is a vector of screen coordinates. Prototype locate devices are tablet and joystick. Locate can cause two different types of events:

- the locate event proper, announces the selection of a position on the screen
- the locate boundary event: a position is selected that lies closer to a boundary of the current viewport than some user specified distance.

The latter event is necessary to implement rubber band and dragging techniques as we will see in section 3.3.2.2..

The valuator potentiometer delivers as value a real number in the range [0,1]. The prototype device is obvious. As is the case with locate, a potentiometer can cause two types of events:

- the potentiometer event proper
- the potentiometer boundary event: a value is delivered that lies in the interval [0,alpha] or [betha,1], where alpha and betha are user defined values ($0 < alpha \leq 1$).

3.3.2.2. Implementation of more complex input operations

Given the set of input primitives just mentioned, it is straightforward to implement more complex input operations. As an example we show how dragging can be implemented by means of these input primitives.

Dragging proceeds in two steps. First a picture or picture part on the screen is selected. Next the selected item is "attached" to a positioning device and the position of the selected item is changed whenever the screen position at which the positioning device points is changed. If the item has reached a desired position, it is "detached" from the the positioning device. The following sequence of input primitives implements dragging:

1. start dragging (button)

2. select item (pick)

3. attach (button)

4. move item (locate)

5. detach (button)

The item being moved remains unchanged as long as no part of it crosses a viewport boundary. To be able to detect the crossing of a viewport boundary, a locator causes a locate boundary event if it points to a position within a certain distance from the viewport boundary. For this distance one can choose the overall size of the item being dragged.

3.3.2.3. Incorporation of input primitives in the DDT

It is the responsibility of each device driver to handle the interrupts of each specific input device. However, the detailed information provided by these input devicess does not reach any other system module than the device driver itself. The device driver can pass input information to higher levels in the system by means of the system routine CAUSE_EVENT, that adds a new entry to the event queue. This new entry contains a description of the input event and can ultimately be inspected by the user program by means of GET_EVENT. This information stream can be divided into smaller components as is shown in figure 3.3.. The interrupt routine, which is part of the device driver, delivers input descriptors in a machine dependent form to the parse input process. Parse input is the counterpart of the traverse SPD process on the output side of the system. Parse input calls (via the DDT) machine dependent routines to interpret input descriptors for any of the five input primitives. In this way symmetry between input and output primitives is achieved.

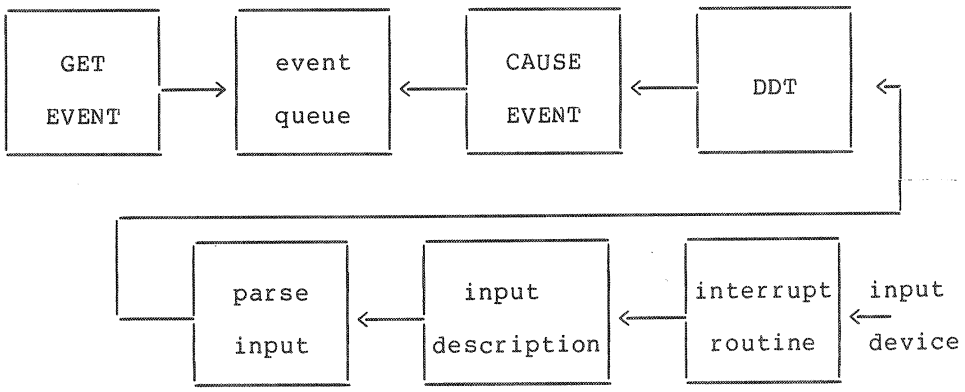


Figure 3.3. Incorporation of input primitives in DDT.

The details of the operation of an input device, get lost on the path from input device to user program. Thus each device driver defines which input device can realize which input primitive and even to simulate non-existent input devices. Ultimately all input primitives can be simulated on a 64 character set keyboard, but it is not clear whether people can be hired to use this input facility.

3.3.3. Picture manipulation

For the purpose of this discussion we consider as SPD a tree-like datastructure as can be found in several special purpose graphics systems for electrical network design and civil engineering applications. The nodes in this datastructure contain:

- control information (transformations, linestyles etc.)
- pointers to subtrees (subpictures)
- actual drawing information

A node is a leaf (terminal node) if it does not contain pointers to subtrees.

Only three edit operations are required to (re)construct a picture from a (modified) SPD:

CREATE_NODE

DELETE_NODE

MODIFY_NODE

Thus these operations should be added to the kernel.

As we have seen before, the DDT entries corresponding to CREATE_NODE, DELETE_NODE and MODIFY_NODE may either contain the address of a machine dependent routine or the address of a standard system routine. In the former case advantage can be taken of hardware transformation and clipping. In the latter case software transformation and clipping are used and new nodes are generated using DDT entries such as MOVETO and DRAWTO.

The precise contents of the DDT and of the DDT routines also dictate the moment when a picture must be regenerated for storage displays. Thus a different strategy can be used for different devices. A reasonable strategy is to regenerate the whole picture for each deletion but not for additions. Since this method involves the transmission of considerable amounts of data, an other scheme may be preferred.

4. Bibliography

[1] NEWMANN, W.M. & SPROULL, R.F., Principles of interactive computer graphics, McGraw Hill, 1973

[2] ENCARNACAO, J.L., Computer-Graphics: Programmierung und Anwendung von graphischen Systemen, R. Oldenbourg Verlag, Munchen, 1975

[3] SMEDS, B., A 3-colour ink jet plotter for computer graphics, BIT 13 (1973) p181-195

[4] BITZER, D.L. & SLOTTOW, H.G., The plasma display panel - A digitally addressable display with inherent memory in AFIPS FJCC 29 (1966) p541-547

[5] NEWMANN, W.M. & SPROULL R.F., An approach to graphics system design, Proc. of the IEEE, 62 (1974) p471-483

[6] BLINN J.F. & GOODRICH, A.C., The internal design of the IG routines, an interactive graphics system for a large timesharing environment, in Proc. of the third annual conference on computer graphics, interactive techniques and image processing - SIGGRAPH 76, Computer Graphics 10 (1976) 2, p229-234

[7] SWITZKY, C.M., Device independent graphics through table driven translation, University of California at Los Angeles, UCLA-ENG-7655, 1976

[8] NEWMANN, W.M. & SPROULL R.F., The design of gray-scale graphics software, in Proc. of the conference on computer graphics, pattern recognition & datastructure, may 14-16 1975, p18-20

[9] FOLEY, J.D. & WALLACE, V.L., The art of natural graphic man-machine conversation, Proc. of the IEEE, 62 (1974) p462-471

[10] WALLACE, V.L., The semantics of graphic input devices, in Proc. ACM symposium on graphic languages, 26-27 april 1976, SIGPLAN notices 11 (1976) 6, p61-65 or Computer Graphics 10 (1974) 1, p61-65

G P G S
GENERAL PURPOSE GRAPHIC SYSTEM

L.C. CARUTHERS
Informatika / Computer Graphics Group
University of Nijmegen Faculty of Science

1.0. INTRODUCTION

GPGS is the definition of a universal graphics interface to be used as an extension of high-level languages. The subroutine call mechanism common to these high level-languages provides an easy means of extending them with facilities for interactive and passive graphics. Thus any (mini)computer with FORTRAN is a possible candidate for a GPGS implementation. By taking care of all the problems associated with operating systems, communications and the physical characteristics of devices, GPGS provides for the portability of graphics programs between devices and even between computer systems.

In 1972 common graphics hardware for the Universities of Nijmegen, Delft en Cambridge was recommended, and subsequently acquired. The desire for common software for these configurations lead to the design of the GPGS subroutines. GPGS was designed to replace and improve upon such packages as IBM's GSP [1] for 2250's and Calcomp's well known plotting subroutines. Among the three universities experience with device independent graphics had already been gained with the Cambridge GINO system. Rather than reimplement GINO for the new [2] hardware it was decided to design a new package with more facilities than GINO and to give the subroutines a better packaging with different names.

Once the initial design of GPGS was completed in 1972, parallel implementations were begun on an IBM 370 with a PDP-11/45 as a satellite at Nijmegen and on a PDP-11/45 in stand-alone mode at Delft. These two

assembly language implementations have been in more or less continuous development since then. In 1974-75 the graphics group at the computing center at the University of Trondheim in Norway made an ANSI FORTRAN implementation based on the Delft PDP implementation.

The agreement between Nijmegen and Delft has always been that an application program would run equally well on either implementation. This agreement, together with the parallel implementations on different computers, created a situation where each group checked the work of the other. As the implementations progressed both groups also checked the initial design, the changes to which were negotiated to suit both parties. Thus the dual implementations have ensured a doubly proven design.

2.0. DESIGN CONSIDERATIONS

The design considerations will be discussed in two groups. First those matters pertaining to the creation of a subroutine package and GPGS in general will be discussed. This will be followed by those issues more specifically concerned with device independence and an explanation how GPGS looks at real devices.

2.1. What Kind of Subroutine Package?

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

The limitation of a subroutine package, when compared with a lanquage, is that you only have one type of syntax for expressing the graphics functions, a subroutine name followed by its easily forgettable arguments. On the other hand this simpler syntax makes the package easier for programmers to understand, especially because the syntax is that of a language they already know. GPGS also provides a graphics interface for existing non-graphics programs or applications packages.

A very large advantage of a subroutine package is that it automatically splits the implementation of the package into small modules. Each module corresponds roughly to a single facility of GPGS. Thus the space

(memory) overhead incurred by an applications program is proportional to the number of GPGS facilities used.

The subroutines included in GPGS were chosen to be just far enough removed (indirect) from the hardware to provide device independence and at the same time allow the applications programmer to (indirectly) control the hardware of an advanced CRT display. An additional guideline as to what features to include in GPGS was given by the desire to make the package *general purpose*. Thus those features required or extremely useful for a wide range of applications (2 and 3d windowing and clipping, transformation) were included. Those with more limited applicability like hidden line removal, data structure, and animation, were not included.

When designing the subroutines themselves the key concept was *simplicity*, meaning that the subroutines should be easy to understand and to use. The name of a subroutine indicates what its function is an each subroutine has as few arguments as possible. GPGS supplies reasonable default values for any unspecified picture making conditions.

2.2. Device Independence

In the days when only device dependent (manufacurer supplied) graphics packages were available, applications programs that provided a facility for plotting a picture that had been created on-line, required the simultaneous use of two different packages. Each package has its own device independent overhead, a double cost of space for the applications program.

To make GPGS *device independent* it was decided that each implementation should be divided into a device independent part to be called by applications programs, and as many execution time loaded, device dependent *device drivers* as there are devices for that implementation to support. To the applications programmer this means that he can write his graphics program once and use it with different graphics equipment without changing the source code or relinking his program. It also meant that the device independent overhead part of the graphics package only had to be incurred once for all the graphics devices that the program may be using. Since a device driver only needs to be resident in core when it is in use, two or more drivers could alternatively use the same core space. For managing resources GPGS allows the initialization of several devices (NITDEV) or several buffers (NITBUF) before the corresponding releasing of any of the

resources. To allow the applications programmer to designate which device or buffer is to be used there are routines for selecting (SELDEV, SELBUF) the device or buffer to be used. The selected device or buffer is then referred to as the "current" device or buffer.

Since his program may run with a different device each time it is executed, the applications programmer should imagine that he is programming for a single *idealized device*. This idealized device draws pictures by moving a drawing mechanism from position to position. This characterization of graphics output devices applies to plotters, microfilm output, storage tube displays and moveable beam CRT refreshed displays. By designing GPGS to handle refreshed CRT's, a design was achieved which was suitable for the simpler types of devices as well.*

The idealized device concept is also appropriate for describing the GPGS scheme for handling input from the user sitting at the display console The GPGS idealized device has the following single tools:
refresh clock, alphanumeric keyboard, lightpen for picking, audible alarm; and at least one tool in each of the following *classes of tools*: dials (1 dimension), tracking cross and tablet (2 dimension), joystick (3 dimension), function switches. Each single tool or class of tools has a format for returning integer and/or floating point data to the applications program. Those tools not available in hardware may be simulated in software by the device drivers.

GPGS uses the concpet of an idealized device with real tools rather that a set of abstract tools [5,4] for all devices. The benefits of this approach are that when real hardware exists the applications program can use it directly, whereas with the abstract tool approach there will always be a layer of software mapping real tools into abstract tools. This extra indirectness is a conceptual burden for both the applications program writer and the console user, both of whom must figure out what the use of an actual hardware tool is going to mean to the program.

---

* By its design GPGS provides downward comptability from more complex to simpler graphics devices. Upward comptability between GPGS devices is achieved by software simulations in device drivers which give the device the appearance of having more facilities than actually appear in the hardware.

2.3. Real Devices as seen by GPGS

At this point it is appropriate to explain how graphic devices are viewed by the GPGS designers. Some graphic devices like plotters can never be interactive. Due to this physical limitation some GPGS devices are interactive and some are not. It would be nonsense to call a GPGS subroutine requesting interactive facilities from a non-interactive device.

Another major distinction between devices in GPGS is whether or not a device is buffered. This is purely a logical distinction because buffering is the software facility for storing all the picture elements that are used to make a picture. As described in the GPGS User's Tutorial, all the device dependent code for each physical device is in a device driver module.

Graphics Devices and picture buffers are explicitly controlled resources provided by GPGS to the applications program. A picture buffer is always required for controlling the picture building process, even for devices that do not store their picture in a buffer. Before picture making can begin device and buffer resources must be allocated to GPGS. Buffering is a software facility which is either provided or not provided by each device driver.

In Appendix C there is a table of the GPGS facilities that are available from the different types of device drivers. The device drivers are classified as to whether or not they are buffered and whether or not they are interactive. The "pseudo driver" is an abbreviated way of describing the picture compiler for pseudo picture segments which is implemented as a device driver module.

3.0. GRAPHIC I/O

The GPGS facilities are based on the activity cycle of interactive programs. First a picture is presented on a display, then input is received from the console user, which is used to make the next picture. In this section we will first describe how an applications program uses the facilities to build a picture, and second how the applications program obtains information from the console user.

3.1. Picture Making

The interaction cycle begins with the applications program making the first picture. The applications program does this via subroutine calls.

GPGS software processing in the CPU prepares a display file, or picture program for the display processing unit (DPU) of the display device. Since the applications programmer is always thinking of the idealized GPGS device, he can always imagine that he is building a display file.

A common representation of a picture is a tree structure. In such a tree structure each lower level of a tree is a component breakdown of a higher level. The final bottom of the tree consists of the indivisible picture elements. To make a picture of an object represented by the tree-like data structure, the applications program would scan the tree structure to find the picture elements at the bottom nodes. These picture elements would be passed to GPGS via subroutine calls where they would be converted into display file elements and placed in a linear display file, one after another.

3.2. Picture Segments and Picture Elements

The display file is built of ordered collections of picture elements called picture segments. Picture Segments are the GPGS unit of picture manipulation. Picture Segments may only be created, extended an deleted. The beginning of a picture segment is indicated to GPGS by an applications program call to the BGNPIC routine, which passes the unique picture segment identifying number to GPGS.

After BGNPIC the applications program calls picture element creation subroutines to fill the picture segment. A call to ENDPIC terminates the definition of a picture segment and serves as a command to make the picture segment visible.

Picture segments are built by the subroutine calls that pass picture elements and picture elements attributes to GPGS. GPGS accepts line segments, character strings and circles as picture elements that can be specified by one call to GPGS. One attribute of a line segment is line type (solid, dashed, dotted, endpoint or invisible) of the line. This attribute must be specified with each call to create a line segment. One attribute of a character string is the size of the characters, which unlike the line type of lines, is a global condition which is specified to GPGS by a separate subroutine, CSIZES. All character strings then are made with the same size until another call to CSIZES is made to change the character size.

A third kind of picture element attribute is one that is global within a picture segment and gets reset to a default value by each call to BGNPIC. These attributes include: blinking, intensity, colour, depth modulation and lightpen sensitivity, for CRT displays these last attributes correspond to processing conditions of the display processing unit (DPU). Thus the call to request blinking picture elements would precede the subroutine calls to create the picture elements which will blink when they appear on the CRT display. So you can see that calls to set picture element attributes will be intermixed with the calls to create picture elements during the building of a picture segment.

To provide for the creation of a device independent representation of tree structure pictures, GPGS has pseudo picture segments. A pseudo picture segment is built like a normal picture segment by passing picture element attributes and picture elements to GPGS. The picture elements are processed thru the software pipeline and then stored in device independent format in the pseudo picture segment. Pseudo picture segments are allowed to contain one additional type of picture element, a "reference" to another (lower level) pseudo picture segment. Thus a tree structure of pseudo picture segments can be built to correspond to the tree structure data structure that reperesents an object.

Once pseudo picture segments have been created they can be used many times to build display picture segments. When the applications program calls the INSERT routine with the number of a pseudo picture segment, picture element attribute settings, names and picture elements are retieved from the pseudo picture segment, passed back thru the CPU processing pipeline and given to the currently selected device driver. During the inserting process all the references to lower level pseudo picture segments are followed, so the result of an insert is always picture elements without the references.

3.3. Setting Viewing Conditions

Picture elements are passed to GPGS as cartesian coordinates in a 2 or 3 dimensional user coordinate space. Since the coordinate representation of an object only occupies a portion of the infinite coordinate space, GPGS must be told what part of the coordinate space is to be used for the picture. This is done by specifying a rectangle or box (*window*) to GPGS as high and low houndaries on each of the X, Y (and Z) coordinate axes.

Once the applications program has indicated the portion of the user coordinate space that is to be displayed, it may also specify a *viewport*, which is the portion of the display surface that the contents of the window are going to appear on. To provide a frame of reference for specifying viewport limits to GPGS, Normalized Device Coordinates are defined to have the range 0.0 to 1.0 along the side of the largest square area (cubic volume) that will fit on the display. If the display area is not square then the viewport specifications allowed for that device will have legitimate values outside the range of 0.0 to 1.0.

But what does this have to do with picture segments? A picture segment can have only one viewport and preferably only one window. These viewing conditions must therefore be established for each picture segment before the creation of the picture segment is begun by a call to BGNPIC.

3.4. Picture Element Processing Pipeline

Picture elements passed to GPGS go through the software (CPU) processing pipeline where all the previously allocated resources are used and all the picture making conditions are applied to the picture element. In this way the picture element acquires all the attributes the applications programmer wants it to have when it appears on the display.

For line segments the processing pipeline begins with the homogeneous coordinate transformations which, if they have been requested, are applied to the coordinates of the line. The transformed coordinates are then compared with the boundaries of the window to see if any part of the line segment is to appear on the display. If clipping has been requested some or all of the line segment might not appear on the display. Next, for those line segments which have not been clipped, the transformed coordinates are converted into fractions of the window in a range of 0.0 to 1.0. These fractional coordinates are then passed to a device driver which uses them as the corresponding fractions of the viewport for deciding where to draw the line segment.

3.5. Picture Segment Manipulations

After a call to ENDPIC there are only a few manipulations still permitted for a picture segment. The picture segment can be "reopened" for extension by adding more picture elements, and the picture segment can be deleted. Further, the visibility can be switched off and on, as can the

lightpen sensibility of the whole picture segment. Finally, if the device has real, or device driver simulated, "hardware" transformations, the size, location and orientation of the viewport on the display surface may be changed. These viewport transformations provide dynamically changeable viewing transformations which are very nice for the visualizing of three dimensional objects and elementary animation.

3.6. Input from Console Tools

The graphic output facilities of GPGS enable the applications program to make a picture on the display. But once the picture is made the applications needs information from the console user to find out how the picture should be changed.

For easy identification by the applications program each tool is assigned a permanent integer identifier. To request information from the interactive console the applications program passes GPGS a list of the tools that it is willing to accept information from as a parameter of the INWAIT function subroutine. GPGS then looks at the interrupt queue, to see if the console user has used any of the tools, if he has, the information from that tool is returned to the applications program. If no tools have been used yet, then GPGS examines the time parameter which was also passed with the call to INWAIT. If the time is positive GPGS will return to the applications program either when the time expires or the console user uses a tool. If the time is zero, information is returned from a tool only if the console user had used the tool prior to the call to INWAIT, otherwise GPGS returns without providing any tool information. If the time parameter is negative, INWAIT returns to the applications program only after the console user uses one of the tools in the list. Though not all implementations currently support it, INWAIT is designed to wait for information from more than one device at the same time.

The queing discipline used by GPGS is to allow each tool of each initialized device to make at most one entry at a time in the common (to all GPGS devices) interrupt queue. Thus the first interrupt from a tool stays in the queue until it is either passed to the applications program or rejected because it was not requested by the applications program. The interrupt queue can be cleared by calling INWAIT twice with the refresh clock as the only tool in the list of tool codes.

When INWAIT returns to the applications program it gives back the information from only one tool. To allow the applications program to inquire as to the status of other tools, GPGS has the REATOL subroutine which has the tool number as input and returns information in the same format as INWAIT. On many kinds of display hardware there are program controlable mechanism for assist with getting information from the console user. To allow the applications program to set various tools, GPGS has the WRITOL. Writol has the same parameter list as REATOL, but the information is going to the tool and not coming from it.

3.7. Additional Facilities

Picture segments may be stored off-line in picture segment libraries which are controlled by the applications programmer much in the same way as the device and buffer resources. Indeed libraries can be thought of as extensions of buffers. They are particularly useful for making large pictures on small computers, saving display picture segments of stand menu's or as a file of standard drawing symbols (picture parts) stored as pseudo picture segments.

To allow the applications program to find out the properties and status of its currently allocated resources, and to retrieve previously established attribute settings, GPGS has subroutines for returning execution environment information to the applications program. This information is sometimes very useful when used in conjunction with the GPGS error handling facility which will allow the applications program to specify one of its subroutines to receive control on the occurrence of an error condition.

4.0. CONCLUSIONS

In this final section we will evaluate the succes of the GPGS design and implementations in meeting design objectives. The discussion will examine how well device independence was achieved, how easy GPGS has been to use, the consequences of not having a data structure, the acceptance of the overall design, and the suitability of GPGS for writing programs in specific applications areas.

4.1. Acceptability of Design

The decision to make a subroutine package instead of a language has proven to be a good one. This is largely seen in the fact that there are multiple implementations and that the subroutines are easily understood and used by applications programmers. Indeed the decision to make a new package and not to reimplement GINO is born out by the fact that the Trondheim group considered both the GINO and GPGS designs, and schoose GPGS even though it would mean making a separate implementation.

4.2. Access to Devices

Device Independence has been achieved. Programs that can make a plot on a plotter can make the same picture on a refresh CRT. But in order to allow this, the plotting program has been forced to abide by the same picture segment creation rules as a CRT program. In our experience in writing device drivers we have seen that a driver for a simple device like a plotter or printer is very easy to write, and that the driver for an interactive device, though much more work is certainly simpler than creating a whole new package and conversion interfaces for other devices. Making a new driver is usually a matter of modifying the lowest level of some existing driver. The table on the next page shows the drivers implemented at the "home" installations of three GPGS implementations:

Devices Supported	GPGS Implementations		
	IBM-370 Nijmegen	PDP-11 Delft	FORTRAN Trondheim
Vector General	X	X	
Tektronix 4010---4015			
Buffered	X		
Unbuffered		X	X
Plotter: Calcomp	X		
Tektronix		X	
Kingmatic			X
Printer		X	X

The decisions to leave out a data structure and make a device independent picture representation optional have resulted in packages which take little memory space and which run quickly. With its Tektronix 4014 (15) driver the RT-11 version of GPGS is smaller than the nucleus of the

batch version of the RT-11 operating system. Similarly the basic routines for making a picture and interaction with the CRT and the satellite of the IBM-370 implementation takes about half again as much space as the FORTRAN I/O modules (IBCOM + FIOCS). For highly interactive (not much computation) applications programs on the IBM implementation, the CPU utilization and response time are comparable to that of text editing programs.

4.3. Suitability for Applications

The final comments on the accomplishments of GPGS are on how "general purpose" GPGS has proven to be. That is, how easy it is to write applications programs. For computer-aided design programs, where a fairly low level interface is needed along with multiple devices (interactive and plotter), GPGS has proven to be ideal. For people who just want to make plots of their data GPGS is useable, but nonetheless a set of graph making routines to go on top of GPGS has been designed.

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

Thus we have seen that GPGS has largely achieved its original design goals of being a device independent, easy to use subroutine package. GPGS provides applications programs with access to multiple graphics devices through the same subroutine calls. The GPGS design has given to be implementable a small and large computers alike.

REFERENCES:

1. IBM - Graphics Subroutine Package (GSP) for FORTRAN IV, COBOL and PL/I: form GC27-6932.

2. GINO, P.A. Woodsford (Ed.), Computer Aided Design Group, Corn Exchange Street, Cambridge, England.

3. GPGS Reference Manual, D. Groot, E. Hermans, Rekencentrum, T.H. Delft and L.C. Caruthers, J. Patberg, Informatika, Faculty of Science, University of Nijmegen.

4. J.D. Foley and V.L. Wallace, The Art of Natural Man-macine Communications, Proc. IEEE. 62, 4, pp. 462-471, 1974.

5. U Trambacz, Towards Device-Independent Graphics Systems, Computer Graphics (siggraph-ACM) 9, pp. 49-52, 1975.

Appendix A: Sample Program

The sample program* on the following page is a straight forward demonstration of the elementary graphics output and interactive facilities of GPGS. After declaring the necessary arrays and putting the constant data in the list of tool codes and the initial visibilities of the line segments, the program initializes a buffer and what should be an interactive device. The picture segment number 100 has six named lines followed by two character strings (one with a line control character), to give instructions to the console user. Once this picture segment is displayed at the call to ENDPIC, the program waits for a lightpen hit or function key push on key 1 or key 2. If a lightpen hit is received the line segment hit is made invisible in the next regenerations of the picture segment. If key 1 is pushed, all the lines are made visible again. If key 2 is pushed the program releases the device and terminates execution.

```
C SAMPLE PROGRAM
      DIMENSION IBUF (2000), LTC (4), FDA (3), IVIS (6), FDA (3)
      DATA LTC/3, 401, 402,-17
      DATA IVIS/1,1,1,1,1,1/
      CALL NITBUF (IBUF,2000)
      CALL NITDEV (3)

C DRAW BOX
    5 CALL BGNPIC (100)
      CALL LINE (0.2,0.5,0)
      CALL LINE (0.6,0.5, IVIS(1),1)
      CALL LINE (0.6,0.9, IVIS(2),2)
      CALL LINE (0.2,0.9, IVIS(3),3)
      CALL LINE (0.2,0.5, IVIS(4),4)
      CALL LINE (0.6,0.9, IVIS(5),5)
      CALL LINE (0.2,0.9,0)
      CALL LINE (0.6,0.5, IVIS(6),6)

C
C DISPLAY INSTRUCTIONS FOR THE USER
      CALL LINE (0.25,0.4,0)
      CALL CHAR ('THIS IS A BOX*.')
```

---

* Written and tested by J. Schwarz of Nijmegen.

```
      CALL LINE (0.15,3,0)
      CALL CHAR ('LIGHTPEN SEGMENT TO DELETE IT*FUNCTION KEY 1: X*NRESTORE
      FUNCTION KEY 2: STOP*')
      CALL ENDPIC
C
C PICTURE SEGMENT IS NOW VISIBLE
C
C WAIT FOR AN INTERRUPT
      INDEX = INWAIT (-1.0,IED,IDA,3,FDA,3)
      WRITE (t.111)(IDA(L), (1), L = 1,3)
  111 FORMAT (/ /,316)
      GO TO (10, 20, 99) INDEX
C
C LIGHTPEN HIT
   10 JLINE = IDA (3)
      IVIS (JLINE) = 0
      CALL DELPIC (100)
      GO TO 5
C
C RESTORE FUNCTION KEY 1
   20 DO 25 I = 1,6
   25 IVIS (I) = 1
      CALL DELPIC (100)
      GO TO 5
C
C STOP EXECUTION
   99 CALL RLSDEV (3)
      STOP
      END
```

Appendix B: Console Tool Information Returned

A handy reminder of what information is returned in the integer and floating point data arrays resulting from a call to INWAIT or REATOL. Floating point values for dials, tracking cross, tablet and joystick are all 0.0-1.0.

Information Returned from Console Tools

Tool Type	Integer Data	Floating Point Data
1. Refresh Clock	---	•Duration of last refresh cycle in seconds •User reaction time in seconds
2. Alphanumeric Keyboard	•... Character String in A1 format	---
3. Correlation Device (Lightpen)	•Picture Segt Name •...•Picture Elt Name Stack	•X,•Y,•Z coords of endpoint of line or character detected
100. all Dials	•last dial changed	•settings of all dials
101-110. 1-D Tools (DIALS)	---	•scalar Value
200. both 2-D tools	•last 2D tool changed	•X,•Y coords of tracking cross and •X,•Y coords of tablet
201-202. Tracking-cross or Tablet	---	•X,•Y coordinates
300. Joystick	•always '1'	•X,•Y,•Z coordinates
301. 3-D Tools	---	•X,•Y,•Z coordinates
400. Function Switches	•last key pushed •setting of all keys	---
401-416. Function Switch	•0 if not pushed and •1 if pushed	---

Appendix C: Table of Device Driver Facilities

Driver Property	Type of Driver				
	Unbuffered		Buffered		
Control Functions	Non-Inter-active	Inter-active	Non-Inter-active	Inter-active	Pseudo driver
NITDEV/RLSDEV	x	x	x	x	x
Open /Close	x	x	x	x	x
BGNPIC/ENDPIC	x*	x*	x	x	x
BGNNAM/ENDNAM			x	x	x
VISPIC/COPPIC			x	x	
DELPIC			x	x	x
REFER					x
INSERT	x	x	x	x	x
Screen-properties					
Line Types	x	x	x	x	x
Arc Generator	x	x	x	x	x
Characters	x	x	x	x	x
Intensity	x	x	x	x	x
Blinking	x	x	x	x	x
Depth Modulation	x	x	x	x	x
Color (of lines)	x	x	x	x	x
Background	x	x	x	x	x
Viewport Defn	x	x	x	x	
Viewport Tforms			x	x	
Viewport Clipping			x	x	
Tool Properties					
Clock		x		x	
A-N Keyboard		x		x	
Correlate device				x	
Audible Alarm		x		x	
1-dim Analog		x		x	
2-dim Analog		x		x	
3-dim Analog		x		x	
Function Switches		x		x	

* BGNPIC and ENDPIC must be used with unbuffered devices but they are effectively a no-operation.

PHILDIG (PHILIPS DEVICE INDEPENDENT GRAPHICS)

C. NIESSEN & J.W. ERO
N.V. Philips' Gloeilampenfabrieken, Eindhoven

INLEIDING

Bij de aanschaf van graphische randapparatuur voor computer, zoals plotters of een beeldstation, wordt door de fabrikant meestal een subroutine pakket bijgeleverd dat gebruik maakt van alle hardware eigenschappen van het betreffende randpparaat maar dat verschillend is van soortgelijke pakketten voor andere randapparaten.
Bij het overgaan op een ander randapparaat of op een andere computer moet het applicatie programma worden aangepast, hetgeen zeer kostbaar is.
In het laatste decennium is er dan ook een trend waarneembaar om z.g. "device independent" subroutine pakketten te maken zodat het applicatie programma met enige restricties onafhankelijk is van computer en beeldstations.

De noodzaak van PHILDIG

Bij Philips wordt de ontwikkeling van geïntegreerde schakelingen steeds meer geautomatiseerd. Op verschillende plaatsen in het concern worden verbeteringen in de layout van IC's aangebracht met behulp van computers en beeldstations. Ondanks de verschillen in hardware ontstond steeds meer behoefte aan de uitwisseling van applicatie software.

Teneinde kostbare conversies te voorkomen was het noodzakelijk om een device independent subroutine pakket als gereedschap ter beschikking te hebben.
Onze eisen waren:

1. Efficient in een industriële omgeving.
2. Behandeling van complexe geometrieën.
3. Uitgebreide interactie voor input en modificatie.
4. Besturing van verschillende randapparaten.

In augustus 1973 evalueerden wij bestaande pakketten zoals GINO-F en GPGS.
GINO-F, ontwikkeld door het CAD-centrum in Cambridge, was voornamelijk gericht op geheugenbuizen en plotters. Het had geen faciliteiten voor modificatie en de invoer was specifiek voor een Tektronix terminal.
GPGS, ontwikkeld door de TH Delft en de universiteiten van Nijmegen en Cambridge, had de volgende tekortkomingen:

- niet commercieel beschikbaar,
- modificatie van het plaatje was niet mogelijk,
- de invoer was niet device onafhankelijk.

Derhalve besloten wij zelf een pakket te ontwerpen en te implementeren.

Globale opzet van PHILDIG

De implementatie van PHILDIG kan nooit computer en randapparaat onafhankelijk zijn, omdat voor elk randapparaat verschillende code moet worden gegenereerd. Op het moment zijn er drie z.g. drijvers en wel één voor een "refreshed display", één voor een geheugenbuis en één voor een pseudo randapparaat. Daar het mogelijk moet zijn om meer dan één apparaat tegelijk te sturen bestaat het pakket uit een bovenlaag van randapparaat onafhankelijke routines en voor elk apparaat een aantal speciale routines met een voor dat apparaat specifiek common blok. Verder is er een common blok met variabelen van algemeen belang.
Het blokschema van het pakket is gegeven in fig. 1.

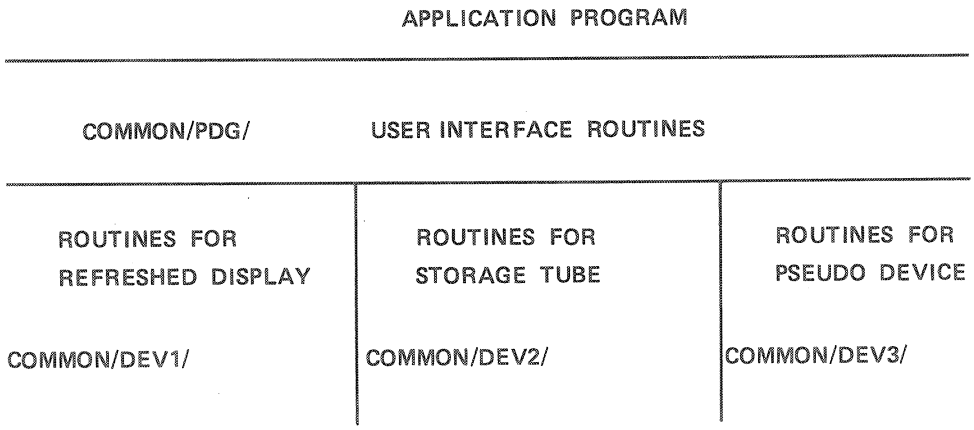

APPLICATION PROGRAM
COMMON/PDG/
USER INTERFACE ROUTINES
ROUTINES FOR REFRESHED DISPLAY
ROUTINES FOR STORAGE TUBE
ROUTINES FOR PSEUDO DEVICE
COMMON/DEV1/
COMMON/DEV2/
COMMON/DEV3/

FIG. 1

Bij het initialiseren van het pakket moet worden aangegeven welke randapparaten zullen worden gebruikt.
Devices kunnen worden vrijgegeven en opnieuw gestart. Indien een bepaalde drijver in het geheel niet nodig is kunnen de bijbehorende routines worden onderdrukt.
In fig. 2 is getekend hoe een vector aan drie randapparaten wordt doorgegeven.
Het pakket is grotendeels geschreven in Fortran behoudens enkele routines voor code generatie en interrupt afhandeling.
Het ontwikkelen van een device onafhankelijk pakket heeft drie belangrijke implicaties: compatibiliteit, standaardisatie en groter geheugengebruik.
Een applicatie programma met PHILDIG calls geschreven voor een refreshed display zal zeker niet zonder problemen lopen op een display met geheugenbuis of erger nog op een pseudo apparaat (=plot file). Omdat de apparaten nu eenmaal verschillend zijn, zullen de onderlaag routines anders reageren of zelfs afwezig zijn.
Bij het schrijven van de applicatie zal men rekening moeten houden met deze opwaartse compatibiliteit.
Een ander belangrijk punt is dat bijzondere eigenschappen van nieuwe displays minder goed of niet toepasbaar zullen zijn. M.a.w. er treedt een bevriezing op van de "state of the art".
Verder zal deze device onafhankelijke aanpak enige overhead met zich meebrengen, hetgeen leidt tot meer geheugengebruik en langere executietijd. Bij goedkoper wordende hardware en meer implementaties is dit natuurlijk geen bezwaar. Anders zouden hogere programmeertalen ook nooit ingang hebben gevonden !
Bij PHILDIG kunnen we onderscheid maken tussen de volgende soorten calls:

1. administratie
2. omlijsting
3. generatie
4. structurering
5. transformatie
6. modificatie
7. invoer
8. fout afhandeling.

```
- -
CALL
- -

SUBR        VECIA2 (X, Y)
COMMON      /PDG/
XNEX   =    X
YNEX   =    Y
IF   (DEV(1) . NE.0)    CALL DV1GEN
IF   (DEV(2) . NE.0)    CALL DV2GEN
IF   (DEV(3) . NE.0)    CALL DV3GEN
RETURN
```

SUBR DV1GEN	SUBR DV2GEN	SUBR DV3GEN
COMMON/PDG/	COMMON/PDG/	COMMON/PDG/
COMMON/DEV1/	COMMON/DEV2/	COMMON/DEV3/
TRANSFORM USER COORDINATES TO SCREEN COORDINATES		
CALL DV1COD	CALL DV2COD	CALL DV3COD
RETURN	RETURN	RETURN

SUBR DV1COD	SUBR DV2COD	SUBR DV3COD
COMMON/DEV1/	COMMON/DEV2/	COMMON/DEV3/
GENERATE CODE FOR CONNECTED DEVICE AND SEND TO BUFFER		
RETURN	RETURN	RETURN

FIG. 2

Niet al deze soorten zijn nodig.
In principe is PHILDIG eenvoudig van opzet en kunnen alleen vectoren en series karakters worden getekend.
Uiteraard kunnen allerlei utiliteits subroutines tussen PHILDIG en applicatie programma worden gemaakt.
We zullen nu de verschillende soorten calls behandelen.

Administratieve routines

Deze routines worden gebruikt om het pakket te initialiseren, eventueel om randapparaten aan- of af te koppelen en om buffers aan het pakket door te geven.
Voorbeelden:

```
---
CALL  NITDEV(P816,1)
CALL  NITDEV(T4014,2)
CALL  PICBUF(1,BUFFER,5000)
---
CALL  CSLPAK                                    (Fig. 3)
```

Omlijsting routines

In het applicatie programma wordt over het algemeen in andere eenheden gewerkt dan plot- of schermeenheden. Daarom dienen de gebruikers coördinaten te worden getransformeerd naar device-coördinaten.
De transformatievariabelen kunnen eenvoudig worden berekend uit "window" en "viewport". Dit zijn twee rechthoekige gebieden in gebruikers-, resp. device coördinaten.
Het plaatje binnen het window wordt afgebeeld op het viewport. (Zie fig. 4)
Gelijkvormige rechthoeken moeten worden opgegeven indien de schaalfactoren in X- en Y-richting gelijk moeten blijven. In het algemeen zal de transformatie worden gevolgd door een afknipproces om het plaatje binnen het viewport te houden.

Generatie routines

In PHILDIG bestaan generatie routines voor een punt, een rechte lijn en een serie karakters.

```
C     RESERVE PICTURE BUFFER
      INTEGER BUFFER (5000)
      - -
C     INITIALIZE P816 REFRESHED DISPLAY
      CALL  NITDEV (P816, 1)
C     CONNECT PICTURE BUFFER TO PHILDIG PACKAGE
      CALL  PICBUF (1, BUFFER, 5000)
      - -
C     INITIALIZE TEKTRONIX STORAGE TUBE
      CALL NITDEV (T4014, 2)
      - -
      - -
      - -
C     CLOSE PACKAGE, DISCONNECT ALL DEVICES
      CALL  CLSPAK
```

FIG. 3

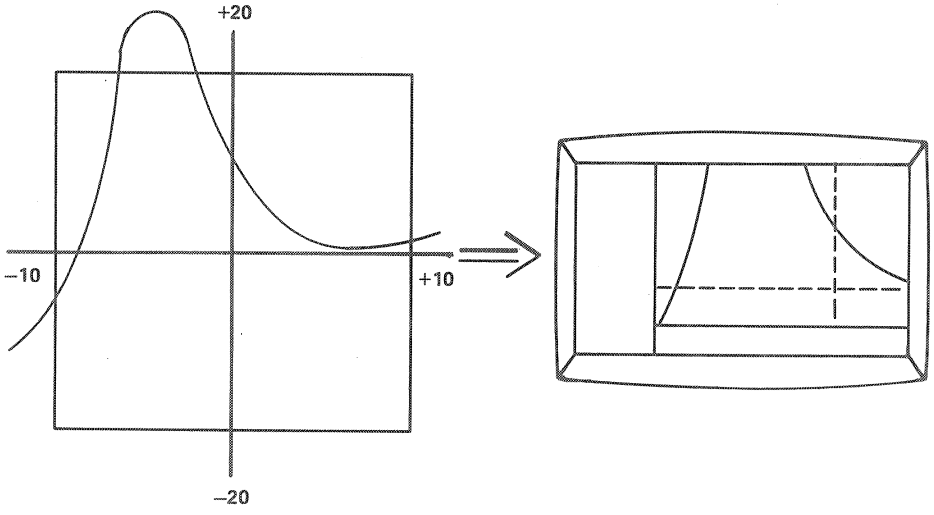


FIG. 4

Coördinaten kunnen worden opgegeven als integer of real, absoluut of relatief t.o.v. het vorige punt, en twee of driedimensionaal.
Bij driedimensionaal werken op een 2D scherm is uiteraard projectie vereist. Een bijzondere generatie call is de aanroep van een object waarbij het object fungeert als een graphische subroutine.
De namen van de subroutines worden samengesteld volgens onderstaand schema:

(PNT(punt))
(POS(positie)) (I(integer)) (A(absoluut)) (2(dim))
(VEC(vector)) (E(real)) (R(relatief)) (3(dim))

In fig. 5 worden enkele voorbeelden gegeven.

Structuur routines

Even belangrijk als de generatie van het plaatje is de mogelijkheid om het geheel of gedeeltelijk te kunnen manipuleren, d.w.z. verschuiven, verdraaien of doen verdwijnen. Dit geldt natuurlijk speciaal voor refreshed displays.
Met behulp van de structuur routines is het mogelijk om gedeelten van het plaatje tot een zogenaamd "item" samen te voegen, waarna het mogelijk is zulk een item te identificeren en te manipuleren.
Een item kan als volgt in symbolische notatie worden gedefinieerd:

```
<item>     =<B> <itembody> <E>
<B>        = CALL BGNITM (I)
<E>        = CALL ENDITM
<itembody> = empty|<element>|<element><itembody>
<element>  = <G> |<item >
<G>        =<vector>|<string>|<obj>
<vector>   = CALL POSIA2 |CALL VECIA2| ...
<string>   = CALL CHAR(. .  )
<obj>      = CALL OBJECT(n)
```

Zie ook fig. 6

```
C     GO DARK TO 10,25 AND DRAW 2 VECTORS
      CALL   POSIA2 (10,25)
      CALL   VECIA2 (10,50)
      CALL   VECIR2 (40,0)
C     DEFINE OBJECT
      CALL   BGNOBJ (1)
      CALL   VECIR2 (-20, -30)
      CALL   VECIR2 (20, 0)
      CALL   ENDOBJ
C     PLACE  OBJECT AT CURRENT BEAM POSITION
      CALL   OBJECT (1)
C     GO DARK TO 80,50 AND PLACE OBJECT AGAIN
      CALL   POSIA2 (80,50)
      CALL   OBJECT (1)
```

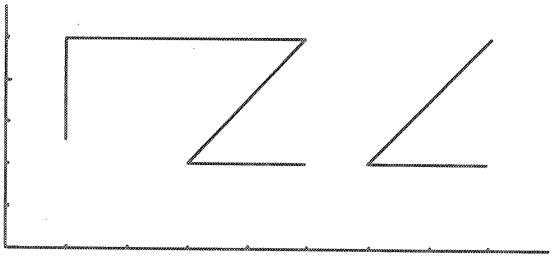


FIG. 5

```
<ITEM>        =  <B> <ITEMBODY> <E>
<B>           =   CALL BGNITM (I)
<E>           =   CALL ENDITM
<ITEMBODY>    =   EMPTY | <ELEMENT> | <ELEMENT> <ITEMBODY>
<ELEMENT>     =  <G> | <ITEM>
<G>           =  <VECTOR> | <STRING> | <OBJ>
<VECTOR>      =   CALL POSIA2  | CALL VECIA2
<STRING>      =   CALL CHAR (KARS)
<OBJ>         =   CALL OBJECT (I)
```

FIG. 6

In het voorbeeld van fig. 7 zijn de eerste twee generatie calls niet gestructureerd. De resulterende lijn kan dus niet worden geidentificeerd en dus ook niet verplaatst.
Let verder op het verschil tussen item 2 en item 1.2.

Transformatie routines

Deze routines zijn niet in het pakket geintegreerd, maar vormen een afzonderlijke utility. Dit verkleint de overhead als een gebruiker slechts zeer eenvoudige transformaties wil zoals een translatie of een rotatie over een veelvoud van 90 graden.
Verder wordt het probleem omzeild om ingevoerde schermcoördinaten via geïnverteerde matrices terug te transformeren.
De in PHILDIG genoemde transformatie routines stellen de gebruiker in staat om een homogene transformatie matrix samen te stellen en de coördinaten hiermee te transformeren.

Modificatie routines

Hierbij hebben we de volgende mogelijkheden:
Het gehele plaatje kan verdwijnen en eventueel weer verschijnen.
Items kunnen worden weggegooid, verschoven of hun hoedanigheid kan worden veranderd in die zin dat ze gaan knipperen of van kleur veranderen.
Uiteraard is dit afhankelijk van de display hardware.

De belangrijkste calls zijn:

```
CALL  DELPIC
CALL  DELIM (item)
CALL  MOVE2I(device, item, dx, dy, code)
```

Bij de laatste call moet een device nummer worden opgegeven omdat de verschuiving slechts op één randapparaat tegelijk plaats vindt. De translatie kan gebeuren onder programma besturing of van echter het scherm m.b.v. een invoer medium zoals bij voorbeeld een "tracking cross"

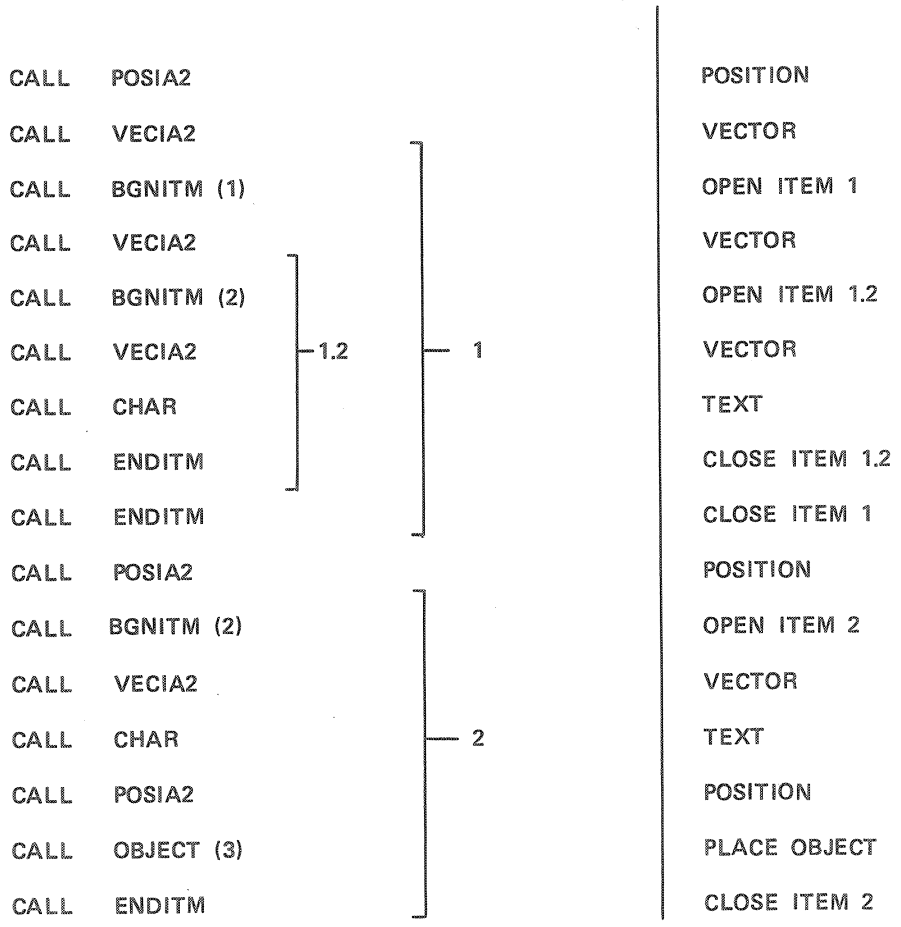

CALL POSIA2
CALL VECIA2
CALL BGNITM (1)
CALL VECIA2
CALL BGNITM (2)
CALL VECIA2
CALL CHAR
CALL ENDITM
CALL ENDITM
CALL POSIA2
CALL BGNITM (2)
CALL VECIA2
CALL CHAR
CALL POSIA2
CALL OBJECT (3)
CALL ENDITM
1.2
1
2
POSITION
VECTOR
OPEN ITEM 1
VECTOR
OPEN ITEM 1.2
VECTOR
TEXT
CLOSE ITEM 1.2
CLOSE ITEM 1
POSITION
OPEN ITEM 2
VECTOR
TEXT
POSITION
PLACE OBJECT
CLOSE ITEM 2

FIG. 7

Invoer routines

Er is veel aandacht besteed bij PHILDIG aan device onafhankelijke invoer routines. We zijn er van uitgegaan dat een applicatie programmeur niet moet worden lastig gevallen met interrupt afhandeling en verschillen tussen een "joystick" en een "trackerball". Hij is slechts geïnteresseerd in de invoer van de volgende logische media:

tekst invoer orgaan
voor invoer en correctie van tekst regels
geometrisch invoer orgaan
voor de invoer van coördinaten
draaiknop voor de invoer van een scalaire grootheid.
Identificatie orgaan om items te kunnen aanwijzen en naam en plaats geretourneerd te krijgen.
keuze invoer orgaan om een keuze nummer terug te krijgen, waarmee de loop van het programma kan worden beïnvloed.

Als physieke invoer organen fungeren lichtpen, stuurknuppel, toetsenbord en draaiknoppen. Verder worden deze invoer organen aangevuld met programmatuur. Zo heeft men voor geometrische invoer een "tracking cross" op het scherm nodig dat met lichtpen of stuurknuppel kan worden bewogen.
Om een keuze te maken moeten er één of meer menu's zijn, onderverdeeld in z.g. "boxes". Voor tekst invoer tenslotte moet een gebied op het scherm worden gereserveerd waar de ingetypte karakters zichtbaar worden gemaakt.

Voor elk randapparaat nu bestaan er twee tabellen waarin wordt aangegeven hoe de logische en physieke invoer organen met elkaar zij n gerelateerd. De eerste tabel is de toewijzingstabel en de tweede tabel heet activatietabel.
Verder is er een logisch status woord, waarin staat van welk logisch orgaan invoer verlangd wordt.
In fig. 8 is te zien hoe deze tabellen gecorreleerd zijn.
Indien logische keuze wordt verlangd, worden die physieke organen geäctiveerd die aan het logisch keuze orgaan zijn toegewezen. Als de invoer geschied is worden de physieke organen meteen weer gedeäctiveerd.

LOGDEV

IDENT.	GEO	DIAL	CHOICE	TEXT
0	0	0	1	0

PHLOAL

LIGHTPEN	1	1	0	1	0
JOYSTICK	0	1	0	0	0
KEYBOARD	0	0	0	1	1

PHLOEN

LIGHTPEN	0	0	0	1	0
JOYSTICK	0	0	0	0	0
KEYBOARD	0	0	0	1	0

FIG. 8

In het voorbeeld van fig. 9 kan keuze invoer worden gedaan door het indrukken van een knop op het toetsenbord, of door het wijzen met de lichtpen in één van de menuboxes.

In PHILDIG hebben we drie soorten invoer calls:

a. voor het definiëren van programmeerbare invoerorganen zoals menuboxen, keuzenummers voor een toetsenbord, e.d.
b. voor het toewijzen van physieke aan logische invoerorganen, indien afwijkend van standaard allocatie
c. voor de eigenlijke invoer.

Voorbeelden:

```
C     identificatie invoer
      CALL  INIDEN(UNIT,ITEM,COOR)
C     geometrische invoer
      CALL  ENBGEO (1)
      CALL  SNGEO (1,COOR)
      CALL  DSBGEO (1)
C     conditionele keuze invoer
      CALL  CNDCHO (1)
      CALL  STCHOI (1,KEUZE,STATUS)
```

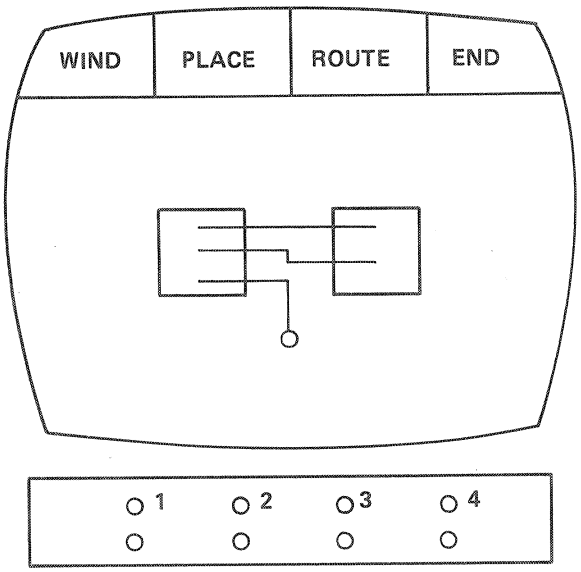
In fig. 9 wordt nog een voorbeeld van keuze invoer gegeven.

Fouten afhandeling

PHILDIG zal proberen bepaalde fouten te detecteren. Een foutcode wordt doorgegven aan een routine IGER, die de gebruiker zelf kan aanpassen aan zijn behoeften.

Implementaties

PHILDIG is tot nu toe geïmplementeerd op een P880 minicomputer en op de P1400 computer, beiden van Philips.
Op de P880-P816 (refreshed display) combinatie wordt PHILDIG gebruikt in de applicatie programma's Circuitmask en Daisy, beiden voor de layout van geïntegreerde schakelingen. Deze programma's zijn oorspronkelijk in ALGOL geschreven voor een ICL 4130 en maakten gebruik van het graphics pakket DISMAN, speciaal geschreven voor een Elliot display.




```
INTEGER   BOXES (10)
DATA      BOXES/1, 0, 1, 0, 1023, 900, 1000, 0, 4, 1 /
CALL      DEFBOX (1, BOXES)
DO        10 I = 1, 4
CALL      BGNBOX (I)
CALL      POSIA2
CALL      CHAR (TEXT)
CALL      ENDBOX
- -
CALL      INCH01 (1, K)
GOTO      (20, 30, 40, 50), K
- -
```

FIG. 9

Om redenen van efficiency werd regelmatig gebruik gemaakt van machine code programmering en bovendien werd buiten het DISMAN pakket om direct in de graphische data structuur ingegrepen.
De gegenereerde beelden zijn twee dimensionaal, bijna alle figuren zijn rechthoekig. Qua structuur moeten enerzijds de diverse componenten onderscheiden kunnen worden, anderzijds moeten de diverse lagen (maskers) te onderscheiden zijn.
De interactie geschiedt voornamelijk op component niveau en behelst:
- toevoegen van nieuwe componenten
- verplaatsen en veranderen van componenten b.v. verbindingsspoor
- verwijderen van componenten
- lengte en afstandsmetingen.

Voor al deze handelingen is het aantal keuzebepalingen nogal groot; bovendien moeten vaak coördinaten worden ingevoerd.
Interactie via tekst is minimaal.

De conversie van deze programma's naar de P880 (Fortran) heeft ons het volgende geleerd:

- In de meeste gevallen kan een DISMAN aanroep vrij eenvoudig worden vervangen door een PHILDIG functie, Alleen met betrekking tot de structuur, de items in PHILDIG,was enig re-design noodzakelijk.

- In beginsel was het mogelijk de gebruiksaanwijzing voor de ontwerper gelijk te houden. We hebben echter de flexibiliteit die PHILDIG biedt t.a.v. de interactie, gebruikt om de ontwerper meer mogelijkheden te bieden.

- De geconverteerde programma's zijn eenvoudiger, enerzijds omdat wij geen machinetaal hoefden te gebruiken, anderzijds omdat vaak een aantal statements door één PHILDIG functie te vervangen is.

- De programma's lopen ongeveer even snel op P880 en ICL 4130. Weliswaar is de P880 2 x sneller maar de woordlengte is slechts 16 bits (ICL 24) waardoor meer instructies nodig zijn.
Het totale geheugengebruik is dus groter.

THE INTERMEDIATE LANGUAGE FOR PICTURES

P.J.W. TEN HAGEN

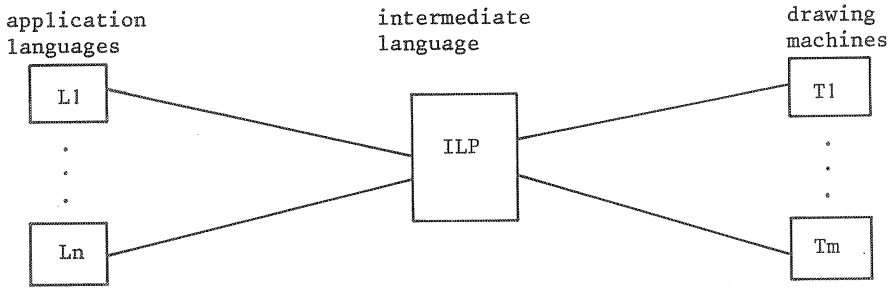
1 Introduction.

The Intermediate Language for Pictures, called ILP, has been designed at the Mathematical Centre in the course of a research project on computer graphics. The project is based on the design and implementation of an autonomous operating system for computer graphics. This graphics system can operate in a mini-computer installation which paticipates in a larger system as a satellite.

During design and implementation of the system new methods are introduced and tested. The ultimate system should contribute to computer science in the sense that new facilities are realised.

To reach both goals the ILP is an important basic means. To show this we will outline the overall structure of the system thereby emphasising the function of the ILP. A more complete overview can be found in [1].

A computer program for a graphics application is running in the central computer. It controls the graphics facilities in the satelite through a communication link.



The ILP is applied in the following ways:

- All pictures in the satellite are represented as ILP programs.

- The communication between (the user at) the satellite and the application program is in terms of ILP programs.

- The high level graphical language of the application program is obtained by embedding the ILP in an existing high level general purpose programming language.

- Pictures can be stored, retrieved and classified as ILP programs.

- The various drawing devices are logically connected by defining a conversion between ILP and device code. This is also true for input devices. The important consequence of the last application is that for the first time full symmetry between input and output can be obtained.

One might have noticed that these applications of the ILP require that the ILP is a datastructure rather than a programming language. However, since the meaning of such a datastructure is also fixed (It represents a picture), one may equally well consider these datastructures as programs, which, when executed (interpreted) produce the picture as output.

For the designers of the system the ILP plays a key role both for designing as well as a means of communication. The ILP gets the full language treatment in the sense that a complete syntax and semantics for it are given. In can be represented in symbolic form just as any other programming language. Compilers and interpreters for it are developed. All these activities are at the same time contributions to the graphics system. We will illustrate this with a few examples:

- There is a one to one correspondence between the syntax and the datastructure representation (of a picture). For instance syntactic non-terminals corespond to certain (groups of) data types. This syntax will therefore define both the form of the internal representation as well as the extensions to high level languages.

- The semantics of the language in the first place completely define the machine independent part of pictures and secondly the effect of the ILP on an idealised drawing machine. Each physical device is connected by defining a correspondence to this idealised machine.

- Any module of the system can be tested seperately by applying it to symbolic ILP programs. In this way the cooperation with other modules can be simulated and the result is available in readable form.

The most important achievement however is that a uniform concept is used throughout the system.

During the definition of the language, the designers have found that the ILP is a means to isolate and characterise the essentials of computer graphics. It is also possible to compare the complexity of various operations on graphical data. One simply expresses these operations in tersm of transforming ILP programs.

2. The basic structure of the ILP.

The ILP is described in [1]. Here we only present a simplified version, which we hope gives a good impression of the language.

The interpretation of a collection of data can result in two types of actions performed by an interpreter, namely external actions and changes of the state of the interpreter. According to this scheme one can devide data entities in action specifiers or for short actions, and state specifiers which we will call attributes. Furthermore one needs an operator to connect attributes with actions, expressing the fact that the actions should be carried out in the state as described by the attributes.

The basic construction for the picture data has the form:

WITH A DRAW P ,

where: A denotes a collection attributes, P denotes a set of actions called a picture and WITH...DRAW... denotes the connection operator. The construction as a whole is again a picture. For both pictures and attributes there exists primitives and complex constructs built from more primitive entities by means of composition.

The ILP is a low level language in the sense that apart from a number of language primitives only a few very elementary means of composition are provided. These composition rules serve two main purposes:

- Compact representation of data. To this end a subroutine-like construct exists for both pictures and attributes. Multiple used pictures or attributes have to be stored only once and can be referenced as often

as needed. Furthermore the WITH...DRAW... construct, which can be nested (by applying the subroutine mechanism, for instance) allows us to minimise the number of state descriptions. The possibility to specify A common state onlly once, although during interpretation intermediate states may be different is refined by giving the opportunity to create, locally, partial exceptions to the current state.

- Structuring the data in such a way that the manipulations anticipated are best suited. To this end optimisation by the representation function can be overruled, so that data collections can easily be split or changed. The possibility to insert empty cells in both picture lists and attribute lists allows us to specify the skeleton for a datastructure in which the actual values can be supplied later. The composition rules simple as they are can build arbitrary graph structures.

The two basic entities picture and attribute have the following syntax:

```
<picture>:      <picture element> | <pname> |
                '{' <pictures> '}' |
                WITH <attribute> DRAW <picture> ;
<attribute>:    [ABS | REL] <basic attribute> ;
<basic attribute>: <attribute class> | <aname> |
                '(' <attributes> ')' ;
```

The non-composite constructs are picture element and attribute class. They will be discussed later. Pname and aname are names of pictures and attributes respectively. An ILP program consists of a set of named pictures and a set of named attributes. A named picture is called a rootpicture if its name is external (global), it is called subpicture otherwise. If, in the sequel, we use the word subpicture we also mean rootpicture, but the reverse is not true. A named attribute is called an attribute pack.

```
<rootpicture>: PICT <pname> <picture> ;
<subpicture>:  SUBPICT <pname> <picture> ;
<attribute pack>: ATTR <aname> <attribute> ;
```

An interpretation of an ILP program is started in a root picture. The picture that constitues the body of a subpicture can be a sequence of picture's between brackets:

<pictures>: <picture> | <picture> ';' <pictures> ;

Similar for attributes:

<attributes>: <attribute> | <attribute> ';' <attributes>

The subroutine and bracketing mechanism allow us to have (directed) attribute graphs and picture graphs. The WITH...DRAW operator combines picture and attribute graph into one picture graph. The picture graph concept is used in the sequel to give the basic semantic rules. The initial node of the graph is the external call to a rootpicture. The direct descendants of that node are the pictures that constitute the body of the rootpicture. Picture elements are terminal nodes. The non-terminal nodes are the other alternatives of the syntax rule for pictures. The WITH...DRAW nodes always have two descendants namely the attribute and the picture. An attribute node only contains attributes as descendants

Pname is a subpicture call. At this moment recursive calls are not allowed, because both pictures and attributes do not contain any form of condition setting. This and other constructs like assignment and parameters are left out of the language for two reasons:

- ILP programs will be mainly used as objects to be generated rather than executed.

- The ILP will be embedded in high level languages where all these constructs are already present.

Stated in terms of the picture graph the basic semantic actions are the following. The interpreter visits the nodes of the graph in preorder. Each time an attribute node is encountered, the attribute (which may be an entire subgraph) is interpreted, which results in a so-called state description. This new state is mixed with the current state into a new current state. Now the picture node of the same parent node is interpreted in this state. Upon return to the parent node the original state is restored. It follows that the interpreter should be capable of maintaining a stack of state descriptions. Each time a picture element is encountered the element at hand is transformed according to the current state. The resulting element is next converted into a sequence of machine dependent actions.

According to this scheme further semantics specify the following items:

- How attributes are converted into state descriptions.

- How two states are mixed, or for that matter, how attributes are mixed into new attributes.

- How attributes transform picture elements.

- How picture elements will be converted into drawing machine instructions.

Traversing the graph from the root to an endnode (picture element) the interpreter may have encountered several attribute nodes. In principle their effect is accumulated. The attributes of child nodes have priority over those of parent nodes, e.g., they are applied first and moreover, they specify whether the parent attributes will be applied at all. The latter is controlled by the tags ABS and REL respectively which may be prefixed to a list of attributes. REL means apply parent- (or because of the accumulation, current-) attribute also. ABS means replace current attribute. This mechanism implements the concept of specifying a common state with local exceptions (ABS) or adjustments (REL). The common state will be specified close to the root, the exceptions closer to the endnodes.

In the next paragraph we will investigate the traversal process in more detail.

3. The interpretation of attributes and pictures.

In this paragraph we will specify the four remaining semantic items. The syntax rule for non-composite attributes is:

```
<attribute class>:  <transformation> |
                    <detection> |
                    <style> |
                    <pen> |
                    <control> ;
```

Attributes are divided into so-called attribute classes. Attribute classes are mutually unrelated. This means that in the process of mixing attributes only attributees from the same class are involved. The result of mixing attribute primitives from a single class is called an attribute class value (or class value for short). Each primitive attribute itself is a particular instance of a class value. The converse however, is not true, i.e. not every class value can be expressed by means of one attribute primitive of that class.

A state description is a list of class values which contains at most one class value for each attribute class.

We will now define how an attribute graph is combined into one state description. In this process we will first explain how primitive values are combined into class values, next how class values are combined into state descriptions, and last how state descriptions are combined into one state description.

The attribute graph that has to become a state description is elaborated as follows. First all references (anames) to attribute packs are replaced by the corresponding attribute pack. The only type of nesting that remains is bracket nesting. The combining operation starts bottom upwards. All attribute lists that contain no sublists between brackets are converted into a state description. This process can be described in the following steps:

-- The primitive attributes are sorted class-wise without disturbing the suborder in each class, e.g.:
(a1,a3,b1,c,b2,a2) => (a1,a3,a2,b1,b2,c).

-- Next the attributes of one class are combined (concatenated) into one class value, e.g.:
((a1*a3*a2), (b1*b2), (c)) .

According to the syntax, each primitive attribute can have at most one ABS/REL prefix. These prefixes are applied as follows:

```
A * REL a * B = A * a * B .
A * ABS a * B =     a * B .
```

Here A and B denote a sequence of attribute primitives of the same class. "*" denotes the mixing operator. The resulting value is further treated as a class value. In case of a single primitive, the primitive value will also be treated as a class value. Any prefix is preserved and becomes a prefix of the class value, e.g.:

```
ABS a => ABS (a) .
```

In this way a list of primitive attributes results in a state description.

Going up one level we find a list consisting of state descriptions mixed with primitive attribute values. All adjacent primitive attribute values are also combined into a state description. We now have on the lowest level a series of state descriptions only. According to the syntax and the special rule for single primitive values, a class value can have at most one individual prefix. The state description as a whole can also have one prefix. A state prefix overrules

an individual prefix. Only when a state description has no direct prefix, the individual prefixes are valid, e.g.:

```
ABS(REL a) = (ABS a) .
```

What remains to be specified is how a sequence of state descriptions is mixed into one state description. This process, which very much looks like the one for primitive values, takes the following steps:

-- The state prefixes are distributed over the individual members in the sense just described, so that at most one prefix per class value remains.

-- The class values are arranged class-wise, without disturbing their ordering (the ordering induced by the order of the state descriptions).

-- The class values are combined into one new class value as follows:

```
A * ABS a * B = a * B .
A * REL a * B = A * a * B .
```

Here a denotes a class value, A and B denote sequences of class values. Note that a composite class value can be replaced by an ABS class value which consists of one primitive value only.

By repeatedly applying the combination rule for state descriptions, each time going up one level, finally one state description is obtained.

The mixing of a state description belonging to a WITH...DRAW construct with the current state description is a special case of the mixing process defined above. The sequence of state descriptions contains at most two elements.

The restoration of the original state after a WITH...DRAW construct has been elaborated, can be described as follows: Upon entry an ABS state desription is generated with a copy of each class value of the current state that will be changed as a result of mixing with the new state. This state description is mixed with the current state description upon return.

During elaboration of a picture element the current state is effectuated as follows:

-- Part of the state description is fed into the drawing machine as control information.

-- The remainder is applied by changing the picture element at hand.

The ideal situation would be that the state description

can be fed into the drawing device, so that altering the picture element is completely carried out by hardware. In practice we have to put one or more virtual machines between the ideal one and the physical device.

The application of a state description to a picture element takes two major steps. First the picture element performs a state selection. Next the state finally obtained is effectuated.

The general form of a picture element is:

"type" <attributematches> "type values"

This is the primitive form of the general construction

WITH A DRAW P ,

preceded by some type specification. The attribute matches control the state selection. A single attribute match is a binairy value. Each attribute class has a corresponding attribute match. A state selector contains one attribute match value for each class. These values select a partial state description from the current state. The partial state is completed to a full state by adding default values for each missing class. The default values may depend upon the type of the picture element. This mechanism is the ultimate consequence of providing a common state with individual exceptions.

The five attribute matches and their corresponding attribute classes are:

match	class	comment
TO/BY	<empty>	Absolute or incremental mode.
TF/~TF	<transformation>	
VS/~VS	<pen>	Penfunctions or invisible move.
DT/~DT	<detection>	Selectable by pointing or not.
ST/~ST	<style>	
<empty>	<control>	Special controls cannot be ignored.

The effect of the individual class values on picture elements will be specified in the section on attribute primitives. In general two ways of description are needed. In the first place one can define the effect for each type of picture element. In that case the effect is expressed by giving a sequence of equivalent picture elements, e.g.:

```
WITH a DRAW pe  <=>
     {pe1; pe2; ... ; pen} .
```

The second way consists of specifying the corresponding control sequence for the idealised drawing machine. The effect on this machine is for the time being described in an informal way.

If a class value is still a composite value of several primitive attributes, these values are applied in "textual order", with the one closest to the root last.

The effect of picture elements themselves on the idealised machine is defined in a way similar to attribute classes. Some can be given in terms of other (more) primitive elements. Some have to be defined more informally. Picture elements never are composite values.

4. Picture elements.

Picture elements are syntactically described by:

```
<picture element>:
     <coordinate type> |
     <curve> | <text> |
     <library> | NIL ;
```

We will now discuss the various picture elements by specifying the meaning of each type-tag and its corresponding type-value.

4.1 Coordinate type.

The type-tags for which the type-values must be coordinates are:

```
<type>:   POINT | LINE | CONTOUR ;
```

They occur in the following syntax rules:

```
<coordinate type>:
     <type> <attribute matches> '{'<coordinates>'}' |
     <type> <coordinate> ;
<coordinate>: PP | PO |
     <attribute matches> <dimensional value> ;
```

PP and PO are special coordinates whose values are calculated during the elaboration of an ILP-program. PP and PO are mnemonics for penposition and penorigin respectively. They represent the penposition in user coordinates at certain moments during elaboration. PP is calculated at the

beginning of a row of coordinates and remains constant, regardless of changes in the penposition during the elaboration of that row. PO is the value of the pen coordinates at the beginning of the smallest picture that encloses the picture element, in which PO is referenced. Recall that each picture element is nested within one or more pictures. PO allows us among other things to specify subpictures that leave the penposition where it was at the start, by adding a picture element like

```
POINT  VI PO
```

as the last element to the subpicture. Note the difference with

```
POINT  VI PP !
```

Before the elaboration of a root picture starts, PP and PO have as value the origin of the user coordinate system.

The primitive action embodied by a picture element with coordinates as type-values can be described as follows. First of all the row of coordinates specifies a series of positions. The positions are found in either of two ways:

-- In the TO-state (attribute match penrel has value TO), the coordinates are absolute values with respect to the current origin.

-- In the BY-state, the coordinates are taken as offsets from the current penposition (incremental mode). This series of positions is the same for all types.

The type-tag is used to specify a "polygon", that contains these positions as vertices. The first and last vertex of the polygon however are different for different types. Let the penposition at the beginning of the action to draw the picture element at hand be X. Let the series of positions be represented by c1, c2, ..., cn. Then the polygon to be drawn is:

-- In case of type POINT: c1-c2-...-cn .

-- In case of type LINE: X-c1-...-cn .

-- In case of type CONTOUR: c1-c2-...cn-c1 .

The possibility X-c1-...-cn-X, can be obtained by adding the special coordinate denoted as PP to the head of the row of coordinates of type CONTOUR. This produces a closed polygon with the original penposition as the first (and last) value, e.g.:

```
BY CONTOUR {PP;(0, 1);(1, 0);(0, -1)}
```

specifies a square that begins and ends in the penposition. If we replace BY by TO in this example, we also get a closed polygon which starts and ends in the penposition. However, we cannot say what the shape will be until we know the penposition.

We now have established which positions the pen will visit while a POINT, LINE or CONTOUR is elaborated. What is actually drawn, and what route is actually taken going from one position to the next, depends on the type-tag and the attributes. The attribute match VI and its negation VI specify whether anything will be drawn at all. In the state VI the route is followed as a sequence of invisible moves. In the state VI the pen-functions that define the colour, linewidth, etc. are applied. In a similar way the attribute match ST or ST specifies whether the current style-functions or the default style-function for that type will be applied.

There is, apart from the initial vertex, a second fundamental difference between a row of POINT coordinates and a row of LINe coordinates. For LINEs the route between succesive positions defined by the coordinates is always a straight line, which will be drawn according to the current style-functions. The route between POINT positions is undefined. For this reason it is impossible to apply any line style-function to the route between these points. It is not defined in which order the positions have to be visited, with the exception of the last one. Hence the only style-functions for POINTs are those which specify any symbol centered around the positions. On the other hand, it is possible to specify a line style for LINEs which shows the positions as points. In that case the initial penposition is always included. It is also possible to superimpose "point" styles on line styles for LINEs. With respect to style-functions the CONTOUR behaves in a LINE-like manner.

4.2 TEXT.

Objects with type-tag TEXT enable us to produce texts as part of a picture. The syntax rules are:

```
<text>:   TEXT <attribute matches> '{'<strings>'}' |
     TEXT <string> ;
<string>:
     <attribute matches> <proper string> ;
<proper string>:
     [ESC <escl> <esc2> '"'tokens'"' |
     '"'tokens'"' ;
<token>: [escl] <basic token> ;
<basic token>:
     <char> | <digit> |
     <esc2> <char> | <esc2> <value> ;
```

The type-value of TEXT is a row of strings. Each element in the row may have its own private escape characters. A single character is a special case of a string, which in turn is a special case of a row of strings.

Characters are grouped in alphabets of 256 tokens. We assume that there are at least 64 printable characters in the system. With the help of 2 escape characters it is possible to specify all 256 tokens. tokens can also be specified by giving their index to the alphabet as a numerical value preceded by esc2. Change of alphabet is possible by means of attributes. In principle an unlimited set of alphabets can be used in an ILP-program.

The attribute matches for TEXT control the same attribute classes as the other picture elements do. Whether an attribute from a given class applies to a TEXT element or not, depends on the definition of the attribute itself. So the important rule is that attribute matches control the attribute classes whereas individual attributes control picture elements. Attributes which are applied exclusively to TEXT elements are called "typographic-functions" and are a subclass of the style-functions.

An important aspect of TEXT values is the way they are positioned; since nowhere in a TEXT value, a coordinate can be specified, the position must be deduced from the current environment. The environment contains as a result of a special typographic function, a so-called page format. The page format defines the effect of tokens like "carriage return", "line feed", "formfeed", "space" and "tabulation". At any time during execution there are two pages in existence. One page starts (with the upper left corner in the current origin, the other starts at the penposition valid at the time a primitive TEXT element begins.

The attribute match BY/TO specifies whether the pen page or the origin page has to be filled or continued respectively. The origin page can be (re)defined under control of the CURRENT attribute. In fact the origin page shifts along with the origin.

So far we have encountered primitives with explicit values. The remaining two types are generators of values of a constant type.

4.3 CURVE.

A CURVE-value consists of a row of curvedescriptions which are functionspecifications. Each function generates a row of coordinates. The effect of a CURVE-value can now be defined as the effect of a LINE with the same attributematches and with the generated coordinates as type values.

The generated coordinates are either two- or three-dimensional curves. Three-dimensional curves need not necessarily to be planar. CURVE-values are functions which produce a list of coordinates. The coordinates are further treated as a row of coordinates of type LINE. The functions are devided in two types: parameter functions and non-parameter functions.

For a parameterfunction one must specify an interval (2D or 3D) and two or three functions of one variable. The coordinates produced are of types (x(t), y(t) [, z(t)]), where t steps through the interval. The stepsize can be calculated by the function itself, can depend on a given device or be given as one of the arguments.

Non-parameterfunctions or systemfunctions are collected in a system function library. Each function has its own name and parameterformat. Parameters may be number values as well as other primitives. The functions only produce coordinates and have no side effects whatsoever on drawing device or environment attributes). The parameters are handed over to the system routine without any modification by current attributes.

The curve concept and especially the function part of it is subject to future extensions and refinement.

4.4 LIBRARY.

LIBRARY is followed by a row of names of external subpictures. The effect of a LIBRARY-call is that a sequence of primitives is produced. The way these primitives are produced inside the LIBRARY-function is not specified in the ILP. For instance, it might involve an interactive session. In general, however, a LIBRARY subpicture is produced in a previous session as an ordinary subpicture.

LIBRARY eleements as opposed to subpictures allow us to consider part of the picture program to consist of "sym-

bols". To all possible operations on picture programs LIBRARY elements are indivisible primitive units.

5. Basic attributes.

The attribute classes are the alternatives of the syntax rule:

```
<attribute class>:  <control> |
     <transformation> |
     <pen> |
     <style> |
     <detection> ;
```

The order in which they are listed here is also the order in which they are applied to picture elements.

per attribute class one must specify the following items:

- How primitive values are mixed.
- How class values are mixed.
- How class values are applied to picture elements.

We will now briefly discuss examples of attributes for each class. For the complete description so far present in ILP we refer to [2].

5.1 Control.

Control contains all attributes that cannot be classified elsewhere. It contains for instance a subclass called machine typical.

```
<control>: <general control> |
     <machine typical> ;
<machine typical>: MACHINE <proper string> ;
<general control>: <clear> |
     <operator call> |
     <feed> | <frame> | <stop> |
     <material> ;
```

Aclass value simply consists of a list of the alternatives given here. They are applied by feeding them into the drawing device as control information, e.g. they do not

transform picture elements in any way.

We intend to keep the class control as small as possible.

5.2 Transformations.

```
<transformation>:
        <position> |
        <matrix> |
        <window> |
        <viewport> ;
```

Transformations are applied to so called dimensional values (coordinates). The effect of transformations is that of coordinate transformations well known in computer graphics.

Position matches the imaginairy origin of the new picture with a position in the current coordinate space. This position is either the existing origin or the penposition (CURRENT). Position can dynamically be converted to a matrix transformation.

Matrix is the general transformation, written as a full homogenuous matrix. One may also build up such a matrix value with the help of sub-matrices like rotation,scaling and projection.

Window has two aspects. It performs clipping along the border of the window. In combination with viewport it defines a (matrix) transformation to screen coordinates. Mixing of windows means intersecting them. Clipping has priority over matrix transformation.

The class value for transformations consists of a matrix and a window. For the moment only rectangular windows are allowed. The combination of two matrix window pairs is in general not possible (rotation). In the absence of rotation combination can be described as:

```
(M1,W1) * (M2,W2) = (M1*M2,W1*M2(W2)).
Here M(W) means the rectangle transformed by M.
M1*M2 implies matrix multiplication.
W1*W2 implies intersection of windows.
```

In the other case one must retain both pairs. So in general a transformation class value consists of a sequence of matrix window pairs. In the case of a generalised window one may again apply full concatenation.

5.3 Penfunctions.

Penfuntions specify all items associated with the hard-copy or screen materials.

<pen>: <thick> | <colour> | <ink> ;

The various alternatives control the thickness of lines, colour and type of ink and also the intensity of the electron beam. If for a given device the attribute is meaningless it simply is ignored.This attribute class can be devided in a number of independent subclasses (of which three are listed). A class value consists of a series of subclass values (one per class). Mixing means superposition or replacement of corresponding subclasses. In the case of an ABS prefixed class mixing means complete replacement.

5.4 Style.

Stylefunctions as opposed to transformations are more directly involved with the actual state of the drawing machine. Stylefunctions describe what kind of lines and characters (and in a future extension of the language what kind of shades and greyscales) are to be produced by the drawing machine. The description is as machine independent as possible. In view of the enormous variety of current and future drawing machines the style-function package has to be incomplete and hence extendable.

The two classes of style-functions that exist so far, e.g. line styles and typographic styles are mutually unrelated. Line styles are applied to coordinate values, typographic styles to strings. We will now first discuss line styles.

<line style>: <period> MAP <value> <reset> ;
<period>: PERIOD '('<period description>')' ;

The line pattern attribute can produce a large variety of dotted and dashed lines. The definition of such a pattern goes in two steps:

Period definition:

Period is a basic pattern which is repeatedly produced going along the line.

```
<period description>:
          <dash> | <dash> ´,´ <gap> |
          <dash> ´,´ <gap> ´,´ <dash> ;
<dash>:        DOT | <value> ;
<gap>:         <value> ;
<reset>:  RESET | CONTINUE ;
```

The period is defined on a straight line piece of 100 units in length: Hence dash1 + gap1 + dash2 + gap2 = 100. Therefore gap2 always is omitted (gap2 = 100 - (dash1 + gap1 + dash2). If dash2 is omitted, its value is 100 - (dash1 + gap1). In that case gap2 = 0. In the same manner gap1 can be omitted (=> dash2 = gap2 = 0). If dash has value DOT. A point is produced on the spot with has a length of 0 units with respect to the period.

Examples:
PER (100) => solid line.
PER (DOT) => one point at the beginning of each period.
PER (0, 100) => blank (invisible) line.
PER (50) => dashed line with gaps equal to dashes.
PER(25, 50) => dashed line with gaps equal to dashes. It starts, however, with a half dash.

Map definition: Map first of all defines the length of the period in coordinate distance units. A period of the given length is rolled along the line. The period reset is a binary value telling whether a period has to be restarted or continued when a new coordinate value of the line is encountered.

The default line style for a LINE type line is

```
PER(100)PM 1 CONTINUE,
```

e.g. a solid line. Both styles are produced also when the style bit of the attribute match has value ST. The line style

```
PER(DOT)PM "some large value" RESET
```

will produce a dot at each coordinate value encountered. This illustrates that style can convert LINE´s into POINT´s. The typographic style is in fact nothing else than a means to specify a given characterset from the sets available.

```
<typographic>: <fount> |
          <size> |
          <italic> |
          <boldness> ;
```

As we have seen before characters are grouped in sets

of 128 tokens, called a basic set. A fount attribute now specifies a basic set. The other attributes can now be used to produce a variant of such a basic set (by giving a new size, italic, position or boldness).

A basic set can contain any kind of symbols, up to complete pictures. In the ILP they will, however, be considered as characters, and can therefore not be manipulated otherwise. It is clear that this attribute suffices to be able to specify an unlimited collection of characters.

5.5 Detection.

The detection attribute provides the primitives for interactive work with pictures. Basically its effect is that it isolates parts of the picture from the rest. The isolated part may be of type subpicture but that is merely coincidental.

To understand really what happens we must use the graph structure description. For each node in the graph (which is not an endnode) the detection mechanism tells us whether that node is detectable or not. For each endnode in the graph, the detection mechanism tells us whether that leave is sensible or not. An endnode is sensible if the attribute match has the value DT. We make a distinction between sensible and detectable, because the detecting mechanism is activated by means of an external action. The primitives are the only objects that can be made subject to external actions like pointing with a lightpen or a tracker ball, or even by textual specification of an ILP primitive. If a primitive picture is not sensible, any external pointing mechanism will have no effect when it points at that primitive. If a node is detectable, it also defines a parent node or itself as the detectant.

The detection attribute has the following syntax:

<detection>: DETECT <dname> <proper string> ;

The detection mchanisme contains more than one detector. Each detector has its own name (dname). There is also a common detector which has no name. Switching from one detector to another is possible by external action which consists of selecting a new name or the common detector. Whenever a node is detected the string (if any) that is attached to it can be returned to the user. This provides him with an facility for identification of the various detection points. Sensibility is the same for all named and unnamed detectors. The sensibility thus in a very crude way devides the picture primitives in two groups. The not sensible ones are not detectable, the sensible ones are. However, if the

parent node of the sensible primitive is undetectable, we have isolated one simple primitive. We can enforce the undetectability of all parent nodes by external action. We only need to select a special named detector which does not actually occur in the ILP-program. If the primitive is part of a subpicture, both the detectbility of the parent node and the detectant can be different for each instance of the subpicture. What remains to be specified is how the detection attribute works on non terminal nodes.

The value of the detection attribute at each node can be any of the following four:

AD	absolute detectable
AU	absolute undetectable
RD	relative detectable
RU	relative undetectable

The absolute/relative value originates from the prefix that can be attached to all attributes. The effect of the four values is as follows: AD means the node is detectable and it defines the current node as the detectant. AU means the node is not detectable, the detectant is undefined. This means, for instance, that if in fig 1 det1 = AD and det2 = AU, than node1 can never be selected by pointing at a primitive child of node2. RD means, the node is detectable. The detectant is the detectant of the parent node if it is defined, otherwise the node itself becomes the detectant. In other words: if the parent node is undetectable, then RD = AD. RU means, the node is undetectable. However, the detectant remains defined. If a child node is set RD, then it will pick up the detectant that is still defined. So the value R (relative) in both cases (D and U) passes on the detectant, if it exists. This gives a very usefull result if one applies this attribute to subpictures. The detectability may vary for each instance of the subpicture. If the subpicturebody is RD, then it will identify each detectable call. If the call is not detectable it will identify the subpicture itself. If the subpicturebody is RU, it will also identify each detectable call, but if the call is not detectable, the subpicture itself will remain undetectable. We now can also classify the values of the attribute match, namely: DT <=> AU and DT <=> RD for the primitive concerned.

So far we have not related the pointing action to visibility aspects. Apart from sensible, each primitive can also be visible or invisible. Many hardware pointing devices (e.g. lightpens) identify sensibility and visibility. We have deliberately chosen for the separate concepts, because we can give a meaningfull interpretation for each combination of (in)visibility and (undetectability. For instance, in order to change an invisible move, one must first identify it. This concludes the description of the detection

primitive, inside the ILP.

On implementation of the interactive satellite facilities we will introduce a number of external actions that are built upon the existing primitives. To give the reader an indication of the type of actions we have in mind, we give a few examples.

One can specify several detectors, even conditionally, so that the basic detection structure of picture can dynamically vary. This gives a possibility to introduce unions and intersections for sets of primitives.

Each detectable node can identify itself by means of a string. In this way one can identify the path through the tree from primitive to detectant.

6. Conclusion.

We tried to show that we designed a general but simple scheme in which a large variety of language constructs can be fitted. It must be admitted that not everything we wanted in the language can be modelled this way. We will give two examples.

In order to be able to consider the two-dimensional coordinates as a subset of the three-dimensional ones we had to add a construct called subspace. The subspace mainly behaves like an attribute. However, we also wanted consistency between various attributes and between attributes and picture elements with respect to dimension. This could be obtained only by putting constraints on ILP programs that contain a dimensional mix. In this case the constraints are completely expressed by syntax, so that they do not have any influence on dynamic aspects.

The scaling (by transformations) of line styled picture elements produces the same line style pattern although the line itself changes. To remove this restriction we would have to maintain a dynamically changing unit of measure which may be inspected by any attribute.

Many other cases (generalised windows, for instance) have been reserved for future extensions of the language.

Our main goal now is to gain experience with it as quickly as possible by applying it in the way mentioned in the introduction.

REFERENCES

[1] P.J.W. ten Hagen, P. Klint, H. Noot and T. Hagen, Design of an Interactive Graphics System. MC Report IW36 1975

[2] T. Hagen, P.J.W. ten Hagen, P. Klint and H Noot, The ILP, Intermediate Language for Pictures, MC Report, to appear.

%

SATELLITE GRAPHICS

J. VAN DEN BOS
Informatica / Computer Graphics
Faculteit Wiskunde en Natuurwetenschappen
Universiteit Nijmegen

ABSTRACT

In this paper we describe the development of computer graphics via satellite computers coupled to a large host computer. We study in particular systems where parts of the application may run either in the host machine or in the satellite or in both, so-called configurable programs. Two major implementations, CAGES of the University of North Carolina, and ICOPS of Brown University receive special attention. The advantages and drawbacks of both systems are discussed. An alternative approach which is in the proposal stage at the University of Nijmegen, utilizing a tightly coupled network of microcomputers in lieu of a satellite is described. This microcomputer complex serves as a special purpose vehicle for graphics through a pre-planned distribution of the graphics workload. Present possibilities and future developments of such a complex are discussed.

1. INTRODUCTION

There are in principle three direct methods to attach a Computer Graphics display device to a computer (see Fig. 1). From a configurational point of view the simplest way would be to have a graphic terminal connected directly to the computer, in the manner of an ordinary I/O peripheral (see Fig. 1a). In addition to performing all the computational work for the application program the computer would create display lists which on subsequent interpretation (scanning) would generate the displayed images on the screen of the graphics terminal by means of direct I/O commands. For a refresh-type terminal this scanning followed by the I/O transfer would have to be repeated at least every 1/30th of a second to prevent flicker of the displayed image. On top of this the computer would have to take care of keeping the display list up-to-date, and of general display buffer management. It would also have to handle interrupts (attentions) from a diversity of graphics input tools such as alphanumeric keyboard, light pen, tablet, function keys, joy stick, mouse, tracker ball, control dials and other esoteric interaction devices. For a moderately complicated graphics application these demands already require a reasonably powerful computer.

In the arrangement shown in Fig. 1b we equip the display station with some local intelligence, the Display Processor Unit or DPU. In this way we move the job of scanning the display list to the graphics terminal. The DPU interpretes the display list through a direct port (DMA) to the memory of the computer. Frequently the DPU is also capable of the initial handling of user or program generated attentions.

Due to the high bandwidth required interference from the DPU with the CPU could become significant. This is especially so for a raster scan display device where this interference is so high as to virtually rule out this configuration.

We advance a little further (see Fig. 1c) by equipping the DPU with its own memory, often but ambiguously called a display buffer. In this situation we have two copies of the display list, one which resides in main memory, the other one which sits in the display memory. In terms of direct graphics support all the central processor does in this case is the construction and maintenance of the original of the display list, and buffer management in general. After each refresh cycle the CPU may (but does not have to) transfer part or all of a display list to the display station.

The construction of this display list in no way interferes with the scanning proces of the DPU, thereby ensuring a smooth and flicker-free display. While the DPU is regenerating the image the CPU can do useful work in the context of the graphics application under consideration or possibly for some unrelated job running simultaneously with the graphics program.

In the situation just sketched it was implied that in principle the computer may be used in a multiprogramming mode. In actual practice the load on the CPU for many graphical applications involving large amounts of "number crunching" or dynamic display (animation) turns out to be such that either the graphics program or other programs are degraded more or less seriously. We face the paradoxical situation that in most cases graphics can co-exist very nicely with other programs, yet in some cases the peak requirements are such that the computer has to be dedicated for longer or shorter time to the graphics program. Since the latter cannot be guaranteed on a multiprogramming system we have to accept the fact that applications such as real-time motions cannot be run while other jobs are present.

A partial solution to this problem is to augment the graphics terminal by a small to medium sized computer which is connected via a broad band connection to a large multiprogrammed host computer (see Fig. 2). If we select for this computer one of the advanced mini- or midicomputers, possibly equipped with floating point arithmetic and disk storage, a number of tasks such as windowing, clipping, keyboard entry, lightpen tracking, coordinate transformations, attention handling and of course regeneration can be done locally. At the same time the host computer will be available for complex computations or data manipulations and for those programs which require a large data base.

In this set-up graphics terminal with local computer function as a *graphics satellite* of the host. This solution offers a local monopoly as well as host multiprogramming. In addition it is possible that the satellite is used to run graphics programs in stand-alone mode in which the connection with the large computer is left unused. The advantages of this alternative in the satellite system are clear: many applications do not need the host any more - thereby improving throughout and response time for all users concerned.

2. PROGRAMMING THE SATELLITE-HOST SYSTEM

Having a graphics satellite offers several alternatives to the graphics program. Part of the program may be run on the host machine and other parts,

e.g. those parts which interact with the user on a short-response time basis may be advantageously put on the satellite computer.

Usually the distribution of program segments is not a prerogative of the applications programmer. The implementor has fixed this choice in the implementation. For the programmer the advantage is that the very existence of the graphics satellite station is transparent. It protects him from getting involved in multiple languages, operating system idiosyncrasies, communication protocol and hardware details. It also provides a measure of portability, since changing the satellite system will probably not necessitate alternation of the source code of the program. In general the applications part of his program will be running on the host machine while low-level graphics support routines will reside in the satellite. Information exchange between the two program components will be accomplished by means of message exchange on the basis of some graphics (or more general) protocol. This way of doing satellite graphics is called *distributed processing* with *static division of labor.*

A typical example of this situation occurs under GPGS [1], a satellite-host graphics support system which allows the user to use the satellite in stand-along mode as well as in a configuration where the application is running on the host routines, while the satellite runs the device-dependent routines such as picture compilers (which convert device-independent display code into device-dependent display lists), display buffer management, regeneration, basic attention handling and keyboard entry. In the latter situation communication between host and satellite is maintained by messages transmitted across a high-speed link. For the programmer there is no way to directly access this level nor influence the distribution of routines over the processors, even if in certain situations this turns out to be non-optimal. In particular, short of running the complete application on the satellite, there is no way to put any application routines on the satellite. In several instances, e.g. when these routines directly deal with man-machine interaction, this may be a severe disadvantage.

Another drawback of this static division of tasks is that the satellite hardware can never be fully exploited by the applications programmer.

MOULTON and CORMAN [2] report on a system consisting of a host computer supporting several dissimilar graphic satellite processors in which remote interaction programs may be specified in the host through an Interaction Language. Semantically this language refers to a virtual graphics

satellite called the Programmable Graphics Processor (PGP). This PGP is equipped with general purpose registers, string registers, and a stack, as well as a large number of special purpose display registers (e.g. LPX and LPY indicate the position of a lightpen hit, etc.). PGP consists of a picture processor and an interaction processor. The latter one communicates with the former one through the display registers and so-called graphic channels. These channels provide access to picture segments, which may be manipulated by assigning values to certain attributes such as color, line texture, intensity etc. of one or all graphic channels. The PGP's run as interpreters in the satellites. Satellites are down-loaded from the host with the pre-compiled PGP interaction programs. After interpretation the interaction routines are activated by attention generated by the user.

This work may be considered as a follow-up of the Interactive Control Tables of COTTON [3] which were also used to specify, in the host, interaction programs to be run on the satellite, thus ensuring a rapid response to graphics interaction demands.

A more powerful and flexible situation occurs with *dynamic division of labor*. In this case the programmer is given some decision-making power with respect to the placement of program segments - his program then becomes *configurable*. This could be utilized to enhance the efficient execution of his program, to imporve response time, or to adapt his program configuration to the total load of the host computer.

In Section 3 we discuss a system that allows configurable programs at compile-time, and in Section 4 a system which allows complete dynamic configurability, i.e. at execution-time. Finally in Section 5 a multi microcomputer system for satellite graphics is proposed.

3. CAGES

Dynamic division of labor may be done at compile-time, at binding-time, load time, or at execution-time. In this order we find increasing flexibility but also increasing complexity in support. If the division is done at compile-time, then in many cases the user has to program in two languages, one for the host and one for the satellite.

The University of North Carolina's system CAGES [4] (for Configurable Applications for Graphics Employing Satellites) is a graphics programming system which allows configurability at compile-time (actually at

preprocessor-time) to programs written in an extended subset of PL/I, which has special language constructs, mostly of declarative type, which describe the desired program configuration. Program procedures and data can be easily reassigned from one computer to the other, thus modifying the division of labor for a given program. CAGES actually simulates a dual processor system with a single main memory, through the following services (quoted from ref. 4):

"(1) Inter-computer subroutine CALLs and RETURNs. A subroutine call executed on one computer and targeted to a subroutine resident on the other computer is known as a *remote procedure call*. It is supported just as one would expect. A message containing the subroutine's name and parameters is sent to the computer where the subroutine is to be executed. Following its execution, a message containing results (modified parameters) is sent back, and the calling routine continues its execution.

(2) Inter-computer CONDITIONs. A SIGNAL statement or interrupts on one computer raising a CONDITION for which there is an enable ON-BLOCK in the other computer is much like a parameterless remote procedure. It is supported in essentially the same manner as a remote call.

(3) GLOBAL data references. Variables with EXTERNAL scope which may be referenced from subroutines in both computers are known as GLOBAL variables. If, when referenced, a GLOBAL variable is not located in the memory of the referencing computer, the reference is said to be a *remote reference*. The CAGES system handles such references by obtaining the needed data from the remote computer and placing a local copy of it in the memory of the referencing computer. The program is than allowed to access this copy."

A block diagram of the complete system is given in Fig. 3. The CAGES system is implemented on an IBM 360/75 host computer connected via a 9600 baud line to a PDP 11/45 satellite with a Vector General display device. In this system the intelligence is distributed over three rather than two processors, because the display device has a quite powerful DPU itself.

Application programs are written in the PL/I subset, which is roughly the PL/C (Cornell PL/I compiler [5]) subset. At this point the programmer still assumes his program will run on a single computer. Next he specifies the desired division of labor by indicating the assignment of remote

procedures and remote data (GLOBALS). He does this with the language constructs in the extension of the subset. The complete source code goes to a preprocessor.

The preprocessor creates two files of source code, one for each computer. These source files are processed by PL/I compilers with code emittors for the host and the satellite. Remote procedures are replaced (see Fig. 4) by stubs (dummy procedures) which call the remote procedure by sending a formatted message, containing the entry point and the parameters, to the other machine. On return from the remote procedure call the stub will update any variables affected by returned results, and finally the stub will return control to the next statement in the (local) calling program. Global variables are not assigned to one computer but move between computers depending on which computer references the global (see Fig. 5). Initially, globals exist nowhere. Storage will be allocated for each one of them by the computer which first references them. Somewhere before this happens in the program the preprocessor has inserted a run-time system call to acquire, if necessary, one or more global variables from the remote computer. Checks have to be made to ascertain that the required copy is the most recently changed one. Furthermore, since the allocation of globals is done dynamically some garbage collector has to free the storage used by abandoned globals.

From the description it should be clear that CAGES is not a graphics package in the usual sense. Indeed, any graphics subroutines or other graphics support used by the application program appears to the CAGES system as part of the application. Thus CAGES does not require or provide for any particular type of graphic data structure. In fact, the CAGES system is equally useful to non-graphics programs for which distribution is desired.

The decision to allow remote data has been an expensive one. In the first place it is not trivial to describe data items in a computer independent way; furthermore checking the validity of a global variable, allocation and garbage collection add substantially to the overhead. In fact it turns out that 80% of the code in the run-time system is for managing the globals. As we shall shortly see, these arguments have been sufficient reason for ICOPS not to allow remote data.

4. ICOPS

Although CAGES is a great stride forward from static division of labor, it still suffers from the fact that the distribution remains fixed after compile-time. In particular, configuring a program with the objective to adapt it to the external load on the host computer, which may vary from instant to instant, is not possible. This may mean a suddenly poor response time for a program that initially was configured correctly, due to time-dependent factors which are beyond control of the programmer. In practice it could mean that a user would have several different configurations of one and the same program. A particular version would then be started according to a calculated "guesstimate" by the user.

From the theory of binding we know that maximal flexibility is achieved in a situation where binding is postponed as long as possible. In the context of distributed processing this means that ideally the binding of procedures and data to a processor should be done at execution time, so that the user may reconfigure his *running* program as the environment requires.

The Interconnected Processor System (ICOPS) of Brown University [6] is a system which allows this completely dynamic distribution of programs.

It is implemented for an IBM 360/67 host computer, running under the CP/CMS operating system, and attached (via its multiplexor channel) to a graphics satellite (see Fig. 6), consisting of two microprogrammable Digital Scientific META 4 computers, the META 4A and the META 4B. The META 4a has been microprogrammed to serve as a general purpose (satellite) processor. It has therefore been equipped with an IBM/360 like instruction set extended with special instructions for list processing, data handling and communication. The META 4B functions as a programmable display processor driving a Vector General display device. This processor serves in effect as a layer around the already reasonably powerful Vector General display. This extra level offers a display instruction set tailored to the wishes of the systems programmer. The satellite system is also equipped with an in-house designed high-speed parallel matrix processor (SIMALE) to perform homogeneous coordinate transformations, windowing and clipping of graphic data for up to 1500 vectors during a single refresh cycle. The META 4A runs under a layered operating system, the lowest level of which is implemented in firmware. Both META 4's share the same main store.

ICOPS allows procedures to be reallocated between host computer and

satellite at run-time. As a consequence the system supports inter-processor calls and returns. Since procedures move dynamically special provisions have to be taken to trap calls to procedures absent on the processor from which the call originated.
One can distinghuish three major components in ICOPS:

1. A high-level language with a compiler capable of generating code for host as well as satellite. It should contain constructs which facilitate the movement of procedures from processor to processor. Originally LSD (language for System Development [7]) was used. This is a powerful PL/I like language which allows the programmer to get as close to the hardware as he wants yet at the same time providing him with all the advantages of a high-level language. Due to implementation problems LSD has eventually been dropped in favor of ALGOL W [8]. In addition to object module output the compiler for the language also produces symbol tables to go with each object module.

2. A link-edit time preprocessing system which creates information which later on allows procedure reallocation. It basically performs three functions:
   a. It assigns initial placement attributes to the procedures and marks the modules which are eligible to be moved between processors (a movable procedure is called ICPable)
   b. All ICPable procedures are embedded in a dummy procedure in which a static switch variable indicates where the called procedure resides, and the call to the procedure is replaced by the switch selected real CALL or a call to the run-time system in case the procedure resides on the other processor (cf. CAGES).
   c. It places information from the compiler symbol table (type, length, etc.) in the dummy procedure for all static variables and parameters.

3. A run-time system which allows the user to manage and control procedure (re-) allocation. This system contains a Monitor which can be entered by user call or via an attention signal from a graphics terminal. This monitor allows the user to request procedure reallocation, statistics and trace facilities. In addition the run-time system takes care of procedure call resolution (setting the switch in the dummy module),

procedure movement, for which parameter passing (using the symbol table) and communication is performed, as well as marking the old copy of a procedure inactive. Furthermore the system offers facilities for performance measurement as a function of allocation, and debugging services through a built-in breakpoint mechanism.

ICOPS does not allow Global variables. The implementation of this type of variable was deemed undesirable because of all the problems related with it. In effect the bookkeeping connected with Globals easily surpasses the work connected with ICPable procedures.

Various other restrictions relate to multitasking, which is not supported, pointers (which can only be relative to some point in a structure which is passed as a parameter of an ICPable procedure), and recursive activations of moving procedures.

We conclude that with a system such as ICOPS a knowledgeable user is not only able to adapt his program configuration to a sudden upsurge in the load on the host computer but he may also fine-tune his procedure allocation by giving him complete control over the dynamic allocation of procedures marked movable, aided by a sensible interpretation of the collected statistics as well as tracing information.

It is clear that systems like CAGES and ICOPS do not aim at the one-shot program or the beginning programmer, but are meant to be used for complicated (graphics) production programs for which optimized computer usage is a primary requirement. For simple programs the run-time overhead exacts a price which, in all likelihood, users may not be willing to pay.

In the meantime the interpretation of the statistics collected during distribution of programs has turned out to be far from easy. Both with CAGES and ICOPS experiments are being done which use a distribution model to guide the movement of procedures [9,10].

5. A MULTI MICROCOMPUTERS SYSTEM FOR GRAPHICS

In this Section we discuss a hardware configuration which not only supports stand-alone and satellite graphics but which in addition refines the concept of distributed processing inherent in the division of labor across main computer and satellite.

The approach taken departs from the general purpose computing idea, because it proposes a configuration tailored to a particular application

area which however may be as wide as computer graphics.

Briefly the method consists of analyzing the particular area for common functions in processing as well as in the way input and output are handled. These common functions are then implemented by fast microcomputers. In this way each microcomputer becomes a special purpose computer, in effect becoming a black box labeled with a certain function.

More particularly, for graphics, we distinguish a number of tasks that have to be handled for each display terminal, such as attention handling, display buffer control and update functions, picture compilation (display list creation), keyboard entry and response, clipping against a viewport, (viewport) transformations, and conversion from logical (device-independent or user) coordinates to screen coordinates. All these functions may be termed device-dependent. They normally (see e.g. [1]) are put in a group of modules that handle the device-dependent functions, the device-drivers.

On the device-independent level we discern such functions as display buffer management, possibly conversion from graphics program calls to an intermediate representation, system transformations (translation, scaling, rotation, shearing, perspective), clipping against a window, and hidden-line removal.

The first configuration we suggest is called the terminal processors system and is depicted in Fig. 7. It consists of a central micro- or minicomputer (/μC0) handling the device-independent functions. This central computer is connected to as many mini- or microcomputers (/μCi, i>0) as there are graphics terminals. Each of these 'terminal' microcomputers implements, possibly with the help of multiple microprocessors, the device-dependent functions for that particular terminal. Note that each terminal has its own microcomputer, even if there are terminals which are alike. The central computer routes device-dependent function requests to the appropriate microcomputer. Although in a first approximation not strictly necessary, functions could be performed in parallel, with the central machine taking care of synchronization. By tailoring the instruction set of the microcomputers to the terminal requirements it is hoped that they will handle their tasks more efficiently than one satellite computer. Furthermore it is possible to make all terminals look alike to the programmer by wrapping the terminal into a layer of code on each microcomputer which presents a uniform interface to the central microcomputer. This would enable the software to implement the idea of a virtual idealized

terminal, a concept on which e.g. GPGS and IG [11] as well as PGP (see Section 2) were built. Moreover, the addition of a microcomputer to a simple graphics terminal (see e.g. [12] may significantly enhance the capabilities of the terminal, in effect making it more intelligent.

An alternative to this configuration, the dedicated function processors system, is presented in Fig. 8. Here all terminals are handled by the central microcomputer, which allows among other things identical terminals to be handled by reentrant modules. Special graphics functions would be handled by dedicated microprocessors, tailored to their application.

A third approach is displayed in Fig. 9. It basically is a combination of the two former configurations, and therefore has the advantages of both. A disadvantage of this system is that due to the greater complexity of it, the controlling software residing in the central microcomputer is going to be significantly more complicated.

Which of the three approaches to adopt is a matter of research. It seems that in terms of the complexity of control software either the terminal processor system or the dedicated function processors deserve to be studied first. During this initial phase the starting point will be static division of labor. Later on when a system has been shown to be successful one might study the influence of movable data and procedures on the configuration. This may involve some kind of analysis of the procedures in order to determine which microcomputer best suits the procedure.

Although the proposals outlined above seem rather static, it is possible, given a more advanced state of the technology, to introduce a great amount of flexibility by using microprogrammable microcomputers. In this way we may alter the function of a dedicated microprocessor as reguired, catering to more generality as well creating the possibility to switch in a spare microcomputer when one of the others fails. Actually these microprocessors already exist in the form of the so-called bit-slice processors. It is our opinion however that combining bit-slice processors into a microcomputer with a user-defined instruction set is something far beyond reach of the experienced programmer or systems designer. We feel that, short of introducing hardware engineers, the only real possibility lies in vertically microprogrammable microcomputers. The ultimate flexibility will be obtained when the computers can be dynamically microprogrammed, meaning that we can change instruction sets (which may be stored on a disk or in the main memory of the central computer) while running. Progress in this area will be greatly aided by high-level microprogramming languages.

6. CONCLUSION

Satellite graphics has been proved a valid concept. Several implementations of systems based on distributed processing with static division of labor exist, although the division of labor has usually been decided on an ad-hoc basis and then only for graphics systems software. Dynamic division of labor has so far only been demonstrated in the CAGES and ICOPS systems. A systematic experiment seems to be needed to determine whether the optimal division of labor will change drastically across a wide range of graphics applications or whether dynamic division is only a one-time experimental vehicle that points out an optimal situation for practically all graphics applications. If variable divisions of labor really turns out to be important then it is almost without question that microcomputers, and especially the dynamically microprogrammable variety, will play a large role in this kind of systems.

REFERENCES

1. GPGS, *A device independent general purpose graphics system for stand-alone and satellite graphics*, L.C. CARUTHERS, J. VAN DEN BOS, Nijmegen University. A. VAN DAM, Brown University, Computer Graphics (Siggraph-ACM) 11,2(1977), pp. 112-119.

2. S.D. MOULTON & P.J. CORMAN, *Remote programmability of graphic interactions in a host/satellite configuration*, Computer Graphics (Siggraph-ACM) 10,2(1976), pp. 204-211.

3. I.W. COTTON, *Languages for attention handling*, Proc. Comp. Graph. 70 Syposium, Brunel University, 1970.

4. G. HAMLIN & J.D. Foley, *Configurable applications for graphics employing satellites* (CAGES), Computer Graphics (Siggraph-ACM) 9,2(1975), pp. 9-19.

5. R.W. CONWAY & T.R. WILCOX, *Design and implementation of a diagnostic compiler for PL/I*, CACM 16,2(1973), pp. 169-179.

6. A. VAN DAM & G.M. Stabler & R.J. HARRINGTON, *Intelligent satellites for interactive graphics*. Proc. IEEE 62,4(1974), pp. 483-492.
   G.M. STABLER, *A system for interconnected processing*, Ph.D. Thesis, Brown University, Providence, R.I. , 1974.

7. D. BERGERON, J. GANNON, D. SHECTER, F. TOMPA & A. VAN DAM, *Systems programming languages, Advances in computers 12,* Academic Press, New York 1972.

8. N. WIRTH & C.A.R. HOARE, *A contribution to the development of Algol,* CACM 9,6(1966), pp. 413-431.
H.R. BAUER, S. BECKER, S.L. GRAHAM & E. SATTERTHWAITE, *Algol W. Comp. science report 110,* Stanford University 1669.

9. G. HAMLIN, *Configurable applications for satellite graphics,* Computer Graphics (Siggraph-ACM) 10,2(1976), pp. 196-203.

10. J. MICHEL & A. VAN DAM, *Experience with distributed processing on a host/satellite graphics system,* Computer Graphics (Siggraph-ACM) 10,2(1976), pp. 190-195.

11. J.F. BLINN & A.C. GOODRICH, *The internal design of the IG Routines, an interactive graphics system for a large timesharing environment,* Computer Graphics (Siggraph-ACM) 10,2(1976), pp. 229-234.

12. R.G. KELLNER & L.D. MAAS, *A developmental system for microcomputer based intelligent graphics terminals,* Computer Graphics (Siggraph-ACM) 10,2(1976), pp. 139-142.

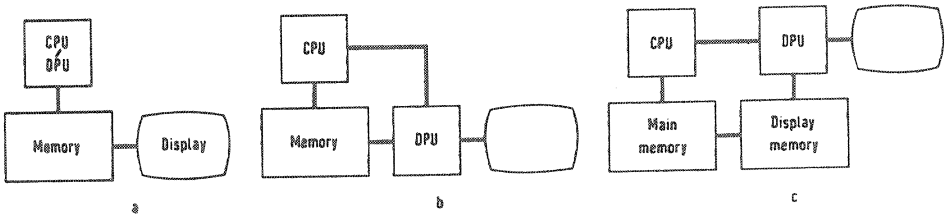


Figure 1

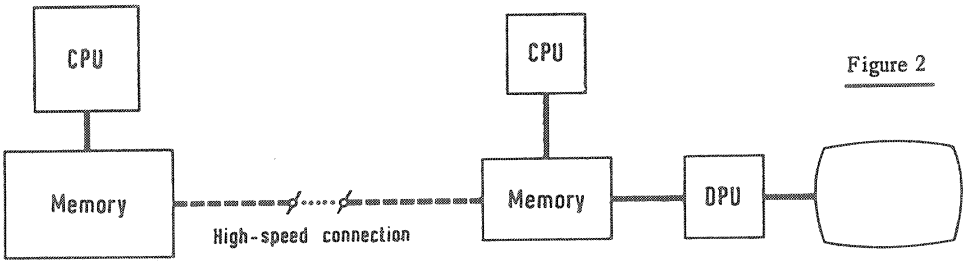


Figure 2

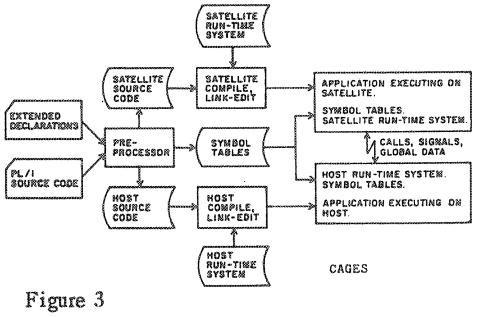


Figure 3

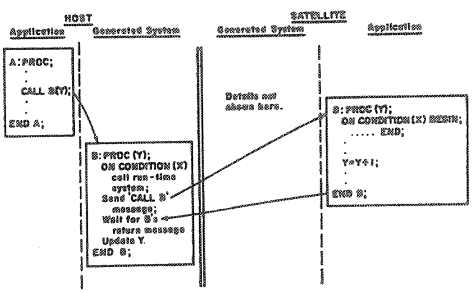


Figure 4

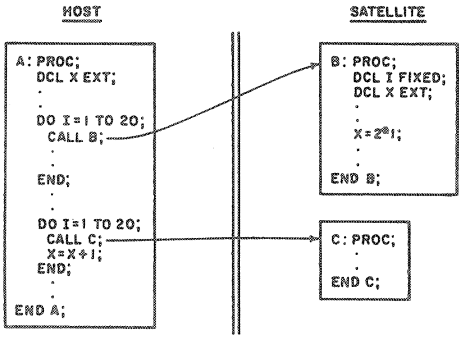


Figure 5

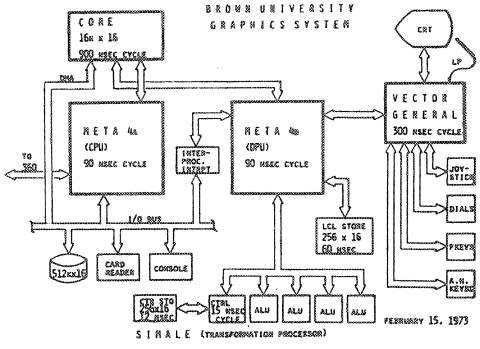


Figure 6

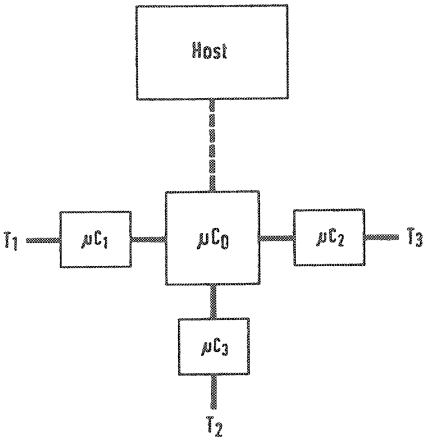


Figure 7

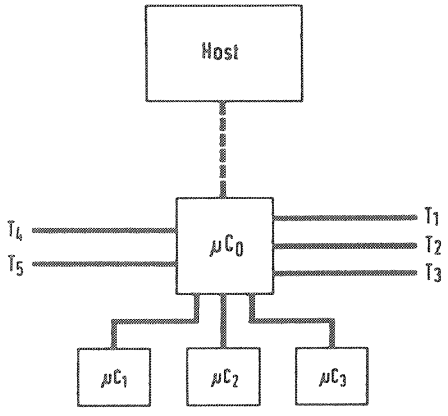


Figure 8

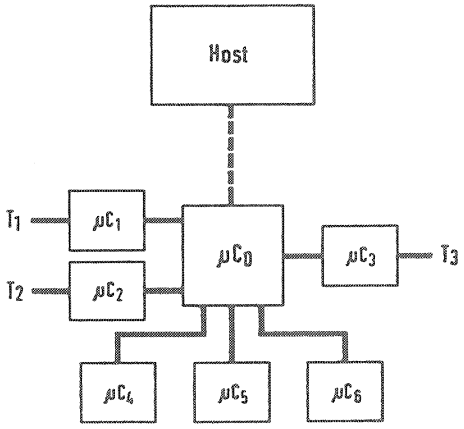


Figure 9

COMPUTER AIDED DESIGN OF MECHANICAL COMPONENTS

H. RANKERS
T.H. Delft

In het kader van dit informatica-colloquium 1976/1977 COMPUTER GRAPHICS zal ik proberen uiteen te zetten, waarom en hoe wij de computer inclusief de beeldbuis in ons werk betrekken. Als werktuigbouwkundig ingenieur en hoogleraar voor bedrijfsmechanisatie en leer der mechanismen zal ik mij beperken tot het ontwerpen van mechanische componenten "met ondersteuning door de elektronische rekenmachine".

De koppeling van de twee vakgebieden -leer der mechanismen en bedrijfsmechanisatie- schept de mogelijkheid theoretische beschouwingen en alle tot nu toe verworven inzichten in opbouw en gedrag van mechanismen toe te passen in het kader van de bedrijfsmechanisatie. De gehele opzet is daarom gericht op toepassingen in de praktijk.

Tot een belangrijk deel van de aktiviteiten in de bedrijfsmechanisatie behoort het uitwerken van een cyclus van het weg-tijd-diagram voor alle in een machine benodigde bewegingen van materiaal en/of van de gereedschappen. Deze deelaktiviteit kan erg werkintensief zijn als op eerder genomen beslissingen moet worden teruggekomen en een iteratieve optimalisatie moet worden nagestreefd.

De volgende deelaktiviteit is dan het zoeken van een energie-omzetter met een dusdanige programmering, dat hij als functie-generator in staat is de gewenste beweging goed of nog aanvaardbaar te approximeren.

In termen van de systeemtechniek is de functie-generator een black box. De gewenste beweging wordt de doelfunctie (goal function, Zielfunktion) van het synthese-proces genoemd. In het kader van deze lezing zal ik mij beperken tot dit ene onderwerp: de synthese van mechanismen.

De algemene theorie van de synthese van mechanismen is beschreven in de dissertatie van de spreker [1].

Volgens deze algemene theorie is een systematische fout in de synthese van eenvoudige mechanismen alleen dan te voorkomen, als eerst wordt bekeken welk type mechanisme uit de totale verzameling als functiegenerator überhaupt voor de benadering van de doelfunctie in aanmerking komt. Pas als dit systematisch vastgesteld is lijkt het zinvol de afmetingen van het gekozen type te bepalen.

Als methode voor het systematisch selecteren werd in [1] voorgesteld de mechanismenfuncties door middel van Fourier-coëfficiënten te karakteriseren en met de Fourier-coëfficiënten van de doelfunctie te vergelijken.

De grote bewerkelijkheid van het synthese-proces was voor de werkgroep aanleiding voor het uitwerken van een computerprogramma TADSOL (= TYPE AND DIMENSION SYNTHESIS OF LINK MECHANISMS) [2].

De resultaten, waarover hier gesproken wordt, werden voltooid in de THD-werkgroep CADOM (= Computer Aided Design Of Mechanisms) en hebben als doel de elektronische rekenmachine toe te passen bij de synthese van mechanismen.

De resultaten zijn onmisbaar bij de opleiding van ingenieurs die aldus in staat zijn concrete opgaven te realiseren.

Als men over een weg-tijd-diagram spreekt moet men ook over de synthese van mechanismen spreken.

Naast het vermelden van de resultaten in mijn hoofdvakcolleges in het 4e en 5e studiejaar wordt getracht de studenten meer inzicht en ervaring te geven door hen bij het werk te betrekken.

Dit gebeurt door het verstrekken van concrete en relatief weinig tijd vragende opdrachten om bepaalde bouwstenen uit te werken en te testen of het systeem aan problemen uit de praktijk te toetsen.

Zodoende is overeenkomstig harde afspraken in ons projectboek een eigen programma-bibliotheek tot stand gekomen.

Hierin staat een verzameling van geteste bouwstenen voor de invoer, uitvoer en de numerieke behandeling van gegevens [3]. Dit wordt naar behoefte uitgebreid en eigenlijk dagelijks met veel profijt bij het schrijven van hoofdprogramma's en subprogramma's toegepast.

Een uitgebreide, op de gebruiker gerichte beschrijving van de synthese volgens het TADSOL-programma is in [2] gegeven, zodat ik mij hier

kan beperken tot het nagaan van de belangrijkste contouren.
(In de laatste 8 minuten van mijn spreektijd zou ik U dan graag een videotape tonen, die kortgeleden door de audio visuele dienst van de Technische Hogeschool Delft is opgenomen.)

De Fourier-representatie van een periodieke functie

Elke functie $f(\alpha)$, die in het interval $0 \leq \alpha \leq 2\pi$ continu is, d.w.z. elke in de techniek voorkomende periodieke functie kan men in een Fourier reeks ontwikkelen en in twee gedaanten weergeven: de c-ϕ-representatie en de cos-sin-representatie.

$$f(\alpha) = C_0 + \sum_{n=1}^{\infty} C_n \sin(n\alpha + \phi_n)$$

$$= A_0 + \sum_{n=1}^{\infty} (A_n \cos n\alpha + B_n \sin n\alpha).$$

De mechanismenfuncties en de doelfunctie worden vervangen door deze Fourier-representaties. Deze vervanging schept de mogelijkheid zowel de mechanismenfuncties als de doelfuncties ondubbelzinnig te karakteriseren en onderling te vergelijken. Mechanismen met overeenkomstige Fourier-ontwikkelingen hebben overeenkomstige eigenschappen en zijn in kinematisch opzicht tenminste bij benadering gelijkwaardig [4]. Dank zij het feit, dat de hogere orde Fourier-coëfficiënten van mechanismenfuncties met aanvaardbare kinematisch-dynamische eigenschappen al gauw tot nul convergeren, is een goede vervanging van de periodieke functies te bereiken met het berekenen van minder dan 24 Fourier-coëfficiënten.

Het karakteriseren van mechanismen

Uitgebreid onderzoek van MEYER ZUR CAPELLEN (o.m. [5]) en sommige anderen (zie lit. in [2]) heeft aangetoond, dat het noemen van de eerste zes Fourier-coëfficiënten voldoende is om een mechanisme d.m.v. de Fourier-coëfficiënten van zijn nulde orde overdrachtsfunctie te laten beschrijven.

Omdat de periode van 2π vereist is, moet de Fourier-coëfficiënt van de eerste orde, d.w.z. C_1 altijd aanwezig zijn. De belangstelling richt zich daarom op het al dan niet aanwezig zijn van de hogere orde Fourier-coëfficiënten van de orde twee t/m zes. Is N het aantal voorkomende significante Fourier-coëfficiënten, dan zijn er K mogelijke combinaties van

Fourier-coëfficiënten met

$$K = \binom{5}{N-1}$$

zie ook tabel 1.

De type-synthese

Het aantal N significante Fourier-coëfficiënten C_n dient als het eerste criterium van de selectie van mechanismen.

De combinatie van de Fourier-coëfficiënt C_1 met N-1 andere significante coëfficiënten dient als het tweede criterium van de selectie van mechanismen.

Vanaf het derde criterium wordt de cos-sin-representatie bij de selectie van het mechanismentype betrokken.

De dimensie-synthese

De berekening van de parameters van een gekozen type mechanisme noemen wij dimensie-synthese. Omdat de Fourier-coëfficiënten volledig bepaald zijn door de dimensies van het mechanisme, stelt de inversie ons in staat de parameters als functie van diverse Fourier-coëfficiënten te formuleren. Deze inversie is niet altijd even gemakkelijk uit te voeren. In het geval van het kruk-sleuf-mechanisme van de tweede soort met een ronddraaiende uitgaande beweging zijn de twee parameters bepaald door [6]

$$i = A_0$$

$$\lambda = \frac{d}{b} = \sum k_m \cdot \left(\frac{A_1}{A_0}\right)^m, \quad m = 1,3,5,7,9$$

met $k_1 = 0.999\ 958\ 40$
$k_3 = 0.124\ 448\ 42$
$k_5 = 0.002\ 166\ 93$
$k_7 = 0.002\ 262\ 12$
$K_9 = 0.002\ 086\ 52.$

Het overschot aan informatie

De zes Fourier-coëfficiënten bevatten meer informatie dan voor het

Spectrum van Fourier-coefficienten						naam verzameling	nummers van de hogere harmonischen	aantal comb.
C_1	—					SINGLE	——	1
C_1	C_K					DOUBLE	$2 \leq K \leq 6$	5
C_1	C_K , C_L					TRIPLE	$2 \leq K < L \leq 6$	10
C_1	C_K, C_L, C_M					QDRPLE	$2 \leq K < L < M \leq 6$	10
C_1	C_K, C_L, C_M, C_N					QNTPLE	$2 \leq K < L < M < N \leq 6$	5
C_1	C_2	C_3	C_4	C_5	C_6	SXTPLE	———	1
TH Delft Bedrijfsmech.	Ordenings-criteria 1^e en 2^e soort							Rankers tab. 1

bepalen van de mechanismenparameters m.b.v. de waarde van de Fourier-coëfficiënten benodigd is. Voor het bepalen van de acht paramaters van het mechanisme T004 zijn b.v. slechts drie Fourier-coëfficiënten nodig. De overige Fourier-coëfficiënten van hogere orde, die mogelijk in de doelfunctie nog aanwezig zijn, doen niet mee aan de berekening van de parameters van het mechanisme. Daarom moet er wel worden nagegaan of deze met het oog op de gewenste goede benadering inderdaad buiten beschouwing mogen worden gelaten. Voor verdere details verwijs ik U naar de publikatie [2].

Voorbeeld voor de synthese

Fig. 1 toont een willekeurig gegeven doelfunctie $s(\alpha^*)$. Fig. 2 geeft het daarbij behorende spectrum van de Fourier-coëfficiënten C_n in percenten van de eerste coëfficiënt C_1.

De syntheseprocedure maakt duidelijk, dat de mechanismen T002, T004 en T008, die in de mechanismen-catalogus [7] nader zijn toegelicht, voor de approximatie van de in fig. 1 gegeven doelfunctie in aanmerking komen. Ik toon U hier de benadering van de doelfunctie d.m.v. het mechanisme T008, die fig. 3, en het mechanisme zelf op schaal onder vermelding van de referentiestand en van de startpositie en de draairichting voor de benadering van de doelfunctie, zie fig. 4.

De afmetingen van het mechanisme zijn gegeven door de output listing

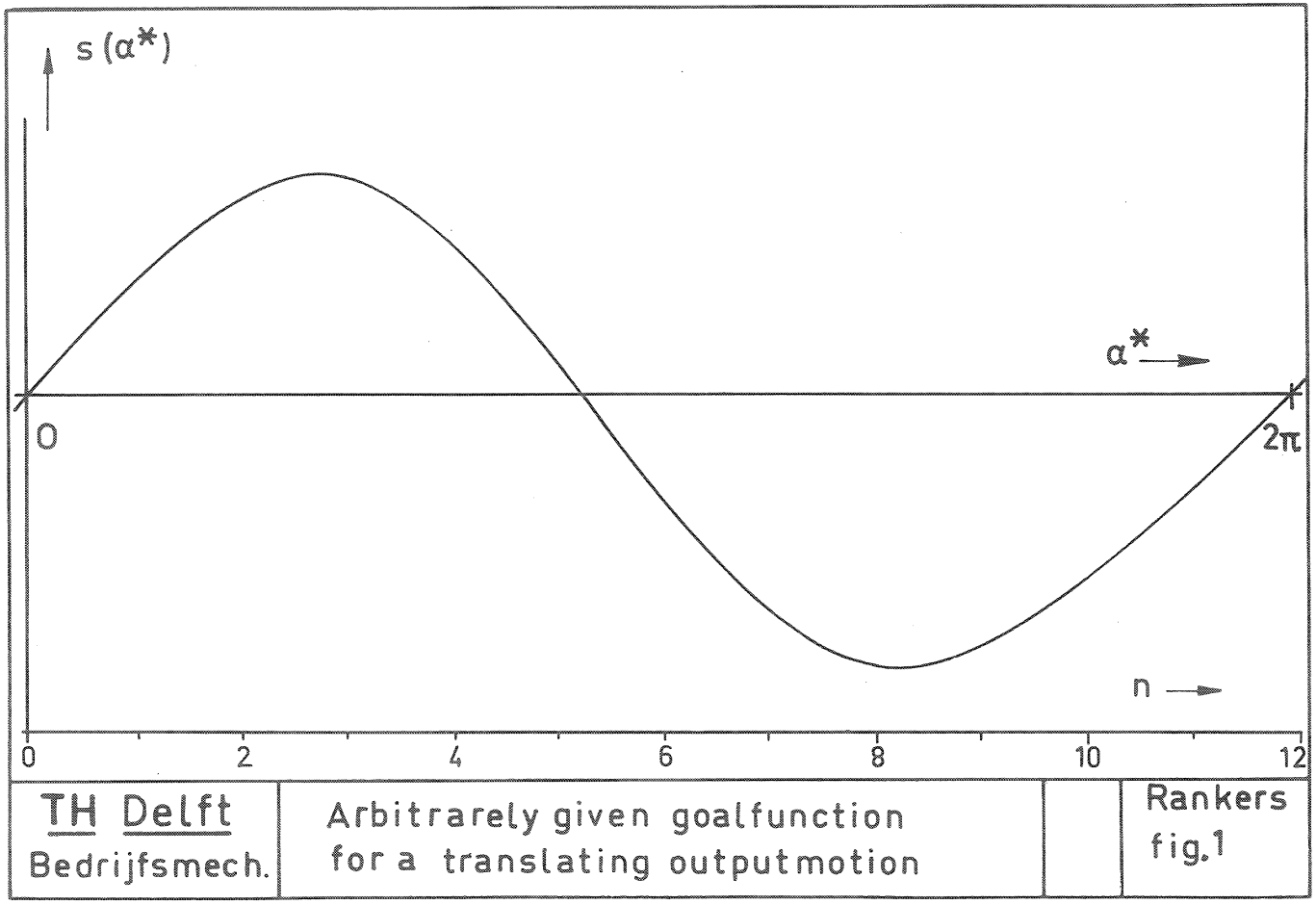
```
MECHANISME NR. T008
PARAMATER 1 HEEFT DE WAARDE     0.244
PARAMETER 2 HEEFT DE WAARDE    43.50
PARAMETER 3 HEEFT DE WAARDE     0.37
TAU =  1.719
L   =  3
U   = -7.642
```

Uitbreiding van het systeem

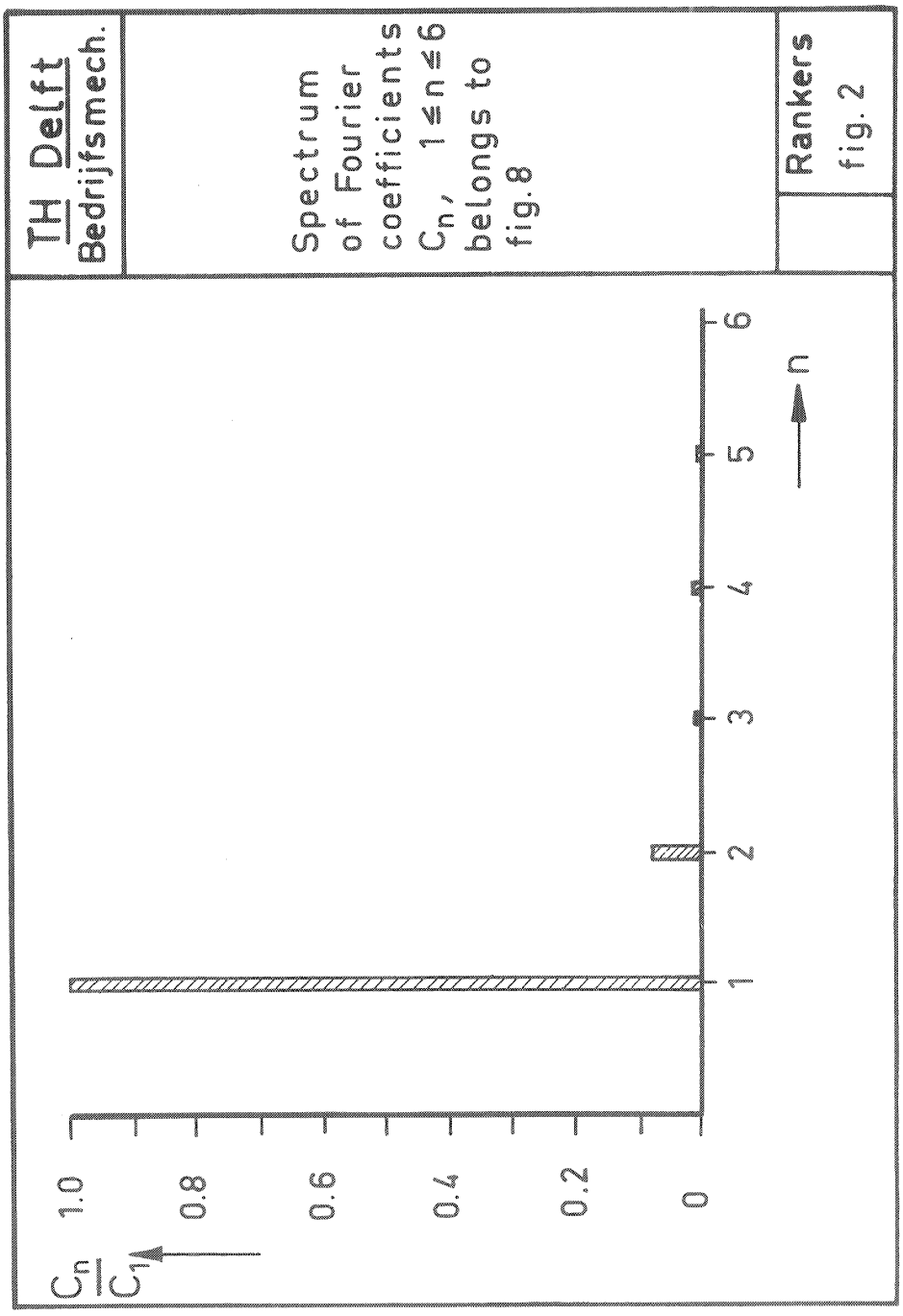
De waarde van de syntheseprocedure wordt mede bepaald door het aantal mechanismen, dat in totaal in de mechanismen-catalogus is opgeslagen.

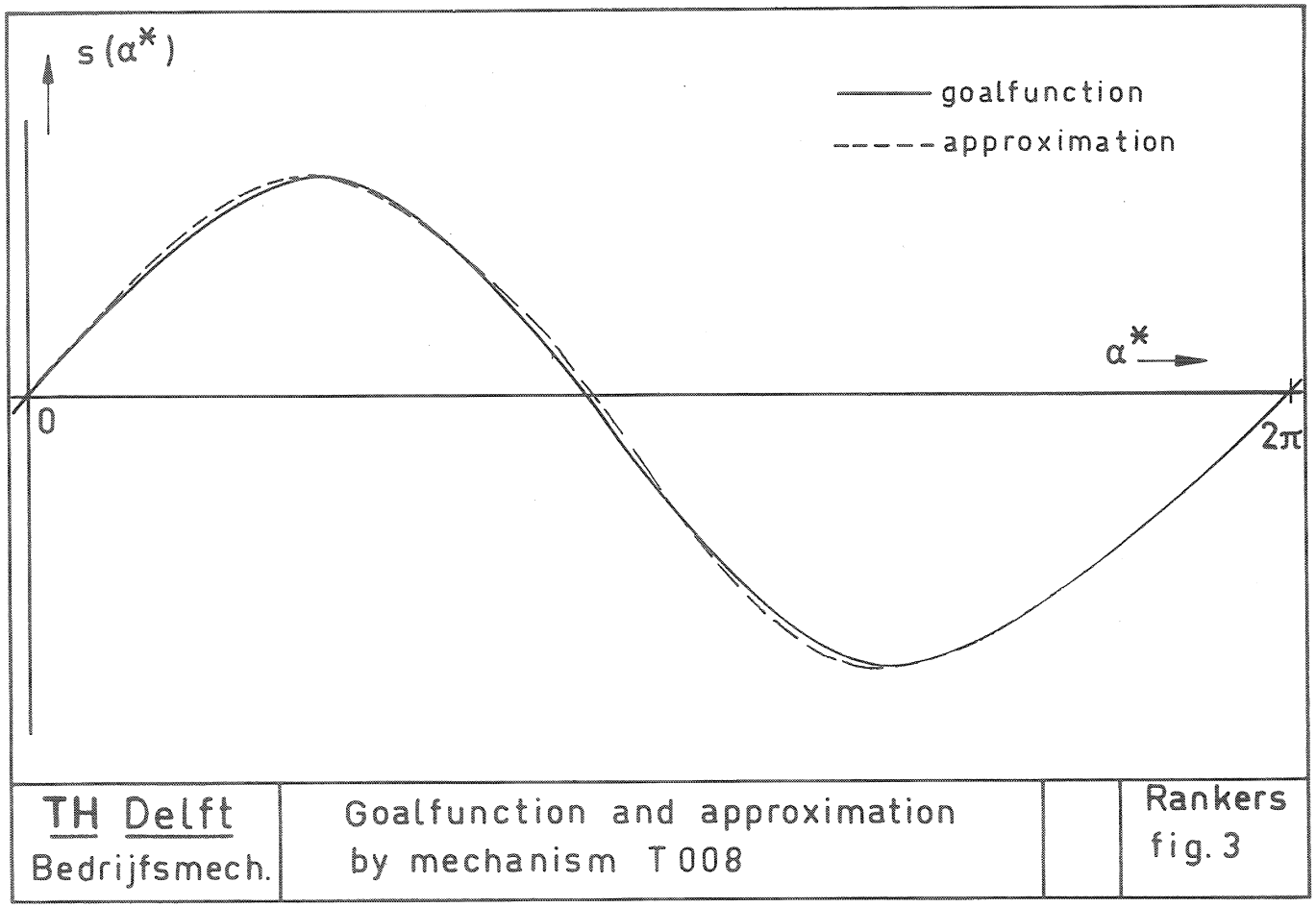
Daarom is de voornaamste bezigheid van de CADOM-groep gericht op het uitbreiden van deze catalogus.

Op dit moment zijn er zes mechanismen met doordraaiende uitgaande beweging, 14 mechanismen met heen en weer slingerende uitgaande beweging

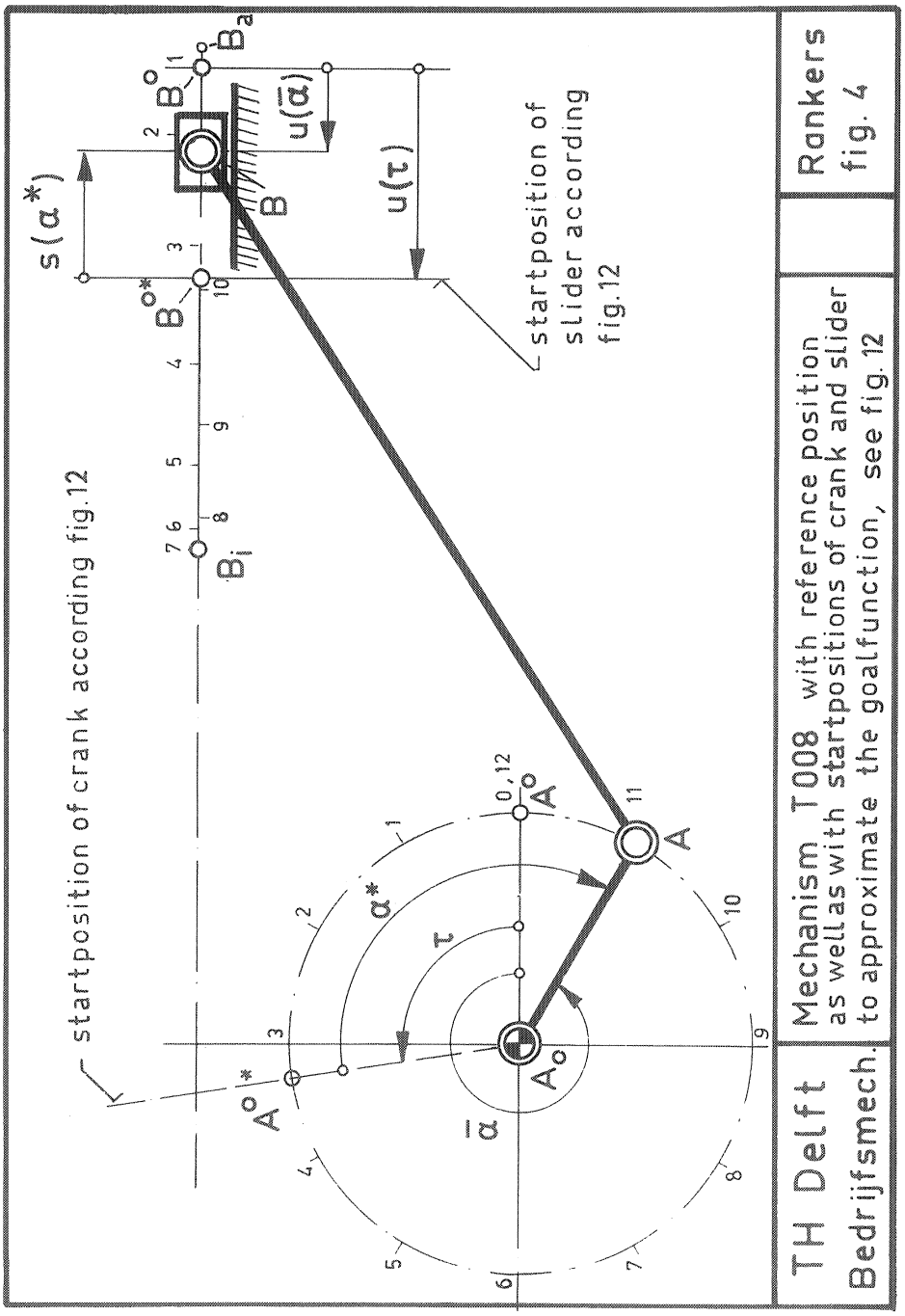


TH Delft Bedrijfsmech.	Arbitrarely given goalfunction for a translating outputmotion		Rankers fig.1


TH Delft
Bedrijfsmech.
Spectrum
of Fourier
coefficients
C_n, $1 \leq n \leq 6$
belongs to
fig.8
Rankers
fig. 2
C_n/C_1
1.0
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
n



TH Delft Bedrijfsmech.	Goalfunction and approximation by mechanism T 008		Rankers fig. 3


s(α*)
u(ᾱ)
u(τ)
B°
B*
B
Bi
Ba
A°
A*
Ao
A
α*
τ
ᾱ
startposition of crank according fig.12
startposition of slider according fig.12
TH Delft
Bedrijfsmech.
Mechanism T008 with reference position as well as with startpositions of crank and slider to approximate the goalfunction, see fig. 12
Rankers
fig. 4

en 19 mechanismen met heen en weer schuivende uitgaande beweging in de catalogus opgenomen.

Waarom benutten wij de beeldbuis?

Tot hiertoe heb ik U uiteengezet wat in het kader van computer aided design of mechanical components nu reeds berekend kan worden. Het doel daarbij is het bepalen van het *type* van een voor de approximatie van een doelfunctie aangewezen mechanisme en om de *dimensies* van de schakels te weten te komen. Daarom richt ik mij nu op de vraag waarom naast het benutten van de computer ook de GRAPHICS DISPLAY toegepast wordt. Daarvoor bestaan tal van redenen.

Geen beslissingsalgorithme

Als het moeilijk of onmogelijk is strenge eisen te formuleren en als beslissingen meer op ervaring en/of intuïtie berusten -maar dan wel op een ander niveau dan de ouderwetse punaise-methode- dan is het verantwoord om op de beeldbuis over te stappen.

Demonstratie

De toepassing van de beeldbuis is bijzonder attractief om te demonstreren wat men eigenlijk aan het doen is. De successen hangen samen met het feit, dat een ontwerper en constructeur meestal visueel ingesteld is en meer informatie kan ontlenen aan een grafiek dan aan een reeks getallen. Het voordeel van de uitvoer op de beeldbuis boven de uitvoer van grafieken m.b.v. de printer of de plotter ligt in de mogelijkheid veranderingen en bewegingen zichtbaar te maken. De aanvraag voor deze toelichting heb ik zelf ook te danken aan het feit, dat de Graphics Groep ter T.H. Delft, na een demonstratie bijgewoond te hebben, plezier had in onze toepassing.

Interactieve werkwijze

De interactieve werkwijze m.b.v. de beeldbuis schept bovendien de mogelijkheid om het programma versneld te doorlopen, lussen te onderbreken en sprongen te maken als dat gewenst is, of als de gebruiker constateert dat het verloop van het programma ongewenste resultaten oplevert.

Tenslotte willen wij niet vergeten te vermelden, dat de interactieve

werkwijze bijdraagt tot het verkrijgen van ervaring met het ingewikkelde rekenproces en de gebruiker in staat stelt de sensatie te ondergaan wat het resultaat van een bepaalde invoer zou kunnen zijn.

Dit laatste heeft vooral betrekking op de resultaten van de Fourier-analyse van een met weinig punten getekende doelfunctie of op de invloed van de toevoeging van een bepaald punt.

LITERATUURLIJST

[1] RANKERS, H., *Angenährte Getriebsynthese durch harmonische Analyse der vorgegebenen periodischen Bewegungsverhältnisse*, Dissertation TH Aachen, 1958.

[2] RANKERS, H. A. VAN DIJK, A.J. KLEIN BRETELER & K. VAN DER WERFF, *TADSOL - Type And Dimension Synthesis of Link Mechanisms.* A user oriented discription of the computer program. Proceedings of the symposium on computer aided design in mechanical engineering, Milan 1976, pp. 51-65.

[3] CADOM-Groep TH Delft: PION - FORTRAN SUBROUTINE PAKKET; periodieke functies: *input, output, numerieke behandeling*, Technische Hogeschool Delft, Sectie Bedrijfsmechanisatie en Leer der Mechanismen, 1974.

[4] MEYER ZUR CAPELLEN, W., *Über gleichwertige periodische Getriebe*, Seifen-Fette-Anstrichmittel, 59.(1957)4., pp. 257/266.

[5] MEYER ZUR CAPELLEN, W., *Die harmonischen Analyse an zykloidengesteuerten Schleifen*, Forschungsberichte des Landes Nordrhein-Westfalen Nr. 835, Westdeutscher Verlag Köln und Opladen, 1961. (Dort weiteres Schrifttum.)

[6] RANKERS, H.: *Synthese der umlaufenden zentrischen Kurbelschleife zweiter Art, Mechanism and Machine Theory*, will be published in 1977.

[7] CADOM-Groep TH Delft, *Mechanismen-Catalogus*, Technische Hogeschool Delft, Sectie Bedrijfsmechanisatie en Leer der Mechanismen, 1976.

INTERAKTIEF ONTWERPEN VAN MULTIVARIABELE REGELSYSTEMEN

A.J.J. VAN DER WEIDEN, P. VALK & O.H. BOSGRA
Laboratorium voor
Werktuigkundige Meet– en Regeltechniek
T.H. Delft

1. Inleiding.

In de regeltechniek tracht men door middel van stuursignalen een aantal belangrijke variabelen van een proces of systeem konstant te houden of volgens een bepaald gewenst patroon in de tijd te laten verlopen.
De stuursignalen worden toegevoerd aan een korrigerend orgaan, zoals bijvoorbeeld het roer van een schip bij de scheepsbesturing of een klep welke een massastroom beinvloedt bij de procesregeling. In veel gevallen worden de stuursignalen bepaald door van terugkoppeling gebruik te maken.
De te regelen grootheid wordt dan gemeten en vergeleken met een gewenste waarde; het verschilsignaal wordt door een regelaar omgezet in een geschikt stuursignaal. In fig.1.1 is dit in blokschema weergegeven.

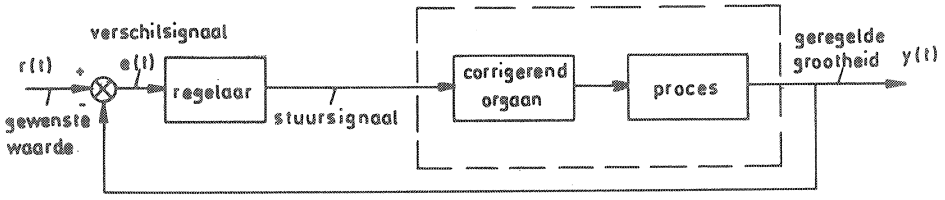


Fig.1.1 : Teruggekoppeld systeem.

De signalen in dit schema moeten beschouwd worden als funktie van de tijd, evenals het gedrag van regelaar en proces. Dit betekent dat van het te regelen systeem het dynamisch gedrag van belang is, zoals dit wordt beschreven door differentiaalvergelijkingen. In veel gevallen is voor een beperkt werkgebied het dynamisch gedrag als lineair te beschouwen, waardoor gebruik gemaakt kan worden van de systeemtheorie voor lineaire systemen en van integraal-transformaties. De regelaar bepaalt het stuursignaal.
Zowel om stabiliteit van het teruggekoppelde systeem te garanderen als om het geregelde systeem een gewenst gedrag te geven dient de regelaar aangepast te worden aan de dynamische eigenschappen van het te regelen proces.
We kunnen hierbij spreken van een regeltechnisch ontwerp, dat in een praktijksituatie de volgende punten omvat:

a. Het vastleggen van de regelopdracht (welke grootheid dient geregeld te worden, en op welk korrigerend orgaan kan worden ingegrepen);
b. Het analyseren van het te regelen systeem (het formuleren van een konceptueel en/of kwantitatief dynamisch model);
c. Het bepalen van de aard van de terugkoppeling (keuze van het type regelaar) en het bepalen van de parameters hierin (regelaarinstelling);
d. Het realiseren van deze oplossing door het kiezen, aanschaffen en installeren van instrumenten;
e. Het eventueel op grond van de werking in de praktijk nader instellen van de regelaar.

Indien er in een proces één te regelen variabele is en op één korrigerend orgaan wordt ingegrepen, spreken we van een skalair systeem.

Minder eenvoudig wordt het als het nodig is om een aantal variabelen van een systeem te regelen en als hiervoor een aantal korrigerende organen ter beschikking staat. Als elk van de korrigerende organen invloed heeft op meer dan één te regelen variabele, dan spreken we van een multivariabel systeem en van een multivariabel regelprobleem. In deze definitie moeten we het begrip "heeft invloed op" in technische zin interpreteren. Als één van de korrigerende organen (hierna aangeduid met ingangsvariabelen) steeds een duidelijke invloed heeft op één van de te regelen variabelen (uitgangsvariabelen) en slechts in geringe mate op elk van de andere uitgangsvariabelen, dan kunnen deze ingang en uitgang worden beschouwd als behorende bij een skalair systeem.

Het multivariabele karakter van een systeem ontstaat als gevolg van de interne relaties tussen de systeemvariabelen. Beschouw daartoe als voorbeeld het mengproces van fig.1.2. In een vat worden twee massastromen gemengd die elk een zekere koncentratie van een bepaalde stof bezitten. De regelopdracht houdt in, dat de koncentratie van de uitgaande massastroom en het niveau konstant worden gehouden. Het proces kan worden verstoord door variaties in de uitgaande massastroom en door variaties in de koncentraties van de ingaande stromen.

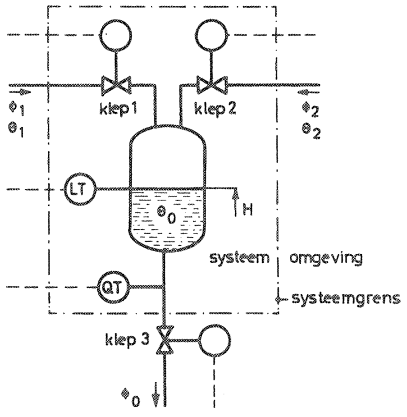


Fig.1.2: Mengproces als voorbeeld van een multivariabel systeem.

In de figuur is een systeemgrens aangegeven; de systeemgrens definieert wat we als "proces" zullen opvatten en wat we als "omgeving" beschouwen.
Signalen welke de systeemgrens passeren zijn:

a. stuursignalen voor klep 1 en klep 2 (ingangen),
b. meetsignalen van een kwaliteitsopnemer en van een niveau-opnemer (uitgangssignalen),
c. massastroomvariaties in de uitgaande stroom en druk- en koncentratievariaties van de ingaande massastromen (storingen).

Een dynamisch model van een dergelijk systeem kan in gelineariseerde en Laplace-getransformeerde vorm weergegeven worden door vier overdrachtsfunkties (fig.1.3).

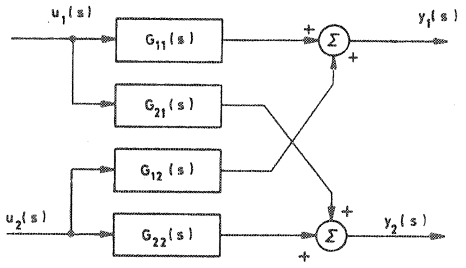


Fig.1.3: Multivariabel systeem met twee in- en uitgangen.

In principe heeft elk van de ingangssignalen invloed op elk van de te regelen uitgangsvariabelen. Dit fenomeen wordt "interaktie" genoemd. Indien twee afzonderlijke regelkringen zouden worden toegepast, dan beïnvloeden deze elkaar op systematische wijze als gevolg van de koppelingen via $G_{12}(s)$ en $G_{21}(s)$.
Er moet dan gezocht worden naar een regelaar welke zelf een multivariabel karakter heeft. Dit is een probleem dat in de volgende hoofdstukken nader uitgewerkt zal worden.
Verschillende methoden zijn beschikbaar om tot een oplossing te komen. Enerzijds kan gebruik gemaakt worden van tijddomein-methoden zoals de optimale regeltheorie; daarnaast is het mogelijk gebleken om de traditionele frekwentiedomein-methoden,

ontwikkeld door Nyquist, Bode, Evans en anderen, te generaliseren tot het multivariabele geval. Deze frekwentiedomein-methoden zijn zeer geschikt als grondslag voor een ontwerpprocedure, omdat de gevolgen van een beslissing tijdens het ontwerp op eenvoudige wijze grafisch weergegeven kunnen worden. Hier kan een rekenmachine met visuele informatieweergave zeer zinvol gebruikt worden. De visuele informatie in de vorm van polaire figuren of stapresponsies van het regelsysteem levert een ontwerper een goed inzicht in het probleem. Door de rekenmachine konversationeel te gebruiken, kan direkt op de gepresenteerde informatie gereageerd worden. De computer kan dan het (soms aanzienlijke)rekenwerk verrichten, de ontwerper wordt alleen met de gevolgen van een ontwerpbeslissing gekonfronteerd.
De interaktie tussen computer en ontwerper maakt het mogelijk om de "trial and error" aspekten van elke ontwerpprocedure verantwoord te laten verlopen. Ook met allerlei praktische aspekten van het regeltechnisch ontwerp kan dan rekening worden gehouden, zoals eenvoud van de regelaarstruktuur, het afwegen van responsiesnelheid tegenover de relatieve demping van het geregelde systeem, of het behoud van stabiliteit bij het defekt raken van een van de signaalopnemers.
Verschillende regeltechnische ontwerpmethoden voor multivariabele systemen zijn recentelijk ontwikkeld speciaal met het oog op interaktief computergebruik met visuele informatiepresentatie (Rosenbrock 1969, MacFarlane en Belletrutti 1973, Mayne en Chuang 1973).
In het hierna volgende zal gekeken worden naar Rosenbrock's "Inverse Nyquist Array" methode.

2. Inverse Nyquist array ontwerpmethode.

Het vinden van een goede regeling voor interaktieve multivariabele systemen is heel wat gekompliceerder dan het bepalen van regelingen voor skalaire systemen.
Een van de eerste methoden om het probleem van de interaktie aan te pakken, was het simpel elimineren van deze interaktie waarna het systeem als een aantal skalaire systemen beschouwd wordt (Boksenbom en Hood 1949).

Wordt een systeem met overdrachtsmatrix G(s) geregeld door een multivariabele regelaar K(s), dan is de overdrachtsmatrix van het resulterende systeem gelijk aan

$$Q(s) = G(s)K(s) \qquad (2.1)$$

Door K(s) nu zo te kiezen dat Q(s) een diagonaalmatrix wordt, is de interaktie uit het systeem geëlimineerd. Dit geeft echter vaak zeer gekompliceerde regelaars die fysisch niet altijd te verwezenlijken zijn.
Het diagonaal maken van Q(s) gaat dus eigenlijk te ver. De I.N.A. (Inverse Nyquist Array)-methode verzwakt de eis dat Q(s) diagonaal moet zijn, tot de eis van diagonaaldominant zijn van Q(s) (of $Q^{-1}(s)$). In plaats van met Q(s) zal met $Q^{-1}(s)$ gewerkt worden, genoteerd als $\hat{Q}(s) = Q^{-1}(s)$.
Voor het systeem van fig.2.1 met terugkoppelmatrix H is de overdrachtsmatrix van de gesloten keten

$$R(s) = [I_m + Q(s)H]^{-1}Q(s) = [Q(s) + H]^{-1} \qquad (2.2)$$

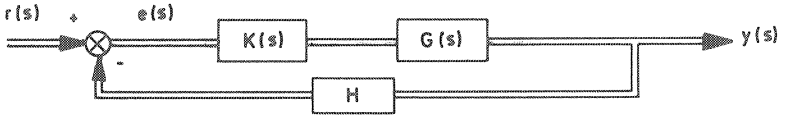


Fig.2.1: Teruggekoppeld multivariabel systeem.

Een eenvoudige berekening levert dat

$$R(s)^{-1} = \hat{R}(s) = H + \hat{Q}(s) = H + \hat{K}(s)\hat{G}(s) \qquad (2.3)$$

Als nu H diagonaal gekozen wordt dan is

$$\begin{aligned} \hat{r}_{ii}(s) &= h_i + \hat{q}_{ii}(s) \\ \hat{r}_{ij}(s) &= \hat{q}_{ij}(s), \quad i \neq j \end{aligned} \qquad (2.4)$$

waarin $\hat{r}_{ij}(s)$ het ij^{de} element van $\hat{R}(s)$ is.

Diagonaal-dominantie van een overdrachtsmatrix kan op verschillende manieren geïnterpreteerd worden. Diagonaal rij-dominant bijvoorbeeld heeft de volgende definitie.

Stel D is de gebruikelijke Nyquist contour in het komplexe vlak, bestaande uit de imaginaire as van $-j\alpha$ tot $j\alpha$ en een halve cirkel met straal α (met $\alpha \to \infty$) in het rechter halfvlak en met het middelpunt in de oorsprong.
Stel dat $\hat{Q}(s)$ een mxm matrix is. Dan is $\hat{Q}(s)$ diagonaal rij-dominant op D als $\hat{Q}(s)$ geen pool op D heeft en

$$|\hat{q}_{ii}(s)| - \sum_{\substack{j=1 \\ j\neq i}}^{m} |\hat{q}_{ij}(s)| > 0 \qquad (2.5)$$

voor $i=1,2,\ldots,m$ en voor alle $s \epsilon D$.
Een overeenkomstige definitie bestaat er voor kolom-dominantie (Rosenbrock 1974). Het belang van diagonaal-dominantie van de inverse overdrachtsmatrix $\hat{Q}(s)$ voor het ontwerpen van regelingen voor multivariabele systemen is het volgende.
Stel dat het inverse Nyquist diagram van het diagonaal element $\hat{q}_{ii}(s)$ van de inverse overdrachtsmatrix $\hat{Q}(s)$ de oorsprong van het komplexe-vlak $\hat{N}_i$ keer omcirkelt in de richting van de klok.
Neem verder aan dat het inverse Nyquist diagram van de rationale funktie $|\hat{Q}(s)|$ de oorsprong $\hat{N}$ keer omcirkelt.
Als dan $\hat{Q}(s)$ dominant op $\hat{D}$ is hebben we de relatie

$$\hat{N} = \sum_{i=1}^{m} \hat{N}_i \qquad (2.6)$$

Verder kan de stabiliteit van het gesloten systeem onderzocht worden aan de hand van het Nyquist diagram van de determinant van de terugkeerverschil-matrix

$$F(s) = I + Q(s)H \qquad (2.7)$$

Daar nu

$$|I + Q(s)H| = \frac{|\hat{R}(s)|}{|\hat{Q}(s)|} \qquad (2.8)$$

kan de stabiliteit van de gesloten keten bepaald worden uit de Nyquist diagrammen van $|\hat{Q}(s)|$ en $|\hat{R}(s)| = |H + \hat{Q}(s)|$.
Zijn nu zowel $\hat{Q}(s)$ en $\hat{R}(s)$ diagonaal-dominant dan kan vergelijking (2.6) gebruikt worden om de stabiliteit van het teruggekoppelde systeem te onderzoeken aan de hand van de inverse Nyquist diagrammen van $\hat{q}_{ii}(s)$ en $\hat{r}_{ii}(s) = h_i + \hat{q}_{ii}(s)$.
Dit leidt tot een grafische methode om stabiele regelingen te ontwerpen.

2.1. Grafische interpretatie.

Veronderstel dat de overdrachtsmatrix G(s) gegeven is, en dat de ingangskompensator K(s) gekozen moet worden. De computer kan nu $\hat{Q}(s)$ $(Q(s))$ berekenen, daaruit $\hat{q}_{ii}(j\omega)$ voor een aantal waarden van ω bepalen en het korresponderende inverse Nyquist diagram op een beeldscherm tekenen. Gewoonlijk zal $\hat{q}_{ii}(j\omega)$ een vorm hebben zoals in fig.2.2 gegeven.

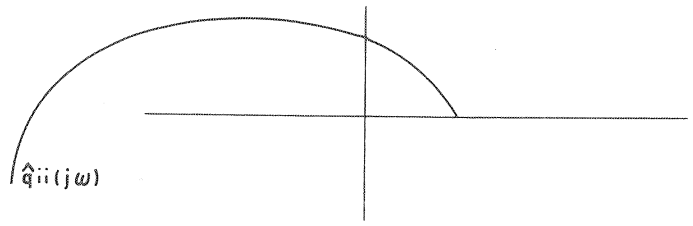


Fig.2.2: Inverse Nyquist diagram van element $\hat{q}_{ii}(j\omega)$.

Voor een bepaalde frekwentie ω_1 ligt de uitdrukking

$$\hat{d}_i = \sum_{\substack{i=1 \\ i\neq j}}^{m} \left| \hat{q}_{ij}(j\omega_1) \right| \qquad (2.9)$$

vast en kan een cirkel met straal $\hat{d}_i$ en middelpunt op $\hat{q}_i(j\omega_1)$ getrokken worden. Als dit gedaan wordt voor een aantal frekwentie punten ontstaat een omhullende van dergelijke cirkels (Gershgorin-band) zoals in fig.2.3.

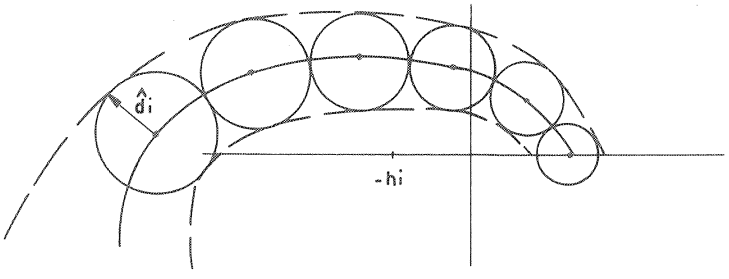


Fig.2.3: Inverse Nyquist diagram met Gershgorin cirkels.

Nu voldoet $\hat{Q}(s)$ op de imaginaire as van de D contour aan vergelijking (2.5) als voor alle m inverse Nyquist diagrammen de oorsprong buiten de "Gershgorin-banden" liggen.

Op dezelfde wijze kan diagonaal-dominantie van $\hat{R}(s)$ bekeken worden door te eisen dat het kritieke punt $(-h_i,0)$, op grond van vergelijking (2.4), buiten de bovengenoemde "Gershgorin-banden" ligt. Als nu $\hat{Q}(s)$ en $\hat{R}(s)$ diagonaal-dominant zijn, kan gesproken worden van het aantal omcirkelingen van de oorsprong (het kritieke punt) van de i^{de} "Gershgorin-band". Dit aantal $\hat{N}_{oi}$ ($\hat{N}_{ci}$) zal gelijk zijn aan het aantal omcirkelingen van de oorsprong (kritieke punt) door de afbeelding van $\hat{q}_{ii}(j\omega)$. Hieruit wordt indien $\hat{Q}(s)$ en $\hat{R}(s)$ diagonaal-dominant zijn de noodzakelijke en voldoende voorwaarde voor de stabiliteit van de gesloten keten afgeleid, namelijk:

$$\sum_{i=1}^{m} (\hat{N}_{oi} - \hat{N}_{ci}) = p_o \qquad (2.10)$$

waarin p_o het aantal instabiele polen van de open-keten is. Is het niet-teruggekoppelde systeem stabiel, dan is $p_o = 0$. Als dan voor alle "Gershgorin-banden" $\sum_{i=1}^{m}(\hat{N}_{oi} - \hat{N}_{ci}) = 0$, dan is aan vergelijking (2.10) voldaan en is het teruggekoppelde systeem stabiel.
De stabiliteit van een systeem kan dus onderzocht worden aan de hand van de "Gershgorin-banden" van de diagonaal-elementen $\hat{q}_{ii}(s)$ van de inverse Nyquist diagrammen.

In het algemeen representeren de diagonaalelementen $\hat{q}_{ii}(s)$ grootheden die niet direkt meetbaar zijn aan een systeem. Door nu gebruik te maken van een theorema van Ostrowski (1952) kan aangetoond worden dat

$$r_i^{-1}(s) \triangleq r_{ii}^{-1}(s) - h_i \qquad (2.11)$$

altijd binnen de omhullende van de Gershgorin-cirkels van element $\hat{q}_{ii}(s)$ blijft, wat ook de waarde van h_j (tussen 0 en h_j) in elke andere keten wordt. Merk op dat $r_{ii}^{-1}(s)$ de inverse overdrachtsfunktie is tussen ingang i en uitgang i waarbij alle andere ketens gesloten zijn, en $r_i(s)$ is de overdrachtsfunktie in de i^{de} keten, als deze open is en alle andere ketens gesloten zijn. Het is deze overdrachtsfunktie waarvoor een skalaire regeling voor de i^{de} loop ontworpen moet worden.

De band waarin $r_i^{-1}(s)$ ligt kan echter nog smaller gemaakt worden. Als $\hat{Q}(s)$ en $\hat{R}(s)$ diagonaal-dominant zijn en als

$$\phi_i = \max_{\substack{j \\ j \neq i}} \frac{\hat{d}_j(s)}{|h_j + \hat{q}_{jj}(s)|} \qquad (2.12)$$

dan ligt $r_i^{-1}(s)$ binnen een "band" gebaseerd op $\hat{q}_{ii}(s)$ en bepaald door de cirkels met de stralen

$$rad_i(s) = \phi_i \hat{d}_i(s) \qquad (2.13)$$

Dus als de versterkingsfaktoren voor de gesloten-keten gekozen zijn, zodat er een stabiel systeem verkregen is, dan kan er een maat voor de versterkingsmarge voor elke keten bepaald worden, door het tekenen van de cirkels met de stralen $rad_i(s)$. Deze kleinere "banden" reduceren ook voor elke keten het gebied waarin de inverse overdrachtsfunktie $r_{ii}^{-1}(s)$ kan liggen.

2.2. Ontwerpmethodiek.

De ontwerpmethode voorgesteld door Rosenbrock bestaat in principe uit het bepalen van een ingangskompensator $K_p(s)$ zodat het produkt $G(s)K_p(s)$ diagonaal-dominant is.
Als aan deze konditie voldaan is dan kan er een diagonaal-matrix $K_d(s)$ gekozen worden zodanig dat het totale systeem aan vereiste specifikaties voldoet.

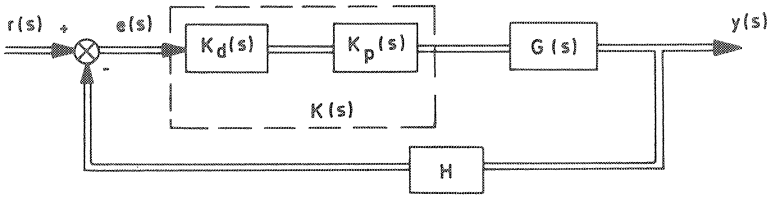


Fig.2.4: Algemeen systeem.

Daar het ontwerp zich afspeelt in het inverse domein, zijn we dus eigenlijk bezig met het bepalen van een inverse kompensator $\hat{K}_p(s)$ zodat $\hat{Q}(s) = \hat{K}_p(s)\hat{G}(s)$ diagonaal-dominant is.

Een manier om $\hat{K}_p(s)$ te bepalen is de gewenste matrix op te bouwen uit elementaire rij-operaties, gebruik makend van de visuele informatie verkregen van alle andere elementen van $\hat{Q}(s)$. Deze methode blijkt in veel gevallen tot het gewenste resultaat te leiden.
Voor verschillende andere methoden voor het bepalen van een kompensator $K_p(s)$ zie Rosenbrock (1974).

3. Programmatuur.

Daar de I.N.A. methode uitermate geschikt is om met behulp van een computer tot een resultaat te komen, is juist voor deze ontwerpmethode een konversationeel computer programma geschreven. Hierbij is er vanuit gegaan, dat de ontwerpmethodiek als volgt gehanteerd dient te worden:

- Men moet interaktief kunnen werken, liefst via een beeldscherm en dit moet snel kunnen gaan.
- De werkvolgorde moet willekeurig bepaald kunnen worden door de ontwerper. Juist bij deze ontwerpmethode is het belangrijk, dat men naar eigen inzicht kan handelen en niet bij voorbaat al vast zit aan een bepaald werkpatroon.
- Men moet snel een goed inzicht kunnen opbouwen in de interaktie van het systeem, zowel bij lage als hoge frekwenties.
- Via visuele informatie over alle tussenstappen moeten de regelaars op verantwoorde wijze ontworpen kunnen worden, vertrouwend op een goede interaktie mens-computer.

De minimale eisen waaraan een computer moet voldoen om gebruikt te kunnen worden als ontwerpgereedschap zijn de volgende:

- Snelle "floating-point" operaties moeten mogelijk zijn om de benodigde rekentijd kort te houden.
- De machine moet beschikken over een groot achtergrondgeheugen, om de benodigde gegevens op te kunnen slaan.
- "Overlay" faciliteiten zijn noodzakelijk gezien de uitgebreidheid van de verschillende algorithmes.
- Er moet een beeldscherm aanwezig zijn met de benodigde "graphics software" zodat interaktief gewerkt kan worden op grond van visuele informatie.

Voor een PDP-11/45 is een modulair opgebouwd computerprogramma geschreven waarmee alle benodigde handelingen kunnen worden uitgevoerd. Met gebruikmaking van "overlaystrukturen" en "dataopslag" op een schijf blijkt het mogelijk om met 16K kerngeheugen te werken. In het programma is de mogelijkheid opgenomen om een systeem, dat in een andere vorm dan als overdrachtsmatrix gegeven is, te transformeren alvorens de ontwerpfase kan beginnen. Hierna wordt de inverse van de rationale matrix G(s) berekend (van der Weiden 1976).
Uit $\hat{G}(s)$ volgt dan $\hat{G}(j\omega)$ voor een reeks van frekwentiepunten waarmee de I.N.A. direkt beschikbaar is.
Door middel van een "overlay" (blok) is nu de vrijheid van de ontwerper ingebouwd. Op het beeldscherm verschijnt de optielijst (fig.3.1). ($K_c = K_d$ en $K_a K_b = K_p$ zie fig.2.4)

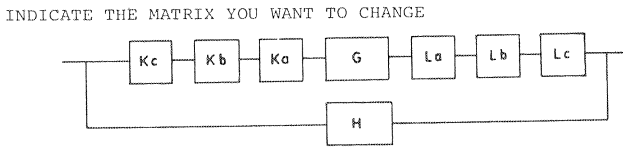


Fig.3.1: Optielijst.

In willekeurige volgorde kan een van de regelmatrices worden veranderd en eventueel worden opgeslagen (L_a, L_b, Lc zijn uitgangskompensatoren waarmee indien nodig ook het systeem diagonaal-dominant gemaakt kan worden).
Het resultaat verkregen met de nieuwe matrix kan direkt op het beeldscherm worden gezet. Hierna is vergelijking met voorgaande niveaus mogelijk. Ook kan de I.N.A. van de overdrachtsmatrix van de gesloten keten op elk tijdstip op het beeldscherm gebracht worden. Bij het diagonaal-dominant maken van het systeem, kunnen de kompensatoren zowel in het inverse vlak als in de oorspronkelijke vorm gekozen worden.

Keuze in het inverse vlak is belangrijk vanwege de eenvoudige interpretatie voor het effekt op de I.N.A., keuze in de oorspronkelijke vorm is van belang om de regelaars fysisch zo eenvoudig mogelijk te maken. Voor het bepalen van de regelaars is men niet alleen afhankelijk van de "trial and error" methoden. Er is hiervoor ook een "subroutine-pakket" beschikbaar, dat gebaseerd is op mathematische operaties. Het spreekt vanzelf, dat na elke stap de "Gershgorin-cirkels" of de "Ostrowski-cirkels" in de inverse polaire figuren van de diagonaal elementen getekend kunnen worden, opdat direkte visuele informatie over de bereikte dominantie verkregen wordt.

Verder is er voor het bepalen van de skalaire regelmatrix $K_d(s)$ de direkte informatie beschikbaar over bijvoorbeeld de fasemarge, de amplitudemarge, het raken aan een M-cirkel en andere belangrijke ontwerpkriteria. Zoals deze bij skalaire frekwentiedomein-technieken gehanteerd worden.
Is de totale regeling ontworpen, dan kan deze in het tijddomein onderzocht worden aan de hand van stapresponsies.

4. Ontwerpvoorbeeld.

Een systeem met 2 in- en uitgangen bestaat uit een reservoir, waarin door korrekties in de koud- en warmwaterstroom de temperatuur en het niveau geregeld moeten worden.

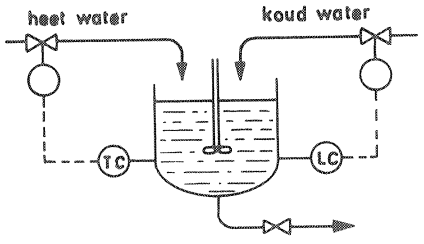


Fig.4.1: Processchema.

De overdrachtsmatrix van dit proces is (van der Weiden 1973):

$$G(s) = \begin{bmatrix} \frac{-1.0.e^{-3s}}{108s} & \frac{-1.0.e^{-3s}}{90s} \\ \frac{0.82.e^{-17s}}{(65s+1)(5s+1)} & \frac{-0.67.e^{-17s}}{(65s+1)(5s+1)} \end{bmatrix} \qquad (4.1)$$

waarin $u(s) = \begin{bmatrix} u_1(s) \\ u_2(s) \end{bmatrix}$, $y(s) = \begin{bmatrix} y_1(s) \\ y_2(s) \end{bmatrix}$ de ingangs- en uitgangsvektoren zijn:

$u_1(s)$ = koudwaterstroom
$u_2(s)$ = warmwaterstroom
$y_1(s)$ = niveau van het reservoir
$y_2(s)$ = temperatuur in het reservoir

De interaktie tussen de niet korresponderende in- en uitgangen kan niet verwaarloosd worden. De skalaire technieken kunnen dus niet zonder meer gebruikt worden om de regelaar voor dit systeem te ontwerpen.

Het ontwerpprocedé en de taak van de computer hierin zal nu aan de hand van een blokschema (fig.4.2) behandeld worden. Nadat de systeemgegevens ingelezen zijn en de I.N.A. van het ongekompenseerde systeem berekend en op het beeldscherm te zien zijn kan er aan het eigenlijke ontwerp begonnen worden. De noodzaak van het interaktief gebruik van de computer komt nu duidelijk naar voren. Het ontwerpblok kan gezien worden als het hart in het programma van waaruit beslissingen door de ontwerper genomen moeten worden aangaande de volgende stap in het ontwerpproces. Deze stap kan willekeurig gekozen worden en wordt geheel bepaald door de ontwerper. Ook kan er naar eerder genomen beslissingen teruggegaan worden.
Voor het ontwerp van een regeling voor bovengenoemd systeem zou nu als eerste stap de konstante matrix $\hat{K}_a$ gekozen kunnen worden. Dit is hier gebeurd aan de hand van "trial and error" methode, waarbij de resultaten van de geprobeerde matrices op schijf opgeslagen kunnen worden. De uiteindelijke matrix waarmee het ontwerp voortgezet wordt is

$$\hat{K}_a = \begin{bmatrix} 1. & 1.2 \\ -1.25 & 1. \end{bmatrix} \qquad (4.2)$$

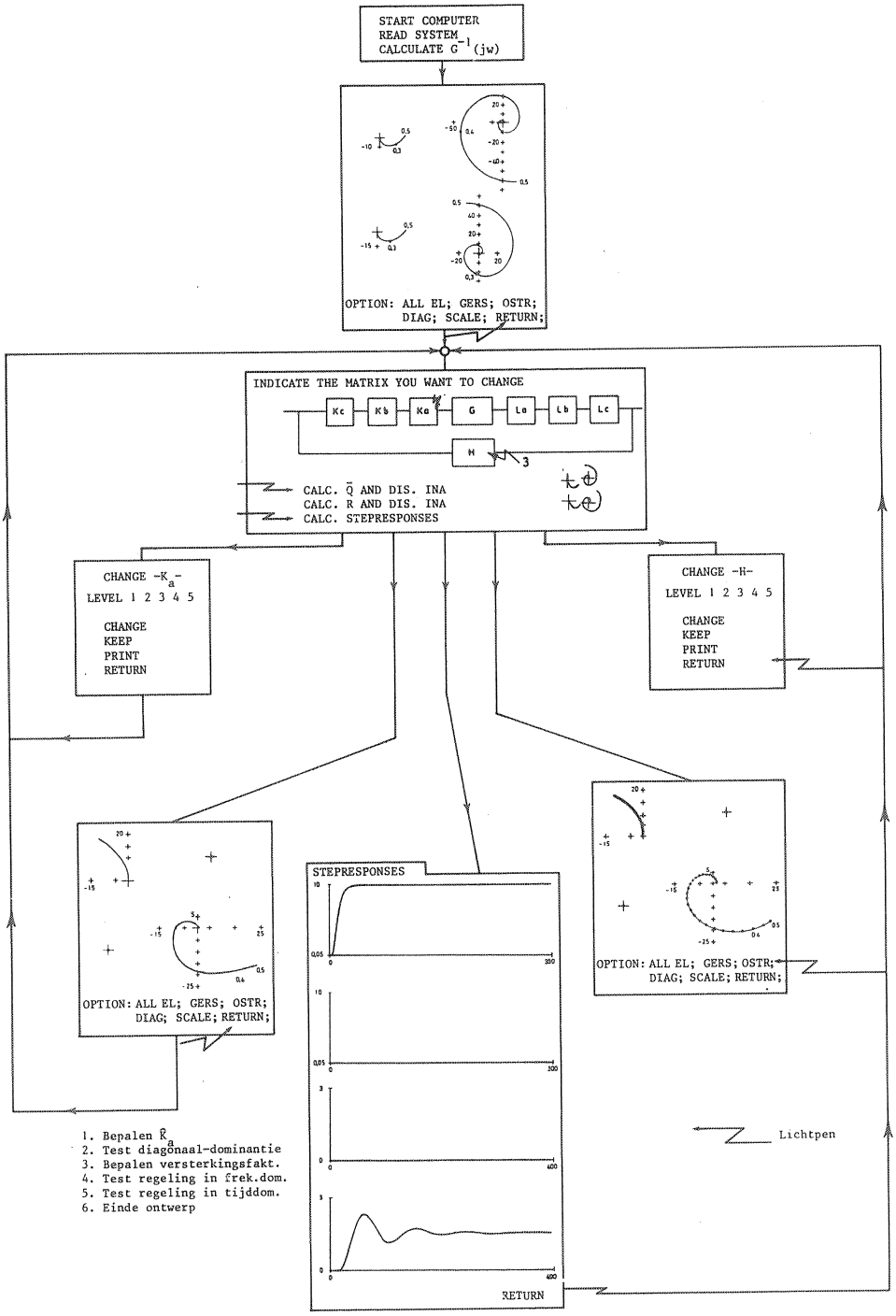


Fig.4.2: Blokschema ontwerp

Met deze kompensator is dit systeem diagonaal-dominant.
Als nu in de terugkoppel-matrix de versterkingsfaktoren $h_1 = 10.$ en $h_2 = 5.$ gekozen worden dan blijkt dat het teruggekoppelde systeem nagenoeg geen interaktie meer in zich heeft en stabiel is.
Hoe goed de regeling is kan nu getest worden aan de hand van de stapresponsies.
Uit deze stapresponsies blijkt dat de regeling nog verbeterd kan worden, door bijvoorbeeld een dynamische regelaar (K_c) in de tweede keten q_{22} te plaatsen.

5. Konklusies.

Een computersysteem met beeldstation blijkt een zeer belangrijk hulpmiddel te zijn bij het ontwerpen van multivariabele regelsystemen. Met behulp van grafische informatie is de ontwerper in staat het inzicht op te bouwen dat hem in staat stelt om verantwoorde en suksesvolle ontwerpbeslissingen te nemen. Belangrijk hierbij is ook het interaktieve computergebruik, waardoor de ontwerper in een willekeurige volgorde een aantal programmastappen kan doorlopen, op genomen beslissingen terug kan komen en proberenderwijs mogelijkheden kan verkennen. Een lichtpen vergemakkelijkt de dialoog omdat daarbij probleemgeorienteerde kommando's gebruikt kunnen worden.
Omgekeerd heeft de beschikbaarheid van dergelijke computerfaciliteiten ertoe geleid dat hierop geënte regeltechnische ontwerpmethoden zijn ontwikkeld (Rosenbrock 1969, MacFarlane en Belletrutti 1973, Mayne en Chuang 1973).
Het is mogelijk om met een relatief kleine computer met 16K kerngeheugen, schijf en beeldstation verantwoord te werken. Toepassingen in een industriële omgeving lijken dan ook haalbaar. De ontwerpmethoden sluiten goed aan bij de praktische randvoorwaarden welke bij industriële regelproblemen naar voren komen. Fysisch inzicht in het gedrag van het te regelen systeem kan daarbij direkt op nuttige wijze gehanteerd worden.
Theoretische achtergronden van de regeltechnische ontwerpmethoden zijn hier niet aan de orde gekomen. Voor verdere achtergronden kan het boek "Computer-aided control system design" van H.H. Rosenbrock (1974) aanbevolen worden.

Referenties.

1. Boksenbom, A.S. en Hood, R.: General algebraic method applied to control analysis of complex engine types. Report NACA-TR-980, National Advisory Commttee for Aeronautics, Washington D.C., 1949.
2. MacFarlane, A.G.J. en Belletrutti, J.J.: The characteristic locus design method. Automatica, vol. 9, p. 575-588, 1973.
3. Mayne, D.Q. en Chuang, S.C.: The sequential return-difference method for designing linear multivariable systems. I.E.E. Conference on Computer Aided Control System Design. Cambridge, 1973.
4. Ostrowski, A.M.: Note on bounds for determinants with dominant principal diagonal. Proc. Am. Math. Soc., vol. 3, p. 26-30, 1952.
5. Rosenbrock, H.H.: Design of multivariable systems using inverse Nyquist arry. Proc. I.E.E., vol. 116, p. 1929-1936, 1969.
6. Rosenbrock, H.H.: Computer-aided control system design. Academic Press, 1974.
7. Van der Weiden, A.J.J.: Inversion of rational matrices. To be plublished in Int. J. Control.
8. Van der Weiden, A.J.J.: Het ontwerpen van terugkoppeling voor multivariabele systemen in het frekwentiedomein met gebruik van een rekenmachine met een visueel display. Afstudeerverslag A-160, T.H. Delft. Lab. voor werkt. Meet- en Regeltechniek, 1973.

LOGICAL DESIGN AND DATA ANALYSIS WITH GPGS

L.C. CARUTHERS
Graphics / Informatica Faculty of Science, Nijmegen

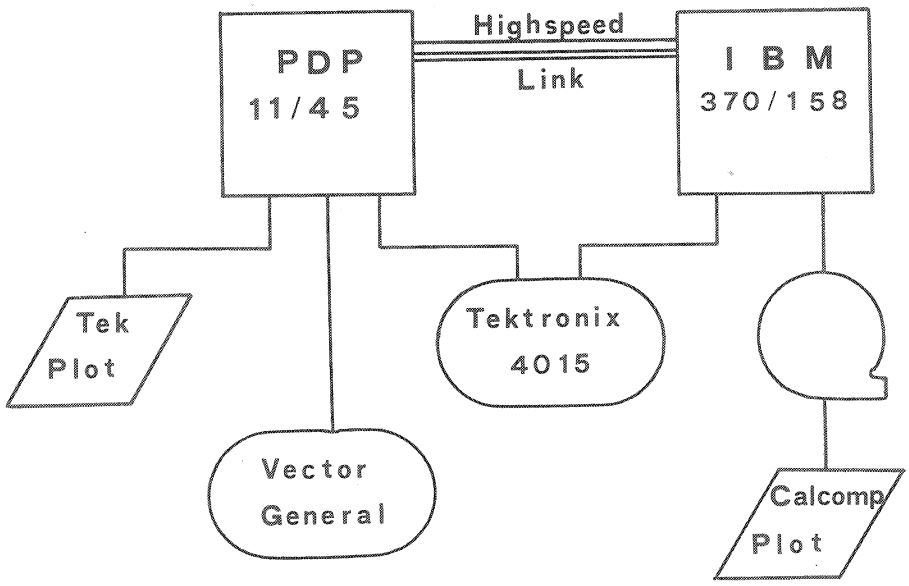
INTRODUCTION

Small, medium, and large are the three simplest words for categorizing the 3 graphics applications covered in this presentation. Each of these applications has been programmed in FORTRAN using GPGS. Each application can thus be run on different computers and with different graphics devices, thanks to the device independence of GPGS.

At Nijmegen there are many possible CPU, graphics device combinations available for use. The diagram on the following page shows our computer configuration with the PDP 11/45 that is connected via a high speed (600 + Kbaud) 16 line parallel link to the IBM 370/158 of the computing center. The small tektronix plotter can only be used from the PDP in stand-alone mode. The Vector General can be used either from the PDP in stand-alone mode or from the PDP operating as a satellite to the IBM. The Tektronix 4015 can be used either from the stand-alone PDP or from the IBM directly via TSO or from a batch job. The Calcomp plotter tapes can only be written from the IBM.

The versions of GPGS available at Nijmegen include the DOS and RT-11 versions for stand-alone use of the PDP. The FORTRAN version of GPGS is being adopted for use under the UNIX multiprogramming system on the PDP. Satellite support of the Vector General for the IBM is currently available with RT-11 or UNIX. Programs using the PDP as a satellite must run in batch on the IBM. The satellite configuration, the direct use of the Tektronix from TSO and batch and the Calcomp plotter are all useable from the IBM 360/370 version of GPGS.

Nijmegen Graphics Configuration



The small application will show the interactive analysis of astronomical data. The medium size application will show the interactive analysis of physics data. The large application is actually a complete system for the semi-automatic construction of digital circuits based on the interactive drawing of the logical components of the circuit.

STARFIT

Optical astronomers use their telescopes to take pictures of the heavens. The most interesting things in these photographs are things that change from one picture to the next. This can be because the photographed object changed its position as is the case with comets or because the object changed its intensity without moving as is the case with variable stars. The astronomy department of Nijmegen has an aparatus for comparing photographs taken of the same part of the sky at different times. By quickly switching from one photograph to the other objects with different intensities will appear to blink. Once these blinking objects have been discovered their intensity in a long history of photographs can then be measured.

This intensity data taken from different photographs at known times can then be plotted on a time axis. Some trial hand calculations and visual inspection will quickly reveal if the intensity variation is periodic. If the data has some promise of periodicity the astronomer attempts to determine the period by a few hand calculations involving either the peaks or the valleys, whichever are scarcer or are represented by the most reliable data. The trial period is then used to slice the time axis into regular intervals and then plot all the data on a graph of one period interval so that all the periods lie on top of each other.

From his first trial plot the astronomer can then make another guess at a better value of the period and then recalculate the interval position of each point to get a new plot. Since each plot is typically a half a days work, a strong desire to automate the procedure can be easily understood. This is exactly what was done for the STARFIT program. The astronomer types in the value of the period he wants to try and the program makes the plot.

The major problem encountered in this application is that the quality of the data can vary from point to point since each point comes from a different photograph taken on a different night. This means that there

can be variations in the atmospheric conditions, the quality of the photograph and even the position of the object on the photograph. Objects near the edge of a photograph are more subject to distortion. To compensate for this variable quality of the data, each data point is encoded as good, medium, or had to allow for differing representations in the period plots.

After the program has read in the data for a star and the astronomer has made a plot with his first trial period, the program provides the following possibilities for interaction. The astronomer can simply type in a new period and get a new plot. He can also specify an amount to change the period, so that he can step thru a range of possible periods in small increments. By stepping backward and forward he can compare two nearly identical periods to see which he likes better, after taking the quality of the data points into consideration. The stepping process can even be made to go automatically either forward or backward with a time of about 5 seconds between each new plot.

To provide the astronomer with a measure of how well a given period groups the data points into a nice curve the program can be asked to make a chi-squared plot of an up to 6th degree polynomial curve which best fits the data. Indeed the program even goes so far as allowing the polynomial curve to be recomputed at each step of automatic stepping.

At this point a reasonable question is how about using some automatic technique for finding the period. Because of the variable quality of the data most attempts at this have not been very succesful. Using the program we attempted to let the chi-squared good run of fit measure determine a direction to seak the next possible period in a valley searching technique. This was sometimes useful for obtaining higher precision in an already known perid. Another unrelated technique that was attempted as an initial search technique was simple Fourier analysis. This never yielded any profitable results as far as I know.

The program runs best on the vector general since the screen must be erased between each plot. Even at 9600 baud the repetitive redrawing of the screen in auto-step mode would be very tedius on the tektronix.

The major discovery from a graphics point of view was that even using the program it was very difficult to determine the period of a star without first doing some hand calculations. Beacuse of the technique of plotting intervals of the five axes on top of each other, a small change in the proposed period can make a big difference in the plot. Thus you cannot

gradually step toward the best period but must start fairly close to it and scan a range of periods to get a better period.

PRISMA - PLOT

To study the structure of matter physicists measure the results of elementary particle collisions. One of the most common ways of obtaining a measureable collision is to bombard a tank filled with liquid hydrogen with a stream of high energy particles. Typically such particles as protons and pi-mesons are used for the bombardment. The target particles are the numclei of the hydrogen atoms in the tank which are simply protons. Thus a typical interaction would be a pi-meson hitting a proton which would result in two pi-mesons and a proton after the collision.

The particles resulting from the collision may be either changed or unchanged. A changed particle moving thru the liquid hydrogen causes the formation of a trail of bubbles in the liquid hydrogen. This trail of bubbles can be photographed. The physicists want to measure the physical properties like energy and momentum of the particular resulting from the collision. To get information about the speed, mass, and change of the resulting particles a magnetic field is applied across the bubble chamber. Since a changed particle moves in a circle in a magnetic field the properties of the changed particles can be deduced by measuring the curvature of the bubble tracks.

The problem is that the interaction of the elementary particles during the collision does not always occur in the same way. So even though the bombarding particle and the end resulting particles are the same the momenta and energies of the resulting particles are not the same and indeed the collision interaction mechanism is not the same. During the collision interaction different short lined elementary particles (also known as resonances) may be formed. These resonances quickly decay into the stable end particles of the interaction. In order to separate the collision events by their interaction mechanism and thus to be able to study just one interaction mechanism at a time, values of variables derived from the basic energy and momentum data are plotted in multidimensional spaces. In these multidimensional spaces where each point represents a single collision event, clusters of points indicate collisions which resulted in the same interaction mechanism. Thus a principle step in the data analysis is to

separate the data points into clusters in multidimensional space.

One way to search for clusters is to plot histograms of various primary and certain desired variables. A more direct way is to plot the values of several variables againts each other in what are called prism plots. These prism plots are the primary display of this program. From observation of the prism plot the physicists can discover which points lie in clusters and then use these points in a certain cluster for making histogram plots.

The current program reads a data file which contains the values of 8 variables for each measured collision event. Any 3 of the variables may be chosen for making a three dimensional point plot on the Vector General. The transformation hardware of the Vector General can then be used from GPGS for rotating the dot cloud in order to search for three dimensional clusterings. Histograms can be requested for any variable of the file with events selected so that the value of another variable lies in a specific range. Two dimensional scatter plots of two variables against each other may also be requested.

Another technique for detecting clustering of the data points is to compute and display a Minimum Spanning Tree (MST). A minimum spanning tree is computed by computing the distance (according to any metric defined on the 8 variables) between each point and all the other points. These distances are then sorted into an ordered list with the shortest distance first. Beginning with the shortest distance between 2 of n points the connections between the first n-1 points that do not form a closed circuit are chosen for the minimum spanning tree.

The program for dot cloud with the selected histograms runs on the IBM version of GPGS using the Vector General via the PDP satellite. During a typical half hour use of the program on this configuration the program used 28 seconds of CPU time send 5400 messages over the satellite link and required 176K of IBM core. The satellite support on the Vector General can be equally well provided under RT-11 or UNIX. The minimum spanning tree program is still small enough to fit on the PDP in stand-alone mode. These programs are a good example of the portability of GPGS programs between the PDP and IBM.

DIGDRA

Semi-automatic Digital Circuit Construction

The electronics department at Nijmegen has the task to build special purpose apparatus in support of the research and teaching needs of the Faculty of Science. To accomplish this task the designs are prepared by a design department to be given to a drafting and detail department which makes instructions for the construction department. In the beginning this was all handwork ending with construction by means of hand wire wrapping the pins of the integrated circuits.

The first step toward automating this construction process was to replace the hand wire wrapping by an automatic wirewrapping machine guided by a punched paper tape. The paper tape is prepared by a computer program which accepts a list of named locations pins and generates instructions on the tape to join pins with the same name. This step of automization serves only to move the manual work of hand wire wrapping into manual work of writing a list of thousands of named locations to be keypunched as program input.

The Digdra program combines the two previous tasks of the drafting and detail department namely the drawing of the logic diagram and the preparation of the list of names. The additional task of assigning the integrated circuits to positions on the construction rack is still carried out manually but the positions are also incorporated in DIGDRA.

The Key concept in designing the DIGDRA system was to have a system that would be operational on a minimal hardware configuration. By matching the processing speeds of the hardware components of the system a powerful low-cost system was achieved. The system is based around a 24K PDP-11 computer with a disk and a Tektronix 4015 storage display. Since the program can retrieve the data structure from the disk just about as fast as the Tektronix can write it on its screen (even at 9600 baud), the core memory can be used exclusively for the FORTRAN program and a few disk blocks from the data structure representing the drawing.

When it is running the interactive DIGDRA program uses a menu file of TTL logic components common to all signal apparatus built at Nijmegen. For each apparatus there is a work file which contains all the specific components of that apparatus. In addition to the Tektronix 4015 graphic device the program also has a Tektronix 4661 plotter available to it.

Starting from a pencil sketch provided by the logic designer the draftsman prepares the logic drawings of the aparatus on a number of different pages of drawing. The pages are the unit of drawing within the aparatus. On each page the draftsman places logic components by assigning their local origin by means of the crosshair cursor of the Tektronix. The logic components (gates, flip-flops) each have a number of connection points or pins where they may be connected to. The connections are represented by lines going from pin to pin or by signal names. In addition to the technical information there is administrative information in a box in the lower right hand corner of the page.

The best descritption of the detailed capabilities of the program can best be given by listing the commands available to the operator of the program along with a short description of each one. We begin with the commands for administrating the creation of the work file and the administrative information for the information block in the corner of the page.

Command	Arguments and Action performed
Begin Page	Page number - Start drawing on a clean page
Head Apparatus	Apparatus identification information
Head Page	Page identification information
Delete Page	Page Number - Remove page from data
Copy Page	Copy all logical components and connections from one page to another
Display Index of Pages	
Make picture	Erase Tektronix screen and redraw the page that is currently being worked on.

In principle each apparatus is drawn in two phases using DIGDRA. First all the logical components and their connections by lines or names are drawn, then the draftsman assigns circuit board locations to the logical components. Since it frequently happens that an integrated circuit will contain more than one logical component, several logical components may be assigned to the same location and thus to different gates in the same chip. The allocation of the gates in the chip to the logical components is handled automatically by the program. Thus each of the logical components assigned to the chip receives a different set of xxx numbers. It would of course be

an error to assign too many logical components to the same physical component. The next set of commands to be listed will be for the manipulation of logical components and connections.

Command	Arguments and Action performed
Add Component	Component Code - Component is added to the current page with its origin at the intersection of the crosshair cursor.
Delete Component	Component Indicated by Crosshair cursor and all connections to the component are deleted.
Modify Component	Add additional drawing data to component
Add Line	Make a connection between two or more pins
Delete Line	Delete Connection
Name Right	Assign a name to be displayed to the right of the pin
Name Left	Assign a name to be displayed to the left of a pin.

Some of the most powerful drawing commands allow the program operator to manipulate groups of connected logical components. Such a group of logical components by connections we call a structure.

Command	Arguments and Action performed
The structure is moved from one place on the page to another.	
Copy Structure	A new copy of the indicated structure is created but without the components physical location codes.
Delete Structure	Indicated

The following set of commands deal with circuit bound position information or are multipin plugs which connect circuit bound to a cable or another circuit board. Each DIGDRA apparatus does not need to be made on one circuit board or rackbut can be made on several racks. There is the restriction, however, that all the logical components on a page must all

be on the same rack to keep the physical location information consistent.

Command	Arguments and Action performed
Add Position	Circuit Board position code
Extend Plug	Add more pins to plug
Reduce Plug	Take pins away from plug.
Modify Frame of Plug	
Pin Definition of Plug	Change pin numbers in plug
Ground Horizontal	Draw Horizontal Ground Symbol
Ground Vertical	Draw Vertical Ground Symbol
Rack number Change	Change the rack description for all the components on a page.

As can be seen above, each command makes a change in the displayable picture. Since the system is based around the storage tube for holding the picture, only additions to the picture are immediately apparent. Deletions are only noticeable after the "Make Picture" or refresh command is executed to blank out and redraw the screen. During normal operation of the program it is expected that the operator will be issueing mostly commands to add to the picture. This indeed turns out to be true. The operators work along until the screen becomes too cluttered with unwanted pictures before they ask for a redraw.

When a command to add a new component or connection to the screen is given the program adds to the data structure on the disk and to the picture on the screen without having to look at any other part of the data structure. Since the amount of work required to add to the picture has been reduced at the expense of the other functions of the program, the program runs efficiently from a computer point of view in that it executes the minimum number of instructions. It also runs efficiently from a human point of view in that it has the shortest reaction time for the most frequently used commands.

The commands issued by the operator to build the picture clearly correspond to operations on the data structure. For each logical component on a page there is a corresponding logical component block in the data structure. All the logical component blocks for a given page are in a doubly linked

list for convenience in adding and deleting them. The logical component blocks (LCB's) contain pointers to the menu block in the menu file that contains the drawing data for the component. The LCB component has been assigned to a location on the circuit board. By assigning the logical component to a chip location the operator requests the program to either find a physical component block that may already exist for that location or to create a new physical component block. There may be only one physical component block per location on the circuit board. Because each location assignment for a logical component requires a search of all physical components to see if it already exists for the specified location, there is an index block with the circuitboard locations and pointers to the corresponding physical component blocks for all the physical component blocks on each circuit board or rack.

Normal logical components like gates and flip-flops have a fixed number of pins, but the plugs by which the circuit board is connected to cables is another matter. The representation of a plug is simply a rectangle surrounding some numbered pins. The representations of plugs is so flexible that some pins of the plug may be on one page, and some on another.

The connections between the pins of the logical components are usually represented by lines for connections on the page and by giving two or more pins the same names for connections that go from page to page. Each name and drawn connection is represented in the data structure by a connector block (CBL). This connector block contains the name assigned to the connection, pointers to the LCB's of the connected pins, and the coordinate data necessary to draw the lines connecting the pins as entered by the operator when he drew the connection.

The connections must go from pin to pin and the logical components those pins belong to must be identified. To solve the problem of correlating the pin indicated by the crosshair cursor with the appropriate logical component block, a list of the pins and their screen locations must be searched. To reduce the amount of searching required, the screen has been divided into 16 sections (4 by 4) and a shorter list made for each section. By this technique the amount of searching has been reduced by an order of magnitude.

At the highest level of organization the data structure for each apparatus is maintained in its own file. Within this file the component, connector, and hash blocks for each page are linked to a page header block.

For the whole file there is an index of the pages which points to the page header blocks and a single space management mechanism. Since all the blocks in the file are potentially variable in length the space management mechanism is an essential part of the program.

Initially all the file is unused available space. As the operator adds to the drawing the space for the new blocks is taken from the unused available space. When compontents or connections are deleted the blocks are put on available space list according to how big they are. Blocks are only allowed to have lengths that are expressable as powers of two. Thus there are only eight or nine possible sizes of blocks. It also means that when a block grows over a boundary it is moved to a new block that is twice as big.

To draw the picture represented by the data structure the program scans the logical component blocks and their associated menu blocks to draw the components. The pin numbers in the components come from the associated physical component blocks. After scanning the logical component blocks the program scans the connector block for the coordinates of the connecting lines. The administrative data in the block in the lower right corner comes from the page and apparatus header blocks.

The major non-graphic output of the system is the list of connections obtained by scanning the connector blocks, which is used as input to the program that generates the tape for the automatic wire-wrap machine. It is in fact this extra output that makes the use of the system worthwhile in the eyes of the operators.

CONCLUSIONS

Each of the three applications described here runs easily with GPGS. The features of GPGS that have proven to be the most useful are device independence and the existence of multiple GPGS applications. The device independence has made the addition of plotting functions for interactive programs a trivial task. The device independence has also allowed programs to be run on secondary devices when the primary device has not been available. The multiple implementation have allowed programmers to begin with smaller programs on the once more accessible and easier to use PDP-11 and then move there programs to the IBM when their core requirements outgrew the space available on the PDP. Indeed the size of an applications program

has proven to be the single largest factor which implementation and which computer was to be used for the program.

ACKNOWLEDGEMENTS

STARFIT - H.J. Thommassen of the Graphic Group S.L.T.J. van Agt of Astronomy.

PRISMA-PLOT - E.W. Kittel, J.D. Schotanus, and D. Weenink of High Energy Physics.

DIGDRA - Ch. A. Timmer of Electronics.

THE "TROTS SOFTWARE PACKAGE FOR THREE-DIMENSIONAL RECONSTRUCTION"

A.H. VEEN
Biology Department
University of Pennsylvania, Philadelphia, U.S.A.

1. INTRODUCTION

As an example of the use of computer graphics in an application directed environment, this paper will discuss a software package for the three-dimensional reconstruction of biological data. This package was designed in 1973 for use at the Biology Department of the University of Pennsylvania in Philadelphia (U.S.A.) in cooperation with Lee D. Peachey. A more substantial description can be found in [1].

In the study of structure, particularly that of biological specimens, it sometimes is useful to slice the object being studied into a series of slabs. These then are examined one at a time in face view, that is, in direction perpendicular to the plane of the slab, and information extracted from these views is used to construct a model of the original object. This process often is called serial sectioning and reconstruction. Usually the model is manually constructed from plastic, wood or other suitable material and requires careful and tedious cutting, sawing and glueing. All though the resulting model in many cases greatly clarifies the structure of the specimen, the amount of time consumed by this manual method has kept it from wide spread usage.

The direct motivation for the development of a new method (and its first test case) was a study in our laboratory by C.H. Damsky of the mitochondria of yeast cells. This study needs to be done on an electronmicroscope and for this purpose many very thin slices have to be cut (typically 20-100 per cell). As a result not much more is known about the spatial structure of these organelles as that it is very complex. It wasn't even known whether all the different parts within a slice belong to one or to several structures.

We have developed a computer graphic method for generating precise three-dimensional representations using the kind of structural information available from microscopic images of serial slices. The software package for this method is known under the name TROTS which stands for Three-dimensional Reconstruction of Objects from the Tracing of Slices. The main objectives in designing the system were:

- only readily available hardware should be used
- the operating costs should be so low that it could become a standard research tool
- it should be easy enough to use so that biologists without computer experience can get meaningful results without the assistance of a separate operator and after only a minimal instruction period.

2. THE PROCESS OF RECONSTRUCTION

Two basic tasks must be accomplished to solve the problem of three-dimensional reconstruction. The first is to recover the position and shape of the two-dimensional projections of an original three-dimensional structure that has been cut into a series of slices of finite thickness. Secondly, an accurate description of the structure must be reassembled from these projections. These tasks can be broken down into a series of sequential steps.

2.1 Preparing the Visual Data

Preparing the tissue, cutting slices of only a few hundred angstrom thick, and taking the electron microscope pictures are the most elaborate, dificult and time consuming steps. I have neither the intention nor the ability to discuss these in any detail and I will simply assume that a reliable set of micrographs of known magnification has already been obtained from a set of tissue slices of known sequence and thickness. Each micrograph represents the cross-sections of a number of structures. The perimeter of each cross-section is, in essence, a profile of that structure.

A major problem in reconstruction is to know, once the specimen has been sliced, how sequential pairs of slices fit together. This is the so-called problem of *alignment*. In many cases this is handled by simply sliding the replicas of the two slices until there is a best fit of the profiles of the structure being studied. This method carries the danger that the final reconstruction will be biased by some preconceived notion of its form. That's why we have paid a lot of attention to the use of "fiducial marks". These can be lines or points whose position or orientation one has reason to believe will remain fixed from one slice to the next. It is essential that these features used for alignment be precisely locatable, and usually they should be external to the structure being reconstructed. Sometimes the best solution is to introduce artificial fiducial marks during preparation of the tissue. Depending on the nature of the specimen these can be plastic fibers, micro tubules (cylindrical subcellular structures) or holes drilled by a laser beam.

2.2 From Visual to Computer Data

Once the profiles to be used in the reconstruction have been found, data must be gathered on their relative positions and shapes. This is done, in our system, by an operator tracing manually around each profile in each micrograph with a stylus linked through an A/D interface to an appropriate computer. This combines the pattern recognition and selection power of the human operator with the arithmetic power of the computer. This tracing method looks primitive at first, but in my opinion the simplicity of the approach is essential to the success of the system. The facts are that cell

images are subtle, variable and complex which only by specially trained observers are easily analysed. Developing a pattern recognition program would be a major undertaking, would have to be rewritten for many different applications and would need an elaborate control input to indicate to the program what is to be recognized and what is to be included in the reconstruction. Another advantage of the biologist doing the manual tracing herself is, that she can use all her special knowledge of the structure to directly influence the accuracy of the reconstruction.

In our implementation the tracing is done using a simple mechanic-electrical tracing device (see Fig. 1) connected to our laboratory computer.

2.3. Transfer to the Time Sharing Computer

The tracing data which have been gathered through the laboratory computer, are transferred by means of a paper tape to the medium scale computer where the rest of the process is carried out. The reading of the tape is under control of program PAPER of which Fig. 2 gives a sample dialogue. The user can specify global parameters as magnification and the thickness of individual slices. The program does some preliminary checking and processing of the data. An automatic transfer is provided to the next program.

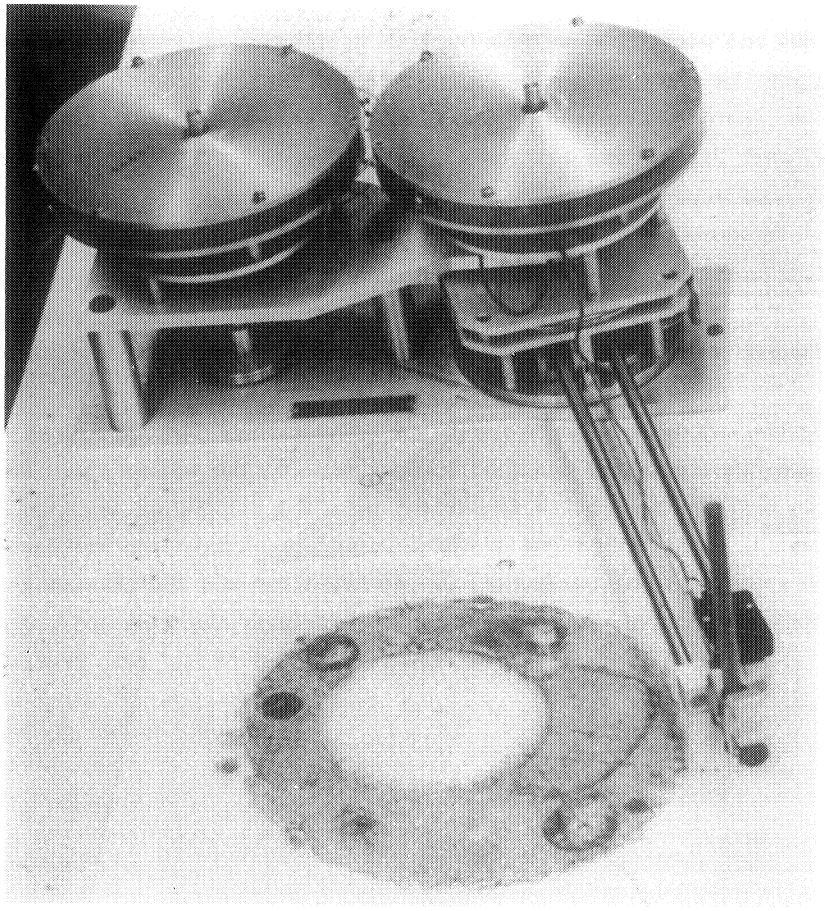


FIGURE 1

Our tracing device is marked by simplicity and low-cost. The tracing stylus is at the end of a lightweight bar which rotates around and slides in and out of a fixed construction incorporating high precision potentiometers to translate angular and radial position of the stylus into electrical signals. The mechanical design is such that the stylus moves smoothly, without backlash, and with no preferred direction so that curves can be traced easily and accurately. A microswitch is positioned near the stylus so that one hand can both trace and operate the switch to give extra signals to the program. The positions of two reference points marked on the table are included in the tape at the start of the session.

```
.RUN PAPER

TROTS-PROGRAM PAPER
FOR HELP TYPE "?"

SPECIAL OPTIONS ?    ?
     IF YOUR ANSWER IS NO I WILL TAKE THE MOST USUAL ROUTE  THROUGH  THE
     PROGRAM  AND SKIP MOST OF THE QUESTIONS.  I WILL USE THE FIRST FIVE
     CHARACTERS OF THE TITLE AS THE FILE NAME.  IF YOU HAVEN'T SPECIFIED
     THE NUMBER OF SLICES ON THE TAPE I WILL ASSUME THAT THEY ARE ALL ON
     ONE TAPE AND THAT THEY ARE OF THE SAME  THICKNESS.   AFTER  READING
     THE TAPE I WILL AUTOMATICALLY REDEFINE THE FRAME IF ANY POINTS WERE
     FOUND OUTSIDE.
     IF THIS IS NOT WHAT YOU WANT TYPE YES IN RESPONSE TO THIS  QUESTION
     AND I WILL TAKE YOU THROUGH THE WHOLE DIALOGUE.
     DEFAULT = NO

SPECIAL OPTIONS ?    YES
FILE NAME ?      SIDE
NUMBER OF SLICES ?   2
SKULL
SIFT NOISE ?     NO
FRAME SIZE (CM) ?    25
PRINT ?__
MAGNIFICATION ? 1
ARE SLICES OF UNIFORM THICKNESS ?    NO
TYPE THICKNESS IN CENTIMETERS
 1? 1.8
 2? 2
READING PAPERTAPE
OUT OF BOUNDS. X=    -0.024 Y=    0.587 SLICE=    1 PROFILE    1
BOUNDARIES: X FROM  -0.0603 TILL    0.8649
            Y FROM  -0.0058 TILL    0.5867
REDEFINE FRAME ?     YES
XLEFT, XRIGHT, YBOTTOM, YTOP ? ___
DO YOU WANT TO RUN EDIT  ?      YES

TROTS-PROGRAM EDIT
FOR HELP TYPE "?"
```

FIGURE 2

A sample of the dialogue generated by PAPER with user responses underlined. This user asked for more information on the first question causing the file containing the TROTS manual to be scanned and the appropriate information to be displayed. Apparently the title ("skull") was on the tape but the number of slices not. The options to print all data and to delete readings that came apparently from noise in the tracing hardware were not selected. The user initially chose a 25 cm frame but points were found outside so it asked for a redefinition of the frame and left it up to PAPER to choose the best boundaries. He uses the chain option to transfer to the next program EDIT.

2.4. Visual Editting

Before the data are ready for three-dimensional reconstruction some data manipulation is usually required such as alignment and "cleaning" of the data. This is done on a Tektronix storage tube under control of the highly interactive program EDIT. Fig.3 gives an impression of its options. Using the *INPUT command the user can select two slices, which need to be realigned with respect to each other. One will be the "active" slice which will "slide" over the other one which will be used as reference only.

The sliding can be done with the use of the more primitive commands *AXIS, *TRANSLATE and *ROTATE. Feedback is given by the *DISPLAY command, which carries out the transformation on the active slice, using the standard 2-D matrix multiplications for each point, and writes on the screen a display as in Fig. 4a. In this example the pair of lines and the pair of big circles are the fiduciary marks and the aim of the alignment is to get these to coincide as well as possible. To obtian the best fit, the user will have to iterate the *T, *A, *R command to specify a transformation and an *D command to get feedback on his progress. More automatic procedures have been divised so that the user only has to indicate his goal and leave it to the computer to calculate the precise transformation. An example of the *LINES command can be seen in Figure 4. Another powerful positioning option is using the cross-hairs with the *HAIR command. With two knobs the cross-hairs displayed on the terminal are positioned at different slices and the program calculates the translation and rotation to bring the selected points into coincidence.

In our application the basic assumption for alignment is that the various profiles within one slice are in a fixed relationship to each other. For other applications it might be useful to treat profiles separately. The *MODIFY command singles out one profile so that subsequent transformations only apply to the selected profile leaving the rest of the slice alone. For even more detailed manipulation, one can use the *CRACK command to break a profile in two so that the two parts can be treated separately. Fiduciary markers that are not being used anymore can be removed by the *ZAP command. Pictures taken at different magnifications can be adapted to each other with the *SCALE command.

A sequence of commands can be stored as an *GLOBAL command string that is automatically executed whenever a new slice is brought into core

```
*?

TYPE IN THE COMMAND YOU WANT HELP WITH
THEN HIT RETURN
FOR LIST OF COMMANDS TYPE "?"

??? ?

TYPING
 A .5,.4      SETS THE ROTATIONAL AXIS
 B 6          BLUNTS WITH LINEAR SMOOTHING USING 6 POINTS
 C            CRACKS A PROFILE
 D            TRANSFORMS AND DISPLAYS
 E            OUTPUTS AND EXITS
 F            FINDS A PROFILE WITH AID OF HAIRLINES
 G            INPUTS GLOBAL COMMANDS
 H            MANIPULATES WITH AID OF CROSHAIRS
 I 3, 4       INPUTS SLICE 3("PASSIVE") AND 4("ACTIVE")
 J 5          JOINS PROFILE 5 WITH THE NEXT ONE
 L            LINES UP WITH AID OF LINES AND PROFILES
 M 3          OPENS PROFILE 3 FOR MANIPULATION
 N            INTERPOLATES
 O            AS D BUT OVERWRITES EXISTING DISPLAY
 P            SELECTS POINTS, FINDS DISTANCES AND ANGLES
 R 52         ROTATES ACTIVE SLICE CLOCKWISE BY 52 DEGREES
 S .01,-.03   EXPANDS ACTIVE SLICE 1% IN THE X-DIRECTION
              AND CONTRACTS IT 3% IN THE Y-DIRECTION
 T .20,.05    TRANSLATES ACTIVE SLICE IN X AND Y-DIRECTIONS
 U 1,2,3,4    CALLS USER SUBROUTINE
 W 2          WRITES LABELS ON PROFILES IN ACTIVE SLICE
 Z 4          ZAPS PROFILE 4 OUT OF THE ACTIVE SLICE
```

FIGURE 3

List of currently available commands for EDIT. The first question mark typed by the user makes EDIT enter the assistance section. The second question mark produces the list.

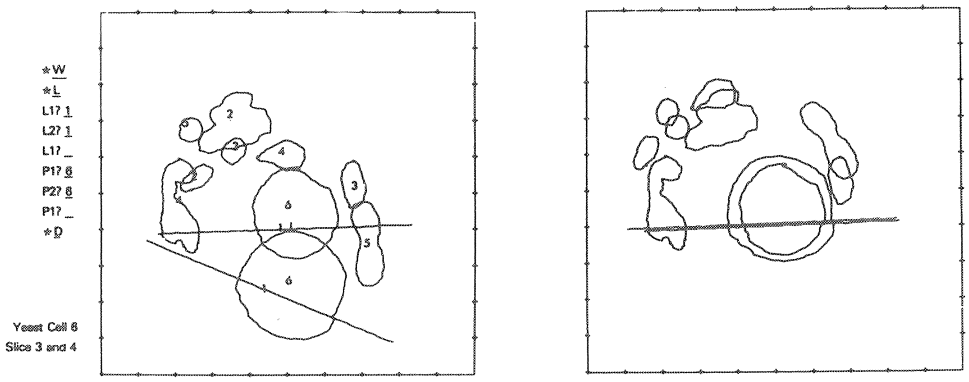


FIGURE 4a (left)

Editing of two serial sections of a yeast cell (user input underlined). The smaller outlines are profiles of mitochondria, the organelles whose 3-D configuration is being studied. The big outlines marked "6" are the profiles of a vacuole, a large organelle extending through all the sections, and useful as fiducial marks for positional reference. The lines marked "1" are fiducial marks coming from lines drawn on the original photographs pointing towards other cells and thus reliable directional references. The user first labels the profiles with the *WRITE command and then specifies with the *LINES command that lines "1" should be used for directional and profiles "6" for positional alignment. A total of 20 of such pairs could be specified.

FIGURE 4b (right)

The *DISPLAY command produced this picture. EDIT calculated the best fit possible minimizing the residual deviations.
(Preparation by Dr. Caroline Damsky).

with the *INPUT command. Since an *INPUT command can be the last of an *GLOBAL command string we can call the string recursively. For example,

```
*G
 G ? T  .1, -.2
 G ? Z 1
 G ? I
```

is a recursive G-string that can be initiated ny merely inputting the first slice (*I 1). The recursive execution of the g-string will now cause all slices to be translated by .1 to the right and .2 downwards and to lose their first profile. In this way, simple editing procedures can be written and stored.

Other commands perform measurements (thanks to Dr. Larry Kerr) or aid in the identification of profiles, while different ways of data reduction and smoothing can also be specified. Any user who is still not satisfied with this list of commands can, with the *USER command, transfer control to his own subroutine tailored to his personal requirements.

2.5 Three-Dimensional Reconstruction

When the user is satisfied with the editted data he can enter the 3-D section consisting of the programs STRIP and FIG 3D. The first of these does the preprocessing to convert the data into a special format required by the efficient 3-D and hidden line algorithm in FIG 3D. In this last program the user specifies a viewing position and some other parameters that determine the appearance of the output. The 3-D figure is now written on the screen or on an incremental plotter. Examples of the output and the influence of the different depth cues can be seen in Fig. 5 and 6. The reconstruction and display goes fast enough (some 40 seconds for FIG. 6 under normal load and baud rate) to make it practical for the user to create reconstructions from many different viewing positions and select the most clarifying view. With the program MERGE it is even possible to remove slices, which is useful when the internal structure is hidden by outer layers.

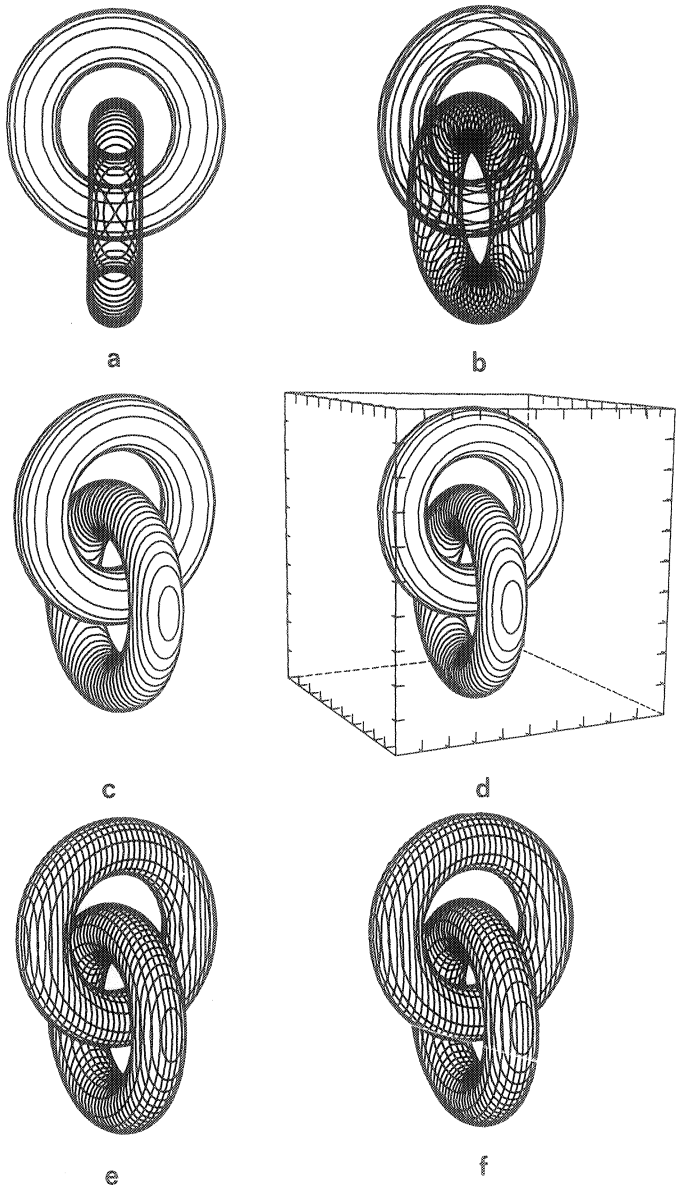


FIG. 5
Effectiveness of the different depth cues. These data do not come from tracings but are mathematically generated by our program DONUT. The shapes are chosen to illustrate how TROTS can handle holes and multiple interlocking parts.
a. Head-on view.
b. View from a more advantageous angle after 3-D transformation.
c. Removal of the hidden lines providing a very powerful depth cue.
d. With perspective, i.e. close-by parts appear larger than more distant parts.
e. After "cross-hatching" with program CROSS.
f. Forms a stereo pair with e. Stereo angle is 4 degrees.

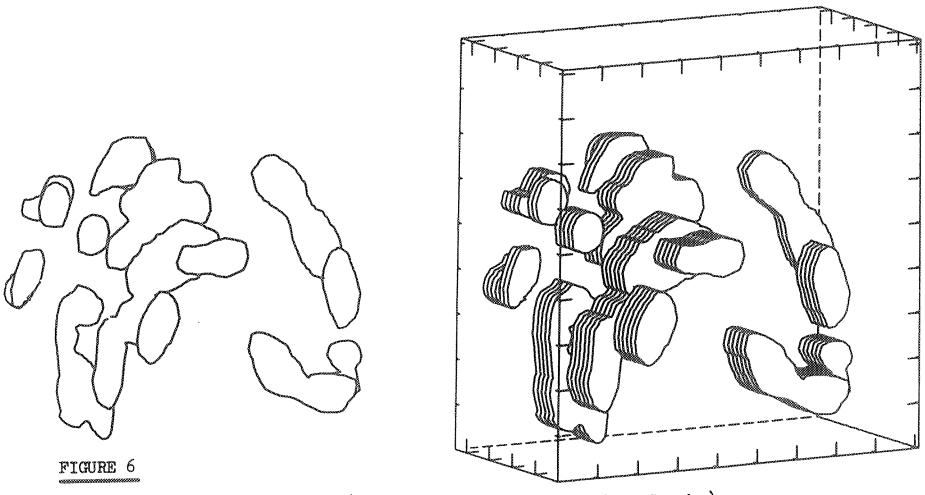


FIGURE 6

Mitochondria of a yeast cell. (Reconstruction by Dr. Caroline Damsky).

a - Has the same depth cues as Fig. 5c but is still very ambiguous. We have no idea of the angle from which we are looking at the object nor of its thickness. Since we are dealing with unfamiliar shapes, i.e. shapes that do not resemble any object we are accustomed to see in 3-D, one needs special depth cues.

b - Our layering technique gives the impression that the slices are cut from some layered material and provides a pseudo shading effect. Each layer or "sheet" is of the same thickness so thicker slices have more sheets. On the program level it is accomplished by simply displaying the slice repeatedly with a displacement in the z-direction equal to the sheet-thickness.

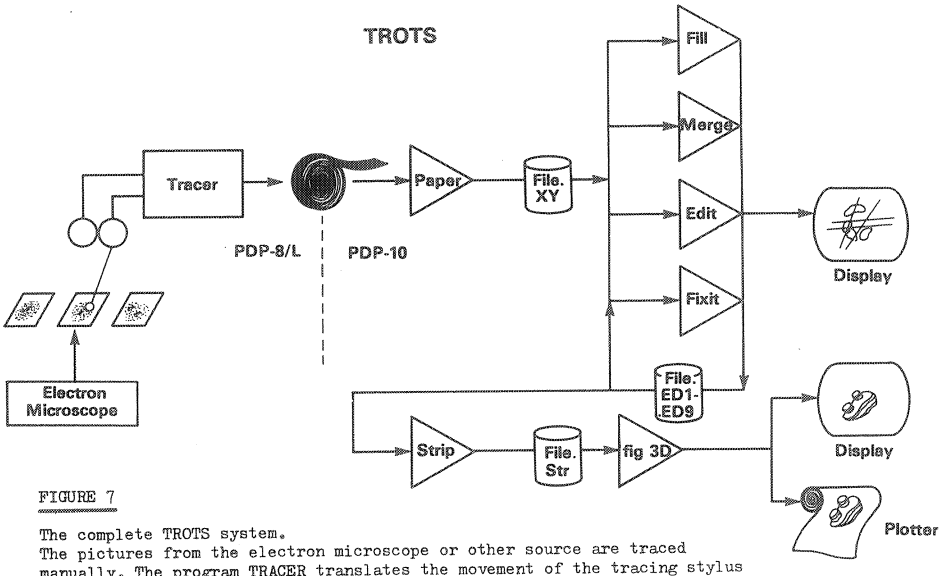


FIGURE 7

The complete TROTS system.
The pictures from the electron microscope or other source are traced manually. The program TRACER translates the movement of the tracing stylus into a paper tape. PAPER converts this into a disc file. The programs FILL, MERGE, EDIT and FIXIT can be used to perform a variety of data manipulations. STRIP prepares and FIG3D produces the three-dimensional reconstruction.

3. THE HARDWARE

Since one of the objectives in designing TROTS was to come to a working system within a matter of months without having to deal with the extra costs and delay for the purchasing of special hardware, the available hardware was, at least in part, determining the final set-up of TROTS. If, for instance, a refresh scope had been available instead of a storage scope, a simpler more efficient EDIT program would have been written. An advantage, however, is that the hardware we are using is, with the exception of the tracing device, fairly typical for a research environment.

The local computer in our laboratory is a most simple PDP8/L, with 4K of memory, teletype and an A/D converter. A limited machine indeed which TROTS only uses for data gathering. Doing this on a time-sharing computer by connecting our tracing device directly, would require special input buffer arrangements, might endanger the integrity of the time-sharing computer and would not be greeted with great enthusiasm by most computer managers. The paper tape interface we use presents no such problems at all.

Our tracing device (see Fig. 1) is designed and built locally by our laboratory shop. At the time this was the only way to get an input device, smooth and accurate enough, for only a few hundred dollars. It might look primitive but it still operates to our satisfaction.

Most of TROTS is implemented on a PDP10 which has an excellent time sharing monitor and good file handling facilities. Among the peripherals are a number of Teltronix 4010 graphics terminals (with storage tube), hardcopy unit and incremental plotter. Maybe most important of all, this set-up is supported by an enthusiastic staff interested in graphics.

4. THE TROTS SOFTWARE

The software for TROTS has been written as a modular package (fig.7). The modules are independent programs communicating with each other through permanent storage data-sets. The advantages are that intermediate results are always available to repeat a step or correct a mistake, that adding a new module does not affect the rest of the system in any way and that it is possible for a user to enter the chain or leave it at any point he wants. For example, some applications use only the data collection part

(programs TRACER and PAPER) as a means of converting the visual data into computer files for further calculation. In other cases, three-dimensional data that are generated by other means (as by our program DONUT for figure 5), can be displayed with hidden lines removed using the three-dimensional programs STRIP and FIG 3D. The functions of these programs are summarized in Table 1.

Trots makes extensive use of the excellent file handling facilities on the PDP-10. A file name consists of two parts: the first name and an extension. The latter generally is used to indicate the type of data stored in the file. In the input and output columns in Table I the convention is shown for the file name extensions that the different programs assume. The series xxx.XY, xxx.ED1, xxx.ED2 till xxx.ED9 forms a hierachy and if an extension is not explicitly specified TROTS programs take the highest one available for input and the next one for output. All file handling is controlled by subroutines from the TROTS library DSK10.

4.1 TRACER: Gathering the Data

The program TRACER running on the PDP-8 ensures that the limitations of the small computer and its lack of fast peripherals does not interfere with an efficient tracing process. There is of course a conflict between the high rate at which the tracer delivers information during tracing and the low rate of the paper tape puncher (10 BYTES/SECOND). To resolve this conflict TRACER calculates from each reading (averaging four successive samples to reduce noise) the angular and radial displacement with respect to the previous recorded point and records them only if either exceeds a threshold set by the user. Thus the number of collected data points is not dependent on time but only on the distance the stylus has moved and the speed of the operator has no effect on the accuracy of the paper tape. An additional advantage of this approach is that the recorded points are more or less equally spaced along the traced line. Furthermore input and output handlers operate independently of each other and communicate through a circular buffer. This way always enough room will be available unless the input part has gotten more than 3600 points ahead, a feat no user has yet accomplished while tracing carefully.

Points are only recorded while the operator keeps the micro-switch closed. So only the profiles itself are included on the tape. Changing

TABLE 1

Program Modules of TROTS

Program	Function	Input	Output
TRACER	Data collection. Monitors profile tracing.	Tracing stylus	Paper tape
PAPER	Check tape, convert from polar into normalized Cartesian coordinates.	Paper tape	xxx.XY
EDIT	Alignment, smoothing, data reduction, etc.	xxx.XY, .ED1-8	xxx.ED1-9
*FILL	Scale data to fill screen.	xxx.XY, .ED1-8	xxx.ED1-9
*FIXIT	Corrects user errors.	xxx.XY, .ED1-8	xxx.ED1-9
*MERGE	Combines specified slices from 2 files	xxx.XY, .ED1-8	xxx.ED1-9
STRIP	Prepares data for hidden line removal and 3-D display.	xxx.XY, .ED1-8	xxx.STR
FIG3D	3-D display with depth cues.	xxx.STR	Graphics

* These "utility programs are relatively straightforward and will not be described in detail.

slices (next picture) is signaled through the keyboard. Both signals (end-of-profile and end-of-slice) cause special markers to be punched in the tape.

4.2 PAPER: Transfer to the PDP-10

The paper tape is read into the PDP-10 under control of program PAPER. In fig. 2 an example of its dialogue was already given. This demonstrates a primary design goal of TROTS: to be friendly, self explanatory and easy to use by a group of people who are normally not accustomed to computers. Reasonable defaults have been defined for most questions so that the unsophisticated user who only wants a standard picture has to make a minimum of decisions. In general, when a question is not understood by the user or deals with an option he is not interested in, simply hitting the return key will cause the program to either skip the option, substitute a reasonable answer, or provide the user with more directions. The latter also happens whenever a question mark is typed. The detailed directions are extracted from a file, which contains the complete manual for TROTS.

The rest of the package works with its own spatial unit chosen such that all data are normalized between 0 and 1. PAPER calculates the magnification factor to relate these units to centimeters. This factor is recorded in the file and gets automatically adjusted with every relevant manipulation. Data are presented to the user either in centimeters or microns depending on the magnitude of the magnification factor. Using the two reference points, (see fig. 1), the readings on the paper tape are transformed into polar coordinates and from these into Cartesian coordinates in box units. These normalized coordinates are compacted (two coordinates to a word), buffered, and written into the file. Slice thickness is included in the output file for later use by the display program.

4.3 EDIT: Visual Data Manipulation

Detailed data manipulation is done with program EDIT. Together with the tracing, operation of this program largely determines the quality of the reconstruction. The power of the program is in the flexibility of the conversation, the great number of commands (fig. 3) and its constant feedback. The programming for most parts is relatively straightforward, with the exception of the input command. This command (fig. 8) is very straight-

```
*?

TYPE IN THE COMMAND YOU WANT HELP WITH
THEN HIT RETURN
FOR LIST OF COMMANDS TYPE "?"

??? I
*I M,N
        M, N POSITIVE INTEGERS
        SLICES M AND N ARE READ FROM THE DISC, M AS THE FIXED  (PASSIVE)
        SLICE AND N AS THE SLICE TO BE WORKED ON (ACTIVE)
        IF M AND N ARE IDENTICAL TO THE CURRENT PASSIVE AND ACTIVE SLICE
        NUMBERS  THE  CURRENT  ACTIVE  SLICE WILL BE "FORGOTTEN" AND THE
        ORIGINAL ONE WILL BE READ FROM THE DISC AGAIN.  THIS  IS  USEFUL
        IF A SERIOUS MISTAKE HAS BEEN MADE IN EDITTING THE CURRENT SLICE
        AND YOU WANT TO START OVER AGAIN.
*I
        THE ACTIVE SLICE WILL BE REDEFINED AS THE PASSIVE SLICE AND  THE
        NEXT  SLICE  WILL BE READ FROM THE DISC AS THE NEW ACTIVE SLICE.
        IF THERE IS NO NEXT SLICE THIS COMMAND WILL CLOSE THE  FILE  AND
        EXIT THE PROGRAM.
*I M
*I Ø,M
        M POSITIVE INTEGER
        SLICE M WILL BE READ FROM THE DISC  AS  THE  ACTIVE  SLICE.   NO
        PASSIVE SLICE
*I M,N,J
        M, N, J POSITIVE INTEGERS
        ONLY ONE OUT OF EVERY J POINTS WILL BE  READ  INTO  THE  BUFFER.
        THIS  CAN  BE  USED  TO  SHRINK A LARGE DATA FILE WITH AN EXCESS
        NUMBER OF DATA POINTS TO A REASONABLE SIZE.  COMBINE  THIS  WITH
        THE  *G COMMAND TO GET AN AUTOMATIC DATA REDUCTION FOR THE WHOLE
        FILE.  FOR INSTANCE TO REDUCE A FILE BY A FACTOR 4 (RETAIN  ONLY
        ONE  OUT  OF  EVERY  4  DATA  POINTS) GIVE THE FOLLOWING COMMAND
        SEQUENCE AFTER INITIAL DIALOGUE:
                *G
                G?IØ,Ø,4
                *IØ,1,4
        THIS WILL READ THROUGH ALL THE SLICES AND EXIT  THE  PROGRAM  IN
        THE USUAL MANNNER.
```

FIGURE 8

Assistance offered by EDIT for the *INPUT command. After the user indicated that he wanted help with the *I command, the file containing the TROTS manual was searched and all information pertaining to this command displayed.

forward in its use, but complex in its consequence. Recovering from near disastrous mistakes is possible by simply restating the last *INPUT command, which inputs the data again in unaltered form. In all other cases the active slice currently in core will be written onto the output file. The user can specify any slice as the active and any other or none at all as the passive slice. Because of its flexibility, the lining-up can be in any direction and any two slices can be compared to attain a better accuracy. The zapping and data reduction features might lead to an output file that is smaller than the input file. To support all these features EDIT reads and writes all files sequentially, maintains intermediate files (transparent to the user), but does extensive bookkeeping to minimize I/O time.

All not defined commands transfer control to a user defined subroutine (if present). In this subroutine the user programmer has access to all data necessary for any measurement or action on the active slice.

4.4 STRIP - Converting to Strip-format

The problem of obtaining the 2-D projection of a point in 3-D space as seen from a specific distance and angle has a rather straightforward solution (see for example [4]). It can be written as a matrix transformation and needs a minimum of nine multiplications and two divisions per point. The problem of deciding whether a certain line of the 3-D object is visible or is hidden behind other parts of the object is more complicated. Many solutions have been prepared but all of these so-called hidden line algorithms consume considerable amounts of computer time and core storage.

Making use of the ordering of our data, I devised a special format for describing the individual profiles, such that both a simple hidden line algorithm can be used and the 3-D transformation simplified. The resulting 2-D display is the equivalent or a good approximation of the true projection.

Our hidden line algorithm is sufficiently general to place no restrictions on the number or the shape of the individual profiles so that the 3-D object can have holes and can consist of many different parts, as is already shown in fig. 5 and 6.

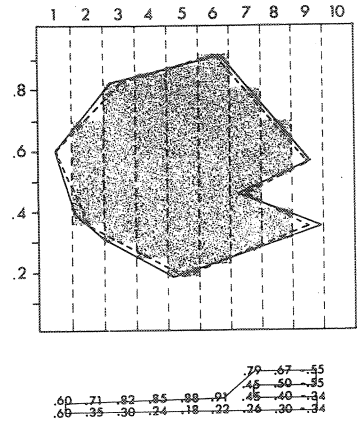
The feature of our data that makes this scheme practical is the fact that all the profiles are parallel and ordered: every profile has a unique z-coordinate, which is not less than that of the previous profile. One consequence is that from any viewing position (as long as we do not try to look

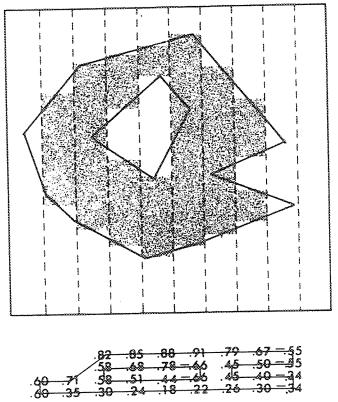
from "behind") any point can be hidden by previous profiles. So if the slices are displayed in order, each point has to be checked only against the scene as displayed so far in order to decide on its visibility.

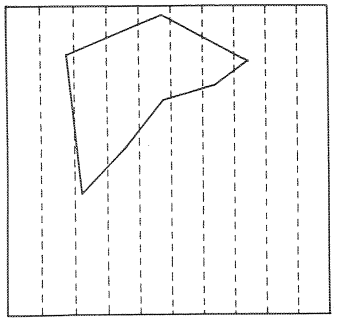
The simplicity of the hidden line algorithm and 3-D transformation is made possible by the "strip-format" into which the individual slices and the complete 2-D display are coded. The "stripping" process is illustrated by Figure 9a. For every strip the y-coordinate of all intersections are tabulated in order of increasing magnitude. If a profile changes direction within a strip, two equal y-coordinates are entered, positive for left turning points (as in <) and negative for right turning points (as in >). It is this coding of turning points that makes it possible to reverse the process and deduce which y-values in neighboring strips should be connected to get the original profile: going from left to right every left turning point is the start of a new pair of lines, while a right turning point is the closure of a pair of lines. Most importantly, it is possible to do this reconstruction serially from left to right, so that for every point in a strip, it is clear to which point in the previous strip it should be connected, without having information about strips further to the right. So after stripping, no information about the profile has been lost except for some accuracy in the x-coordinate.

All profiles are closed and have an "inside" and "outside", since one assumes that every profile is a borderline between the solid matter of the 3-D object and space and that the sections are large enough to completely cleave the object. As a consequence, there is always an even numbers of entries per strip, since every closed profile intersects a strip an even number of times. So every pair of points in a strip marks the limits of a covered section of the strip. Multiple profiles and holes within a slice (Figure 9b) are treated similarly.

The stripping algorithm in STRIP is simplified by processing a profile in two stages. In the first stage a "civilized" version of the xy-data is produced. The civilizing consists of averaging of successive points that fall in the same strip, generation of intermediate entries by linear interpolation if successive points are more than one strip apart, closing the profile by connecting the first and the last point, and removal of a possible overshoot. The latter occurs when the operator more than closed a profile by passing the point where he started tracing. To detect an overshoot, STRIP checks each point to see whether it falls within a specified distance from







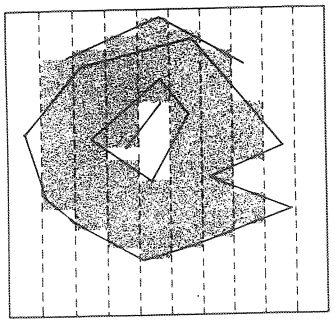


FIGURE 9

Strip format and hidden line algorithm. In the program STRIP slices are divided in vertical strips. For the sake of clarity, the number of strips here is 10 but is, in general, between 100 and 500. The accuracy of the method depends greatly on this parameter.

a – The solid line represents an arbitrary profile and the table its description in strip format. The line in this table is the connection path of these coordinates and is equivalent with the broken line in the figure. The shaded area is the portions of the strips covered by this profile according to this table.

b – One profile within another in the same slice defines a hole. The table is still in order of increasing magnitude and the segment enclosed by an odd entry and the next one is still the covered section of a strip.

c – The second slice and its strip format description.

d – Head-on view of the two slices of b and c with hidden lines removed. The table shows the contents of the cover-up array after merging. The equal entries for turning points do not show up in this array since they enclose no area. The "edge interpolator" in FIG3D eliminates any inaccuracy in the hidden line removal that might remain after increasing the number of strips.

the starting point (closing tolerence). In the second stage, the y-coordinates of the civilized version of a profile are distributed over the appropriate strips of the slice buffer, generating double enties (positive or negative) for turning points.

When all profiles of a slice have been entered, every strip is ordered according to increasing magnitude of y-values, and the whole buffer is written onto the output file in "compacted" format, which reduces a series of zero's to a single nimber.

This stripping is relatively time consuming but the resulting file can be used by FIG 3D for efficient 3-D projection from many different angles.

With perspective however the process becomes more complicated. When the eye is at finite distance and not in the X-Z plane, the projections of the square slices cannot be divided into parrellel strips. STRIP then generates non-parallel strips, in effect transforming the data to an intermediate coordinate system in which the projections of the slices are rectangles again. The hidden line algorithm operates in this coordinate system. Prior to display, FIG 3D performs the reverse transformation. In this case STRIP will have to be run for every change in viewing position.

4.5 FIG 3D: Three-Dimensional Transformation and Hidden Line Suppression

From these data in strip-format FIG 3D now has to create the three-dimensional figure. I will first describe the 3-D and then the hidden line algorithm.

Three facts help to simplify matters:

1 - Without perspective 3-D transformations are linear.
2 - Because of the tracing method all data are ordered in the Z-direction.
3 - Through the stripping process all data within a slice are ordered in the X-direction.

It is therefore sufficient to perform the standard 3-D transformation on the 8 corners of the enclosed box. From these, the projections of the four corners of a sheet (the multiple projections of a slice see fig. 6) can be found by linear interpolation. From these, the projections of bottom and top of every strip can, in turn, be calculated by linear interpolation. With efficient bookkeeping all these interpolations can be reduced to a few additions and one multiplication per point. On computers where multiplication is costly this amounts to substantial savings in computer time compared to the nine

multiplications per point needed for standard transformation.

Without perspective this method creates the exact 3-D transformation. With perspective the transformation isn't linear anymore, so the just described method introduces an error. FIG 3D uses the exact transformation for the corners of the box and linear interpolation for all other calculations. If the viewing distance is not less than four times the size of the object, the error is less than 1%. For most cases this is totally acceptable.

The main justification, however, for the strip-format is the resulting simplicity of the hidden line algorithm. Let us again consider the case without perspective. The case with perspective presents no extra problems because of the transformation already performed in STRIP.

The screen has been divided into strips again and its content is stored in strip-format in the "cover-up array". At first the screen is clear and this array empty. Let the slice of Fig. 9b be the first to be displayed. Every strip of the input slice corresponds to one strip of the cover-up array. Since this first slice is totally visible, the, by the 3-D algorithm transformed, values are copied into the appropriate strips of the cover-up array. Now the non-covered parts of the next slice (Fig. 9c) need to be displayed. FIG 3D compares the projections of every point of a strip in this new slice with the sections defined by successive pairs of entries in the corresponding strip of the cover-up array. There are three possibilities:

1. The point lies within this section: → the point is hidden.
2. The point lies below this section: → the point is visible.
3. The point lies above this section: → get the next pair of entries and check again.

If there are no more pairs within this strip, the point is visible.

If the point is visible its coordinates are stored in a plotting buffer. If the point is hidden while the corresponding point in the last strip was visible or when a right turning point is encountered, the visible polygon is drawn by a plotting subroutine and removed from the buffer.

For plotting on an incremental plotter, a time-saving procedure has been devised. Since the order in which the polygons become available for plotting is rather haphazard, the plotter would spend most of its time moving from one short line segment to the next. Instead of plotting a line segment as soon as it becomes available, an attempt is made to store it in a 2 K buffer. Only when this buffer is full, room is created by plotting the line segment from the buffer that is closest to the present pen position.

At the end of the program, the buffer is emptied the same way.

After the visibility of all the points of a strip has been determined and the visible points entered onto the display file, the strip is merged into the corresponding strip of the cover-up array to obtain the up-dated version: Pairs of points defining covered-up sections are compared resulting in new pairs describing the "visual super position" of the two strips. The number of pairs can decrease as well as increase.

For high precision pictures, the "edge interpolator" is normally in operation,which corrects for the accuracy introduced by the stripping process without comprimising the space and time efficiency of the strip concept. If a line changes from hidden to visible or visa-versa, it must have crossed another line of the image. The edge interpolator tries to identify this line and to calculate the exact point of intersection. Without this option, every visible line segment starts and ends at the midpoint of a strip, as in Figure 9d. For a complex figure as in Figure 10 the visibility test for each point requires on the average only four array references and comparisons and maybe twice that number for the merging process. No addition, multiplication or other arithmetic is needed.

4.6 UTILITIES: MERGE, FIXIT and FILL

Three utility type programs help the user maintain his files. MERGE is used to create a new file out of specified slices of at most two input files. This is useful when a few slices have to be replaced, when a part of a structure has to be cut away to reveal the inside or to reverse the order of slices to take a look at the back side of a structure. FIXIT is used to fix errors made during tracing. It can adjust global values as magnification or slice thickness, join slices or split slices in two. FILL is a simple program to make the structure fill the whole box in FIG 3D for maximum clarity. All these programs are straightforward and will not be described any further.

4.7 EXPERIMENTAL PROGRAMS: CROSS and DRESS

In some cases cross-hatching as in Fig. 5e can markedly improve the clarity of a 3-D picture. For Fig. 5e two sets of data were used (with viewing directions separated by 90°) and displayed on top of each other.

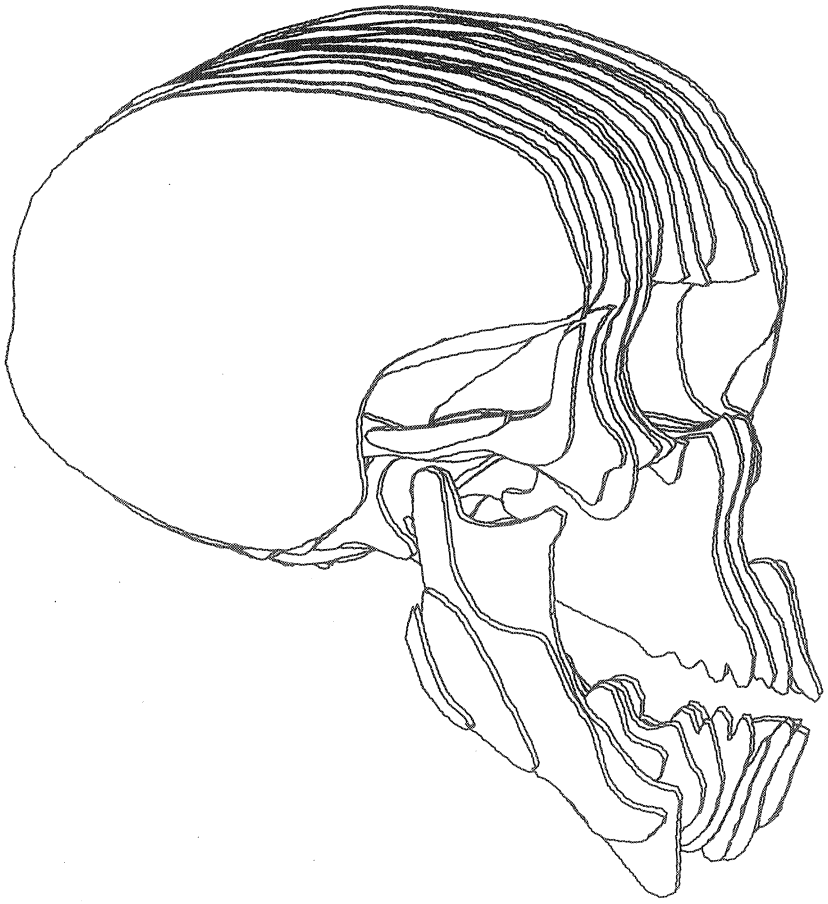


FIGURE 10

Reconstruction of the right half of the skull of the author. For obvious reasons, a non-destructive sectioning technique is chosen based on tomography, an X-ray technique that makes it possible to let only one specific plane of the object show up on the X-ray picture. Here the planes were chosen 5 mm apart so that 30 slices covered the total skull. (courtesy of D. Kerr).

The program CROSS was developed to create the second data set from the first one, both in strip-format. An additional program DRESS (the reverse of STRIP) can be used to create an XY-type file so that all the data handling power of EDIT and the utilities comes available. The combination or the rotating option in EDIT and the cross-sectioning of CROSS linked through STRIP and DRESS makes sectioning under any angle possible.

5. THE SUPPORTING GRAPHICS PACKAGE

The software that supports TROTS is a locally written graphics package that in functions and interface is very similar to the standard Calcomp subroutine package. The relevant extensions are a subroutine to output polygons with savings of 50% in I/O time, and the possibility to write the programs totally device independent: The choice of the output graphics device can be at program or job control level.

The availability of more sophisticated graphics software at the level of GINO or GPGS would not necessarily have facilitated the plotting programming. EDIT and FIG 3D are the only programs producing graphics output. EDIT could have made use of the 2-D transformation facilities and the possibilities of random deletion and addition of picture segments, but this would have simplified the programming only if the transformed data were available to the program (the write-back option) since that is the data to be stored in the output file. FIG 3D would have to forego the efficiency of the strip-format to be able to use 3-D standard software or hardware. But again a write-back option would be essential for the hidden line algorithm. An example of how this combination of 3-D hardware and write-back facility could be made very useful is described in the next section.

6. TOWARDS A REAL-TIME SOFTWARE HIDDEN LINE REMOVER

A very powerful depth cue is provided by the kinetic depth effect which is obtained when a three-dimensional scene is rotated around a suitable axis such that near and distant points will move in opposite directions. If suitable input tools are available (as control dials or joystick), this process can be made interactive giving the user the impression of a real 3-D object which he can turn around and zoom in on producing a very realistic effect. Of course, this requires a display of the refresh type rather than the storage tubes we have been discussing so far.

In applications like these, efficiency becomes very important. To obtain a smoothly changing picture we have to provide the refresh hardware of the display with a new picture at least once every 200 msec. To complete the 3-D transformation and hidden line removal for a whole scene within these 200 msec is hardly possible in software for pictures of any complexity. Several hardware implementations for 3-D transformation are available on the market. We worked with the Picture System of Evans & Sutherland consisting of the 3-D transformation hardware (3-D box), display, 4K refresh memory and data tablet connected to a standard PDP 11/40 mini-computer with 32 K of core. Hardware hidden line removers have also been constructed, but are not to our knowledge commercially available. The efficiency of the TROTS hidden line algorithm combined with a sophisticated interplay between software and the 3-D box makes it possible to produce a good approximation to the ideal case of interactively changing display with all hidden lines removed.

I postulated that human perception has different sensitivities to different inaccuracies. To obtain a flicker-free image on the refresh tube we should display more than 20 fps (frames per second). We set our refresh rate at 40 fps. This only involves the transfer between refresh buffer and display and is independent of the rest of the system (Fig. 11). To obtain a smoothly changing picture it is not necessary that all these frames are different. For film animation, for instance, it is standard procedure to project pairs of identical frames in succession so that there are, at most, 18 different frames per second. I set our frame update rate at 8 fps and still obtained a smooth movement. This update involves adjusting the 3-D box to the new viewing position and starting the transfer of the 3-D data from the main memory through the 3-D box to the refresh buffer. This transfer is handled by a Direct Memory Access module leaving the CPU free to attend to the user interaction and hidden line removal.

Even our fast algorithm is not able to complete the hidden line removal for the total picture 8 times per second, but I reasoned that our perception is not very sensitive to minor inaccuracies in the hidden line removal as long as the major portions of the hidden lines are removed and the major part of the visible lines are displayed. The solution then is as follows: In the main memory we have four arrays. TOTAL contains the xyz-coordinates of the complete scene. The 3-D hardware has the option of writing its 2-D output back into the core memory rather than the refresh buffer. This feature is used for the fast conversion of the 3-D data of a

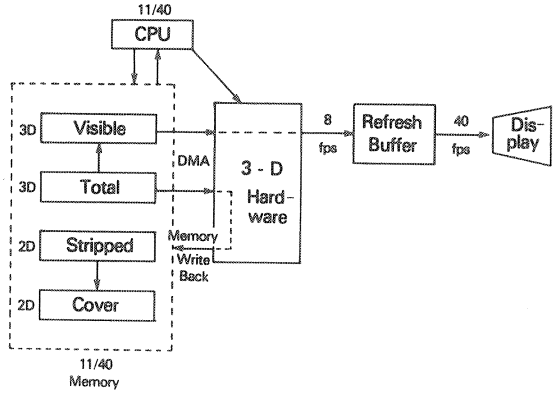


FIGURE 11

Data flow diagram for the real time removal of hidden lines from a dynamic display, using the special hardware of the Picture System.
Array TOTAL contains the xyz-coordinates of all the points. VISIBLE only the xyz-coordinates of the visible points. STRIPPED contains the 2-D projections of one slice and COVER is the cover-up array in strip format.

sheet to a stripped format similar to the one described before. Since, for dynamic purposes, a limited resolution is considered adequate, the image is divided into only 50 strips. This 2-D output of a sheet is stored in STRIPPED. By comparing STRIPPED to the cover-up array COVER, the program decides on the visibility of each point and transfers only the xyz-coordinates of visible points from TOTAL to an array VISIBLE. At the end of every sheet STRIPPED is merged into COVER as described in Section 4.5. It is this array VISIBLE that is used as input to the 3-D hardware for the frame update 8 times per second.

A hidden line update cycle consists of applying this procedure to all the sheets in succession. The viewing position for the hidden line algorithm within one cycle has to be the same for all the sheets. This is called the cycle viewing position which is generally different from the current frame update position which changes constantly during a cycle. The error in the hidden line removal is minimized if the cycle viewing position is chosen such that at the midpoint of the cycle it is equal to the current update position. This can be done if the viewing position changes at a constant rate and if one knows the total length of the cycle. After a few cycles, the program can make a good estimate of the length of the upcoming cycle. Moreover, it can alter this length by changing the number of strips. This opens the possibility of automatic determination of the optimum computing parameters at which the sum of errors is minimal, depending on the speed of the movement.

Note that the only inaccuracies are that certain hidden line segments are displayed and certain visible line segments are removed round the edges of the scene. This is mostly so around the start and the end of the cycle. At the midpoint of a cycle we have a perfect picture. The 3-D perspective projection, however, is at all times exact and not influenced by the stripping process.

Although this system is not yet operational, the software has been written and tested. Many core and time-conserving measures have been incorporated (such as compacted strip format and routines in assembly languages where practical) and it has been demonstrated that with some minor additions to the 3-D box, perfectly acceptable pictures can be generated from fairly complex scenes with this simple mini-computer with the standard amount of core.

7. EVALUATION

One of our objectives was to design a system that could be used directly by the people that are interested in 3-D reconstruction without specialized training or the interference of a separate operator. These people (in our case, biologists) are experts on the object that is to be reconstructed but, more often than not, have no previous experience with computers. Moreover, their main interest is not in gaining computer expertise but in obtaining a reconstruction easier and faster than they could by conventional means.

So there is a need for a "friendly" system, that is self-explanatory, is tolerant of mistakes, guides the user and is not intimidating by forcing a new user to make a great number of decisions whose consequences he is not familiar with. For this reason we have given much attention to the assistance and chaining features and to making the interactions as clear, natural and short as possible.

On the other hand, reconstructing subcellular structures of unfamiliar shape is not a trivial task and the system should give an experienced user some control over the detailed operation of the algorithms. For this reason, the dialogues include some more technical questions that normally are skipped.

The TROTS system has been extensively and successfully used by the research group who provided the original motivation for the project. When the availability of the package became known other groups started making use of it. The different groups at the University of Pennsylvania that were using TROTS in their research include botanists reconstructing a plant's apex from a series of 80 sections, cardiologists studying the shape of the ventricals, and plastic surgeons studying the possibility of predicting the visual result of a proposed operation on a deformed skull. The latter group uses a non-destructive "sectioning" technique based on tomography (see Figure 10) that could conceivably be used for all objects that are x-ray opaque. Some of these groups have added their own sections such as an "operation" procedure for EDIT and elaborate smoothing programs. In all these cases TROTS proved to be as flexible and as easy to use as we had hoped. Generally after a short demonstration users could independently find their way through the system and produce their first reconstruction in a matter of hours. This is often directly followed by a new tracing session, since the first trial usually brings home the point that careful tracing

and choosing adequate fiduciary marks largely determines the quality of reconstruction.

One of the reasons TROTS can be used for a wide variety of applications is that it makes no predetermined assumptions about the shape, topological nature or complexity of the objects to be reconstructed. The other is that the software is flexible enough to gracefully incorporate new developments without affecting the integrity of the system. This is facilitated by the USER option in EDIT, while the file handling through the DSKIO subroutines makes adding new programs extremely easy.

The package is also cheap in its operation. A complete process from tracing to final reconstruction can be done in a few hours using less than one minute CPU time.

The TROTS software is completely written in FORTRAN and it is expected that it can be run on all computers that have sufficient memory and graphics display capabilities of either the refresh or storage type. Magnetic tapes containing source programs and documentation will be supplied to any facility that has the necessary hardware and programming skills to adapt the system to the particular installation. This should present no particular problems since the programs are profusely documented.

REFERENCES:

[1] A. VEEN & L.D. PEACHEY, *TROTS: A computer graphics system for three-dimensional reconstruction from serial sections*, Computer and Graphics, (in press).

[2] W.M. NEWMAN & R.F. SPROULL, *Principles of Interactive Computer Graphics*, McGraw-Hill, N.Y. (1973).

SHAPE PROCESSING FOR MECHANICAL COMPONENTS

I.C. BRAID
Computer Laboratory, Cambridge University

1. A MECHANICAL COMPONENT OBSERVED

Computer-aided design has benefited many areas of engineering but has been slow to make an impact on the mechanical field. Among the many reasons that might be adduced for this lack of progress - the conservative nature of the industry, the variety of manufacturing methods, the fact that many products of mechanical engineering are of relatively low value when compared with the work of say civil and electrical engineers, one stands out: the sheer complexity of shape enjoyed by these ubiquitous components.

In this talk, I shall examine the shapes of engineering components and show how their complexity can be tackled. I shall then touch briefly on previous work in the field, and finally describe some of the present research at Cambridge.

Fig.1 shows a mechanical component of moderate complexity, in fact part of a mounting for a gyroscope. Considered as a point set in three dimensions it can be seen to have an interior, in this case occupied by metal and bounded, an exterior of air unbounded, and in between a boundary, that is, its surface. Although in principle a component might be modelled in a computer by storing a description of its interior or even its exterior, the finite size of computers makes such a scheme impracticable. Instead, a representation of the bounding surface of the component is stored. By giving in addition a sense to the surface, it is then possible to infer whether any point is within the component, outside it, or on its boundary.

There are other, more indirect, ways of modelling a component's shape. Traditionally, engineering drawings formed the medium for storing

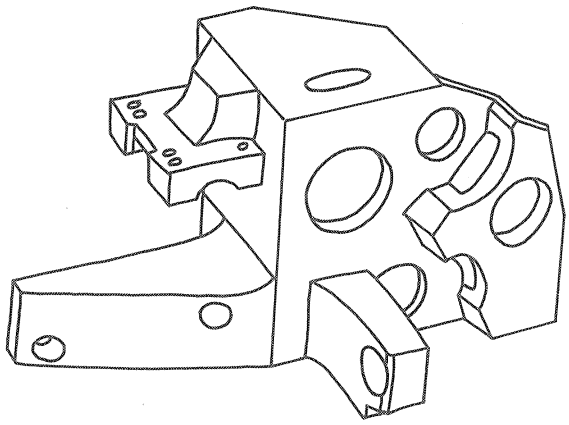


fig. 1

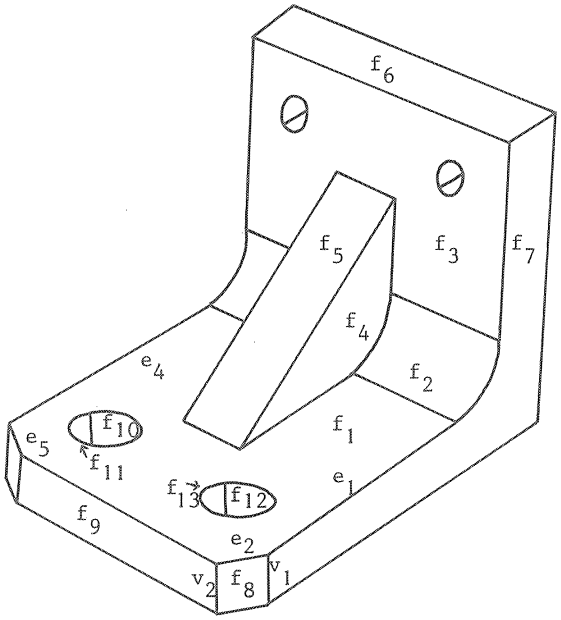


fig. 2

and transmitting shape information. Indeed, a Frenchman, Gaspard Monge (1746-1818) must be given credit for devicing a systematic method based on orthogonal plane parallel projections. Early successes in computer graphics encouraged digital encoding of drawings, in some respects an unfortunate development as it gave the impression of storing shape information yet did so in a form that, except for certain simple classes of shapes, could be interpreted only by human intelligence. It is, in general, impossible to compute the weight of a component from a stored drawing, still less to find and draw a section through the component.

With the advent of numerically-controlled machine tools, a second indirect method of shape representation arose. In this case, it was the three-dimensional motion of a cutter that was stored. Once more human intelligence, (of a part-programmer) was needed in order to translate from a drawing to a coded description of cutter motion and once more, that description failed as a basis from which to deduce, for example, the weight or shape of a component.

It must still be shown, of course, that drawings, NC tapes, weights and so on can all be derived from a stored shape description, and it must be admitted that, despite substantial progress, this task has yet to be completed for a generality of component shapes. Before describing recent work in this area, I would like to return to the question of how to restore the surface shape of a component.

2. MODELLING THE COMPONENT's SURFACE SHAPE

Fig.2 shows a component whose boundary has been split up into pieces. Each piece is termed a *face* and each face lies in a *surface*. For example face f_3 lies in a surface S_3 which is a plane whereas f_2 is in a cylindrical surface S_2. Faces meet at edges and edges lie in *curves*. Edge e_1 is in a straight line C_1, edge e_6 is in a circular are C_6. Edges meet at *vertices* and vertices are at *points*. We term the face-edge-vertex structure the topology of the component's shape, and the surface-curve-point information, its *geometry*. In essence the topology describes how the pieces of the boundary are connected together while the geometry describes the shape and position of the individual pieces. When a component is altered, either the geometry or the topology or both may change. We shall see that

the separation of shape information into topology and geometry not only helps a conceptual understanding of the problem, but has important consequences in the realization of a practical design system.

3. BUILDING UP A SHAPE MODEL

Although in principle a model could be input directly by giving the position of points and equations of curves and surfaces together with their corresponding vertices, edges and curves and details of how they are connected, such an approach is tedious and error prone. Programs for finite element stress analysis that employ a low-level form of input exhibit the difficulty. Since the model is inherently redundent - a curve equation can be deduced from the surfaces of the faces intersecting in the edge that lies in the curve, for example - questions of model consistency soon arise.

One approach is to supply just the surface equations. The TIPS system of Okino uses this form of input, each surface being given by an equation in the form $f(x,y,z)=0$ where the sense of the surface is fixed by saying that the function is positive on the inside of the surface. If the shape is convex, no further information is needed. A convex polyhedron is fully defined by its n face equations but to find and draw its edges, requires $0(n^3)$ operations. To describe a non-convex shape in this manner, the order in which the faces are intersected must be given by supplying a boolean expression. The TIPS system stores an object in the form

$$F = \bigcup_{j=1}^{m} \{ \bigcap_{i=1}^{n} f_{ij} \},$$

that is, as the union of the intersections of the surfaces f_{ij}. It uses a three-dimensional array of points, each being tested against the functions f_{ij} to find points near the object surfaces. A potential function defined in terms of the f_{ij} speeds up the location of points on the actual boundary of the object, given a nearby point in the three-dimensional array. Even so, computation times are long. The system is unable to make use of topological information since this information is never stored explicitly.

A second approach is to input boolean combinations of primitive shapes rather than boolean combinations of surfaces. This has immediate advantages from the user's point of view as he is now describing shapes in terms of bounded entities or volumes. Their effect is limited to a finite

part of design space and moreover, they are of the same type as the object being designed. There is no need for the user to understand details of surfaces or boolean algebra: he can be presented with a design system in which he simply adds and substracts building blocks in order to create a complex shape.

For the implementor of such a system, there is still a choice to be made. He can store the building blocks as collections of directed surfaces (as is done in the PADL system [Voelcker]) and also the boolean expressions implied by the user's sequence of additions and subtractions. Only when a picture of the object is requested need he actually evaluate the boundary to find finite edges for display or faces for producing an NC tape.

Alternatively, he can include topological and geometric information in the stored descritption of the primitive building blocks. By doing so, he can greatly assist the evaluation of addition and subtraction operations, and can arrange to keep an up to date, evaluated description of the shape at each stage of the design. The existence of the evaluated description makes display of wire-frame pictures of an object a trivial matter.

Undoubtedly the greatest difficulty in implementing such a system is to evaluate the intersection of two shapes and to produce one or more shapes as a result. Initially it seemed that algorithms for this purpose always contained a host of special cases. However, over the last two years, when we have been developing a new shape design system, we have given much attention to the general intersection problem and believe that we now have the special cases under control.

4. A NEW SHAPE DESIGN SYSTEM

An underlying premise of the new system is the belief that the topological and geometrical aspects of shape should be kept distinct. The topological treatment is an extension of work at Stamford [Baumgart] where it was observed that polyhedra obeing Euler's rule could be built up from a single primitive using just two operations. Euler's rule states that

$$n_f + n_v = n_e + 2$$

where n_f, n_v, n_e are the numbers of faces, vertices and edges respectively. The simplest Euler polyhedron for which $n_f = n_v = 1$, $n_e = 0$ we take as our

primitive object. The two operations are

OP_1 : add edge and vertex $\quad n_e \leftarrow n_e + 1, \quad n_v \leftarrow n_v + 1$, and

OP_2 : add edge and face $\quad n_e \leftarrow n_e + 1, \quad n_f \leftarrow n_f + 1$.

By starting with the primitive polyhedron and applying the operations an appropriate number of times, any (n_f, n_v, n_e) satisfying eqn. (1) can obtained. In practice we take a more general form of (1) to allow for multiply-connected faces and polyhedra with handles.

Baumgart also proposed a representation for shape topology which allows connected edges, faces and vertices to be found without searching (Fig.3). Every edge is stored together with pointers to the vertices at its ends, to the faces on either side, and to the neighbouring edges (termed the winged edges) going clockwise or counter-clockwise about these faces. Each vertex and face carries a pointer to one of its edges. Using the winged-edge pointers, all the edges at a vertex or round a face can be found directly. As faces can be multiply-connected, we introduce a further data type, a loop and say that a face is bounded by an outer loop of edges and possibly by inner loops denoting holes. Each edge then belongs to two loops which in turn belong each to a face.

A cube and the other ready-stores primitives of the original system are now built up from the single primitive and application of the two operators. Rather than apply the operators directly, it is useful to set up three intermediate operators. These do the opposite of a projection, that is they increase the dimension of an object by one. The first takes a point object and makes it linear, the second sweeps a linear object into a two-dimensional object, and the third takes a 2D object and sweeps it along the third dimension to make a 3D object. Fig.4 shows a cube being made in this way. Edges added using OP_1 are shown as a full line, those added by OP_2 as dashed lines. The final cube has required seven applications of OP_1, five of OP_2.

We have chosen to add geometric information as the sweeping operators are applied. Conventionally the 0D to 1D operator first moves the vertex one unit in the negative x direction and then sweeps forward two units in the positive x direction with a single application of OP_1. The 1D to 2D operator works similarly in the y direction, here making two calls to OP_1 and one to OP_2. The 2D to 3D operator will in fact sweep any 2D lamina in the z direction; in this case it calls OP_1 four times and OP_2 four times.

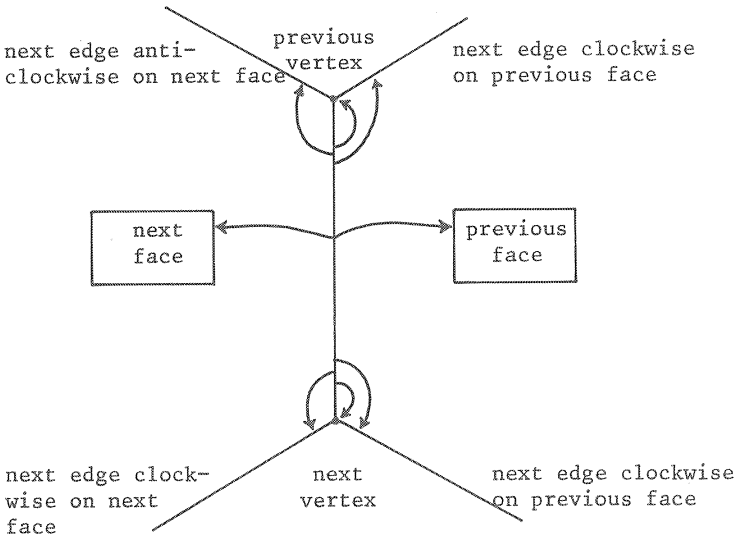
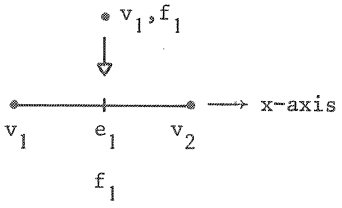
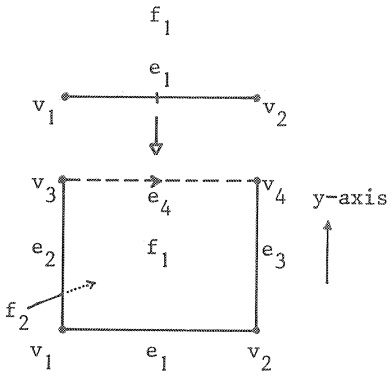

previous
vertex
next edge anti-
clockwise on next face
next edge clockwise
on previous face
next
face
previous
face
next edge clock-
wise on next
face
next
vertex
next edge clockwise
on previous face

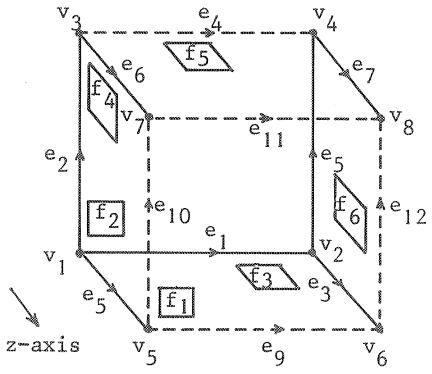
fig. 3



sweep point into line



sweep line into lamina



sweep lamina into solid

fig. 4

Other intermediate operators are provided. One sweeps a linear object around an axis to make a 3D axi-symmetric object.

5. GENERAL INTERSECTIONS

As remarked above, the most powerful operation in a volume-based system is the general intersection. With it, a user can quickly build up complex objects by adding and substracting simple shapes. The intersection routines keep track of all the changes to topology and geometry which occur during an addtion or substraction and are necessarily complex. In the original system, two special cases of a general intersection were handled. One, termed merging or type-1 intersection, combined objects which touched at flat faces but did not interpenetrate. The second allowed an arbitrary object and a cube or cylinder to intersect without restriction.

In the new system, any two objects can intersect. Great care is taken to see that all the cases which can occur are handled correctly. The most common is where two faces intersect in a curve (fig.5), new edges will have to be inserted in the positions of the curve common to the two faces. However, it is possible for the surfaces in which the faces lie to intersect nowhere (two parallel plane faces) or everywhere (two coincident parallel plane faces). The same case of coincidence can occur between any combination of faces, edges and vertices. The intersection algorithm makes no assumptions about convexity fo faces. Although it is being tested on polyhedra, we have been careful not to build in any assumptions which rely on the objects being polyhedra. If an object is cut into pieces by intersection with a negative object, the resulting pieces are identified and returned as separate objects.

As in the original system, intersections occur in two stages. In the first, intersecting faces, edges and vertices are discovered and marked deleted or not. New edges are inserted and linked in to the existing objects. In the second stage, the objects are scanned, edges are collected into loops, loops into faces, faces into objects. Interestingly, the primitive operators of adding an edge plus a vertex or face, are not very helpful in intersections. The problem is that during intersections the objects exist in a form which does not satisfy Euler's rule, that is, they are not Euler polyhedra. It would be pleasing to find a way of performing general intersections efficiently using only Euler operators to handle the changes in topology.

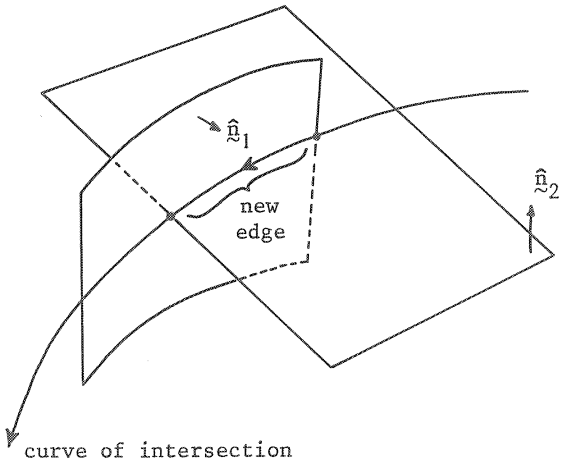


fig. 5

6. GEOMETRIC EXTENSIBILITY

Apart from clarity of expression in programs and algorithms, the greatest gain from separating topology and geometry in a shape design system occurs when we want to extend the range of curves and surfaces handled by the system. For example, in intersections we frequently want the curve of intersection of two faces. This is implemented as a single procedure and in turn calls other procedures to handle the different cases: plane with plane, plane with cylinder, cylinder with cylinder and so on. If a new surface type is to be added, new procedures must be written to handle its intersection with each existing surface and with itself. The geometric growth in number of procedures is unavoidable and is an incentive to limit the number of different surface types.

For any set of curves and surfaces, we must be able to supply the following routines:

a) (surface, surface)[ ]curve where the curve is delivered, possibly in pieces, each piece in parametric form.

b) (curve, curve)[ ]point where the points are those in which the curve cuts the surface. Coincident curves must be recognised.

c) (point, surface)vector where the vector gives a directed normal to the surface ending at the point.

d) (curve, surface)[ ]point where the point(s) if any are those in which the curve cuts the surface. The routine must also recognise the case when the curve lies on the surface.

e) (point, curve)real where if the point lies on the curve, parameter value is returned as a real number.

f) (point, curve)vector where the vector is the tangent to the curve at the point on it.

g) (point, curve, surface)kleen where the kleen, a 3-valued boolean, signifies whether the point is to the left, on or to the right of the curve, both point and curve lying in the surface.

h) (surf, trans)surf where the trans is a linear transformation.

i) (curve, trans)curve.

j) (point, trans)point.

Notice that curves must be provided in parametric form as one step in intersections has to sort points along a curve and does so on the basis of parameter value. There is no requirement for surfaces to be parametrised though a parametric surface is convenient when deriving cutter paths.

7. COMMUNICATING WITH THE SYSTEM

As the system is written in ALGOL 68C, we have been able to set up operators so that objects can be described as expressions in the language. This arrangement gives great power to the user and is especially suited to the development of special-purpose front-ends to the system. For example, it would be possible to write a front-end for design of a class of components such as pistons. The engineering designer would be able to communicate to it in his customary terms. The front-end could include design rules or any other information or practice specific to piston design. In this way, the special knowledge about a class of shapes, which does so much to reduce the amount of input and chances of error, can be separated from the underlying, general-purpose shape processor.

Another possible front-end is of course a command interpreter. We have written one in order to use the system interactively.

8. CONCLUSION

We are now entering a period of consolidation in shape processing. More attention is being given to system design. We find it beneficial to separate a general-purpose core or kernel of shape processing routines from user-oriented front or back ends. Within the general-purpose routines, geometry and topology are best treated separately also. In this way, a system can be made extensible in its geometry, and we are able to overcome the unavoidable complexities of topology once and for all.

REFERENCES

Baumgart, B.G., 1974, Geometric modelling for computer vision, Stanford Artificial Intelligence Laboratory report, STAN-CS-74-643.

Braid, I.C., 1973, Designing with volumes, Ph.D. Thesis, Cambridge University.

Okino, N. et al, 1973, TIPS-1; technical information processing systems for computer-aided design, drawing and manufacture, proceedings of Prolomat '73, Budapest.

Voelcker, H.B. et al, 1974 An introduction of PADL, TM-22, Production Automation Project, University of Rochester.

ALGOL 68 G GRAPHIC EXTENSION OF ALGOL 68

P.J.W. TEN HAGEN

1. Introduction

One of the main goals of the MC project on computer graphics (see [1]) is the design of a high level graphic language. Two major criteria determine this design:

- All non-graphical elements in the language are borrowed from an existing general purpose high level language.
- All basic graphical constructs in the language are derived from a newly designed special purpose low level graphic language called ILP (see [2],[3] and [4]).

Both criteria mean to avoid work that has already been done. For the first criterion this is obvious: (re)inventions take a lot of time. However, if one borrows from an existing language, one also has to face its poorly designed parts and difficult features. For this reason part of the saving is lost again by trying to circumvent or hide from the user all facilities which are too complicated or too clumsy.

The second criterion especially avoids duplication of our own effort, a lot of which has been invested in the design of ILP. This could only be justified by the fact that ILP can be applied in a multitude of ways. For the high level graphic language it means:

- The data structure for the language is defined.
- The I/O routines for graphical data only require a trivial conversion.

- A skeleton for interaction with the high level language is obtained.

The above is nothing else but stating that the high level graphic language is supported by a graphics system which is capable of a direct interpretation of the graphical data structure.

Both criteria are not sufficient however to guide all major aspects of the design. They only provide the basic layer of the language.

2. Layered structure.

The graphic language has a layered structure. On the bottom layer a data structure exists which is obtained by embedding ILP in the host language. This data structure together with the operations on it provided by the host language (assign, compare, select, input, output), already constitute a high level graphic language, say GL0. The next layer, say GL1, consists of GL0 extended with three sets of operations and the new data structures created by them.

The first set contains operators and procedures for extracting GL0 data. A complete picture description in GL0 may contain a lot of irrelevant information with respect to a given set of manipulations. Extraction is used to obtain relevant information in one of the following three ways:

- The structure of the picture can be simplified. This is called compression.
- Certain aspects (say, line styles) are ignored. Such operations are called reduction.
- The picture is searched for certain properties, we call this selection.

Any combination of these three operations will also be referred to as extraction. Extraction always produces GL0 data.

The second set of operations is called manipulation of GL0 data. This term is reserved for operations which can be defined according to the scheme:

$$g1 * g2 \Rightarrow g3,$$

where g1, g2 and g3 are GL0 data. g3 may be a newly created value, more complex than g1 and g2.

The difference between extraction and manipulation is that extraction never produces a structure more complicated than its operands. In principle extraction has priority over manipulation. The combination of extraction and manipulation is obtained by using the feature of (if any) of the host language to write expressions.

The third group of operations associates GL0 data with non-GL0 data. The result of these operations is a data structure outside GL0. Schematically:

g <> h => x.

Either g or h must be in GL0. Hence these operations carry us at most one step outside GL0.

The extension of GL0 with extractions E0, manipulations M0 and associations A0 is called GL1. In a similar way GL2 may be obtained from GL1 by adding a new layer of operations and data structures of the three types:

GL2 = GL1 + E1 + M1 + A1.

The associations can be divided according to two criteria namely whether the new information adds structure or data, and whether the new information has a graphical interpretation or not. The four associations are called graphical threading, graphical coupling, external threading and external coupling respectively.

3. The Host Language.

Under the assumption that all basic graphics facilities are included in ILP, the main embedding principle is that whenever the host language offers a certain facility that resembles an ILP concept, it will be used for that purpose. For instance, if the language has an array construct it is used for rows of coordinates as well as for lists of pictures.

This principle may lead to a more efficient implementation on the one hand, on the other hand it presents the graphics facilities in a way already familiar to the user of the host language.

We have decided to take ALGOL68 as the first host language. The richness of facilities present in ALGOL68 allows us to embed ILP almost directly in the form of mode declarations in a so-called particular prelude. This means that no extra compiler, preprocessor or library is required to test ALGOL68G programs. The ALGOL68G language will be used as a blueprint for combinations with other languages.

We will however strongly encourage efforts independent of the blueprint to embed ILP in other languages.

In the remainder of this paper we will put emphasis on the graphical aspects of the language rather than on the ALGOL68 aspects. The way in which ALGOL68 facilities are applied (or left unused) will not be justified. Moreover, as ALGOL68G is still under development, we present a snapshot of the first implementation. Before the definition of the language will finally be published, most of the language as presented will have been redesigned as a consequence of intensive exercising.

The state of ALGOL68G is that GL0 is being tested.GL1, GL2 etc. are not yet designed or implemented. The remainder of this paper is devoted to GL0. Since it is a nuisance to rewrite in this paper all details about ILP, the reader is advised to read the lecture notes of this colloquium on ILP first.

4. The Basic Layer.

The embedding of ILP is straightforward in the sense that each syntactic terminal or non-terminal of ILP can be found in ALGOL68G, more specifically in GL0, as a 'mode'. For example the syntax rule from ILP

```
<picture element>: <coordinate type> |
                   <text> |
                   <generator> ;
```

leads to the 'mode' declaration:

```
'mode' 'pictel' = 'union' ('plc'#coordinate type#,
                           'text', 'generator'
                          );
```

Not all 'mode's in the basic layer are derived from the ILP syntax. Extra 'mode' declarations are introduced mainly for two purposes:

- Part of the semantic requirements like consistent dimensions can be enforced by introducing explicit 'mode' differences for different dimensions. In this way the 'mode' checking guarantees correct dimensions.

- Each picture can be represented in an infinite number of ways. Efficiency requires a canonical representation at all places where a more specific one is not required. By introducing special 'mode's for these representations (which are submodes of other modes) they can in many cases be determined statically. For these pictures all operations take advantage of the

structure known beforehand.

In principle 'mode's of this kind are hidden from the user. The user declares the supermode, but the values he produces are stored as so-called secret modes. In general secret modes have the advantage that they can be changed by the designers/ implementers without consequences for the user interface.

4.1 Geometry.

The most essential part of graphical data characterises them as geometrical objects. This includes things like coordinates, dimension, points, lines, faces and transformations.

The basic object for specifying a position in geometrical space is the dimensional value ('dv'):

```
#'mode' 'dv1' = 'real'#
  'mode' 'dv2' = 'struct'('real'x,y);
  'mode' 'dv3' = 'struct'('real'x,y,z);
```

A dimensional value appears in picture elements, transformations and subspace selections. As picture elements contain rows of successive positions rather than isolated points (the four corners of a square), there is a mode for lists of positions:

```
'mode' 'ldv1' = 'flex'[1:0]'real';
'mode' 'ldv2' = 'flex'[1:0]'dv2';
'mode' 'ldv3' = 'flex'[1:0]'dv3';

'mode' '?coord' = 'union'(
                  'ref''ldv1', 'ref''ldv2', 'ref''ldv3');
```

The 'mode' '?coord' as secret 'mode' forces the user to invoke system provided operations for manipulating coordinates. This makes dimension checking full proof.

A list of dimensional values ('ldv') is the basic object for geometric manipulations. It can represent a sequence of one, two or three dimensional absolute or incremental vectors. It can specify sequences of points, line pieces, contours etc. All dimensional values must refer to positions in the unit (hyper) cube. If they don't, apropriate transformations and positioning must be attached to them. All this additional information becomes specified when an 'ldv' is given as part of a picture.

```
Example:
 'ldv2' flag := ((0,2),(1,0),(0,-1),(-1,0));
 'ldv2' cross := ((0.5,0.5),(0,0.33),(0,-0.67));
 'ldv2' standard := ((0,2),(1,2),(1,1),(0,1),(0.5,1.17),
                     (0.5,1.5),(0.5,1.83));
```

In this example, flag followed by cross can be made to specify the same positions as standard. To this end flag and cross must be interpreted as incremental positions, standard as absolute positions.

We will now introduce some picture elements that contain dimensional values.

```
'mode' 'plc' = 'union' ('point','line','contour');

'mode' 'point' = 'struct'('?coord' p);
'mode' 'line'  = 'struct'('?coord' l);
'mode' 'contour'='struct'('?coord' c);
```

The values for 'plc' must be specified by procedures in which the right dimensions must be specified.

```
Example:
        'point' pl := point2(cross);
        'line' ll := line2(flag);
```

Now pl specifies three "points" to be drawn at the positions of cross. Similarly ll specifies 4 "line pieces" to be drawn between the positions of flag. Note that since flag is of type 'line' the pen position is added at the beginning.

To ensure that pl is drawn immediately following ll we can put them together in a picture list,e.g.:

```
        [1:2]'picture' flam := (ll,pl);
```

or, as expression:

```
        [1:2]'picture' flam :=
                (line2(flag), point2(cross));
```

In order to produce the same picture starting with standard, we might proceed as follows: We must devide standard in a line part and a point part.

```
´point´ p2 := point2(standard[5:8]);
´line´ l2 := line2(standard[1:4]);
```

Next, we must indicate that the dimensional values are absolute positions. To this end we change the coordinate mode and put the result in the list.

```
[1:2]´picture´ stam :=
        (mx([1:1]´xplc´:=(´not´CM,l2)),
         mx([1:1]´xplc´:=(´not´CM,p2))
        );
```

or, using the special operator to change to absolute coordinates:

```
[1:2] ´picture´ stam := (´abco´l2,´abco´p2);
```

Adding an <attribute match> to a picture element changes its ´mode´ from ´plc´ to ´mplc´. In the latter the matches are stored, whereas in the former they have default values (all ´true´).

The procedure mm adds a two level match system to a ´plc´ sequence. mx only adds exceptions. The special cases for which operators exist act either on one particular match or on one particular position.

The pictures flam and stam produce exactly the same effect if the first positions of flam and stam coincide, e.g.:

```
PP + flag[1] = standard[1].
```

This relation is independent of transformations, same positions incremental or absolute remain the same under the same transformations. Hence, if one wants to convert between absolute and incremental mode one needs to know the penposition. This is acceptable since one must know the penposition anyway: Either to know what the first line segment looks like(absolute) or where the line will be positioned (incremental). The alternative requires that the first position will be made explicit. In that case one cannot easily connect drawings when the preceding operation takes place under different transformations.

To enforce that a line starts in a given absolute position on may use the operator

```
´op´ ´fiv´ = (´dv´ pos, ´plc´ plc) ´plc´:
```

This operator adds an invisible absolute "move" at the beginning.

4.2 Transformations and Subspace.

Like all attributes transformations must be specified with correct dimensions. Although all ILP transformations are present in GL0, we will, for the sake of brevity only mention rotation, translation and scaling.

Each primitive transformation has a 2d and a 3d version. An arbitrary transformation can be specified with the help of so-called "star" expressions for attributes.

```
'trafo' tf := rot2(angle) * tr2(dv2) *
                    sc2(dv2) * tr2 (ddvv2);
```

This expression delivers a value of the 'mode' 'ref 'trafo'.

To apply a transformation to a picture we use the 'wd' operator.

```
'picture' tp := tf 'wd' flam;
```

This is a special case of a more general construct for attributes we will see later on.

Transformations contain all simple (and efficient) means for modeling pictures. The subspace mechanism is a more powerful, and therefore more expensive modeling tool. With a subspace selection one can:

- alter(decrease) the dimension;
- transfer dynamically to a position;
- specify a new non-orthogonal coordinate system.

In general a subspace transformation can be calculated statically but for a translation to the pen position. A subspace also may reset attributes and limit the scope of matches.

4.3 Attributes.

The embedding of attributes has resulted in two ways of representation. One is equivalent with the directed attribute graph representation of ILP. The other consists of so-called state descriptions. It is an example of a more compact canonical representation. Both data structures use the same way of representing primitive attributes, the so-called attribute classes.

```
'mode''attrclass' = 'union'(
                        'ref''trfcl', 'ref''detcl', 'ref''poscl',
                        'ref''stylecl', 'ref''pencl',
                        'ref''cntrlcl'
                        );
```

Attribute classes can either be put into a 'state' as state component or into more complicated attributes, e.g. 'attrpack' or 'attrlist'. Each 'attrclass' can contain all values that might be produced by mixing attributes from the same class.

```
'mode''trfcl' = 'struct'('ref''tmat' mat, #matrix#
                         'ref''window' wdw,
                         'ref''trfcl' parent
                        #backwards linked list#);
'mode''poscl' = 'struct'('ref''bool' byto);
'mode''detcl' = []'ref''dtset';
'mode''stylecl' = 'struct'('ref''linest' lst,
                           'ref''pointst' pst,
                           'ref''typogr' tg
                          );
'mode''pencl' = 'struct'('ref''real' intens,
                         'ref''colour'colour,
                         'ref''blink' blink
                        );
```

The mixing is expressed with the operator * . The expression

```
a * b
```

is defined if a and b are in the same class. a and b may both be 'attrclass' values or primitive attribute values. The result of a * b is again of the 'mode' 'attrclass'. Hence, * - expressions may be of arbitrary length.

The sampling of class values into a 'state' is denoted

with the + - operator. The + - operator is defined for any two attributes. Its effect is that it turns both operands into a 'state' and next combines both 'state's into a new one.

A 'state is a list of 6 state components, one for each attribute class.

```
'mode''state' = []'stcomp';
```

Each 'attrclass' has a fixed index in the row. It is the most compact representation for an arbitrary combination of attributes.

only way to obtain 'state' values is by writing + - expressions. This guarantees that the bounds or the indices obtain unique values.

The alternative representation for attributes constitute of the

```
'mode''attrlist' = 'struct'('int' dim, 'ref'[]'attr' al);
'mode''attrpack' = 'struct'('int' dim,
                            'string' aname, 'attr' attr);
```

An 'attrpack' itself is not an 'attr', but a 'ref''attr' is. The corresponding reference in ILP uses the 'string' aname. This is a typical example of using the 'ref' mechanism of the language for a similar concept in ILP, although a naming convention is defined there. The aname of the 'attrpack' however, still plays an important role It allows us to differentiate between an ordinary 'ref''attr' (as will occur in variables) and 'ref''attrpack'. When the latter is output, for instance, the corresponding pack will be output precisely once. The value of the variable will be output for each reference to it.

4.4 Pictures.

The 'mode's for complex pictures are:

```
'mode''picture' = 'union'('pictel', #this one is not complex#
                          'ref''npict', 'ref''wdnode',
                          'ref''plist', 'ref''subspp');

'mode''wdnode' = 'struct'('int'dim, 'attr' attr,
                          'picture' pict);
'mode''plist' = 'struct'('int' dim, 'ref'[]'picture' pict);
'mode''npict' = 'struct'('int' dim, 'string' pname,
                          'picture' pict);
'mode''subspp' = 'struct'('int' dim, 'ref''subsp' subsp,
                          'picture' pict);
```

We will only discuss 'npict' and 'wdnode'. An 'npict' is produced by attaching a string to a 'picture', by means of:

```
'proc' npict = ('string' pname, 'picture' pict)'ref''npict':
```

On output, like for 'attrpack's an 'npict' also produces only one copy.

A 'wdnode' is produced by attaching an 'attr' to a 'picture', by means of the operator 'wd' .

```
'op' 'wd' = ('attr' attr, 'picture' pict)'ref''wdnode':
```

We can give three alternative ways to attach a sequence of attributes, say A,B,C and D, to a 'picture', say pict.

```
'wdnode' wdp1 := A + B + C + D 'wd' pict;
'wdnode' wdp2 := attrlist((A,B,C,D)) 'wd' pict;
'wdnode' wdp2 := A 'wd' (B 'wd' (C 'wd' (D 'wd' pict)));
```

The first one combines pict with a 'state', the second one with a list of attributes. The third one creates a hierarchical data structure, adding an attribute on each level. They all specify the same picture.

4.5 Example.

To conclude this chapter we will write a program that produces a data structure representing the well known pythagoras tree. It illustrates the expressive power already present in GL0.

```
´proc´ pyth = (´int´ order)´ref´´npict´:
´bgn´ ´line´ l1 := line2(((0,1),(1,0),(-0.5,0.5),(-0.5,-0.5)));
      ´line´ l2 := line2((´dv2´(0,-1));
      ´trafo´ sc := scale2((0.5 * sqrt(2),0.5 * sqrt(2));
      ´trafo´ tleft := sc * rot2(-0.25 * pi),
              tright := sc * rot2(0.5 * pi);
      ´heap´´npict´ :=
      ´if´ order < 1 ´then´
      npict("pyth0",plist((l1,line2(´dv2´(1,0)),l2)))
      ´else´
      npict("pyth"+whole(order,2),
              plist((l1,
              tleft ´wd´ pyth(order - 1),
              tright ´wd´ pyth(order - 1),
              l2))
              )
            )
      ´fi´
      ´end´;
```

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.6b B. DORHOUT, J. KRIENS & J.TH. VAN LIESHOUT, *Leergang Besliskunde, deel 6b: Wiskundige programmering 2*, 1977. ISBN 90 6196 150 5.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium Stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium Stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium Discrete wiskunde*, 1968.
ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969.
ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970.
ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970.
ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971.
ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 X.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975.
ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra , deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.Th. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.

MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOFF, *Asymptotische methoden in de toetsingstheorie; Toepassingen van naburigheid*, 1976. ISBN 90 6196 104 1.

MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.

MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.

MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.

MCS 25 *Colloquium Structuur van programmeertalen*, 1976. ISBN 90 6196 116 5.

MCS 26.1 N.M. TEMME (ed.), *Nonlinear analysis, volume 1*, 1976. ISBN 90 6196 117 3.

MCS 26.2 N.M. TEMME (ed.), *Nonlinear analysis, volume 2*, 1976. ISBN 90 6196 121 1.

MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discretiseringsmethoden*, 1976. ISBN 90 6196 124 6.

MCS 28 O. DIEKMANN, N.M. TEMME (EDS), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.

MCS 29.1 J.C.P. BUS (red.), *Colloquium Numerieke programmatuur, deel 1A, deel 1B*, 1976. ISBN 90 6196 128 9.

MCS 29.2 H.J.J. TE RIELE (red.), *Colloquium Numerieke programmatuur, deel 2*, 1976. ISBN 144 0.

* MCS 30 P. GROENEBOOM, R. HELMERS, J. OOSTERHOFF & R. POTHARST, *Efficiency begrippen in de statistiek*, 1977. ISBN 90 6196 149 1.

MCS 31 J.H. VAN LINT (red.), *Inleiding in de coderingstheorie*, 1976. ISBN 90 6196 136 X.

MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

MCS 33 P.J. VAN DER HOUWEN, *Differentieschema's voor de berekening van waterstanden in zeeën en rivieren*, ISBN 90 6196 138 6.

MCS 34 J. HEMELRIJK, *Oriënterende cursus mathematische statistiek*, ISBN 90 6196 139 4.

MCS 35 P.J.W. TEN HAGEN (red.), *Colloquium Computer Graphics*, 1977. ISBN 90 6196 142 4.

MCS 36 J.M. AARTS, J. DE VRIES, *Colloquium Topologische Dynamische Systemen*, 1977. ISBN 90 6196 143 2.

De met een * gemerkte uitgaven moeten nog verschijnen.