UNCLASSIFIED

Courant Mathematics and Computing Laboratory

New York University

Mathematics and Computing COO-3077-148

CONFIGURABLE SOFTWARE FOR SATELLITE GRAPHICS

Peter D. Hartzman

December 1977

U. S. Department of Energy

Contract EY-76-C-02-3077

UNCLASSIFIED

DISCLAIMER

THIS PAGE
WAS INTENTIONALLY
LEFT BLANK

Table of Contents

THIS PAGE
WAS INTENTIONALLY
LEFT BLANK

Acknowledgements

I wish to express my deep appreciation for the guidance, advice, and review of my work provided by my advisors Professor Jacob T. Schwartz and Professor Thomas Stuart. I especially want to thank Professor Stuart for the software support he provided ranging from the development of the Honeywell 316 version of the LITTLE compiler to the intercomputer communications package which was so necessary for this work.

I want to thank David Shields and Peter MacLean for their cooperation in solving little,random problems that occurred.

THIS PAGE
WAS INTENTIONALLY
LEFT BLANK

ABSTRACT

An important goal in interactive computer graphics is to provide users with both quick system responses for basic graphics functions and enough computing power for complex calculations. One solution is to have a distributed graphics system in which a minicomputer and a powerful large computer share the work.

The most versatile type of distributed system is an intelligent satellite system in which the minicomputer is programmable by the application user and can do most of the work while the large remote machine is used for difficult computations. The intelligence of the system is a function of the hardware and software used.

At New York University, the hardware was configured from available equipment. The level of system intelligence resulted almost completely from software development. Unlike previous work with intelligent satellites, the resulting system had system control centered in the satellite. It also had the ability to reconfigure software during real-time operation.

The design of the system was done at a very high level using the set theoretic language SETL for specifications. The specification clearly illustrated processor boundaries and

interfaces which helped to simplify intercomputer communications.

The high level specification also produced a compact, machine independent virtual graphics data structure for picture representation. A possible extension of the structure for hidden line removal is presented.

The software was written in LITTLE, a systems implementation language. This meant that only one set of programs was needed for both machines. A user can program both machines in a single language. It also meant that most of the software is transportable to machines with available LITTLE compilers.

Tests of the system with an application program indicate that it has very high potential. The major problem with the current implementation is the unsatisfactory performance of the intercomputer communications software. Replacement of that package would produce significant improvements and would not affect system design.

A major result of this work is the demonstration that a gigantic investment in new hardware is not necessary for computing facilities interested in graphics.

Introduction

The purpose of this dissertation is to discuss the development of and philosophy behind the design and implementation of an intelligent satellite graphics system.

A significant problem in interactive computer graphics is to find a means to provide users with quick system responses for basic graphics functions such as picture transformations and input/output and also to provide them with the computational power to do more elaborate applications-oriented calculations. An expensive and impractical method is to connect the graphics terminal to a large, powerful computer. The problem with this approach is that the input/output requirements of the graphics terminal and the servicing of trivial requests do not necessitate the power of a large machine and the complex computations occur relatively infrequently. The result is an expensive waste of the computer's power. In contrast, a minicomputer would be perfectly sufficient to handle the trivial processing but its abilities would be severely taxed by the difficult calculations. The solution is to have a distributed graphics systems in which a minicomputer and a powerful large computer share the work.

In a distributed graphics system, the minicomputer is known as the satellite computer and it handles the immediate input/output processing. The new problem is the division of

the remaining software labor between the satellite and the remote large machine. Three approaches have been made to the problem.

The simplest method is to use the minicomputer as a fixed function satellite. The user's application program resides in the remote computer and treats the satellite as a fancy input/output device. The satellite handles only simple interactions and formats data for transmission to the remote machine. The large computer still does much of the trivial graphics processing.

The second method is to use the minicomputer as a variable function satellite. The user's application program in the remote computer can turn particular graphics functions in the satellite off and on through subroutine calls. The remote computer still has to do a significant amount of graphics processing.

In the last type of system, the user has a programmable satellite. The satellite handles most if not all of the graphics functions and interactions while the remote machine does major calculations. The user also has the ability to program the satellite to do some processing for him. At this point we have an intelligent satellite system. The level of intelligence is a function of both the hardware and the software of the satellite.

Except for the graphics display hardware, the satellite system built at New York University was configured with hard-

ware already available in the computing center of the Courant Institute of Mathematical Sciences. Except for the remote computer most of the equipment left a great deal to be desired in terms of inherent capabilities and functional reliability. We were still able to produce a functioning distributed graphics system with system control centered in the satellite and the ability to do real-time system software reconfiguration. The idea of using the satellite as a master computer helped distinguish this system from previous efforts in which control was exercised from the remote machine.

In Chapter 1 we will discuss the tools and philosophy of our system design.

Chapter 2 will present the various types of satellite graphics system with several examples of each.

In Chapters 3, 4 and 5 we will show how a high level design makes evident processor boundaries and interfaces and thus helps to avoid complex intercomputer communications in a satellite system. We will describe the structure of the system developed at New York University and the functioning of the application implemented on the system.

Chapters 6 and 7 discuss the choice of a virtual graphics data structure for picture representation and how pictures are developed and the data structure is manipulated correspondingly.

In Chapter 8 we will speak of the advantages of the virtual structure including its machine independence (important in a distributed system), and its compactness. A possible extension

of the structure for hidden line calculations will be presented. Also a survey of other data structures will be given for comparison purposes

Finally, Chapter 9 will present results, describe problems encountered, propose possible future improvements, and summarize the work.

Acknowledgements. This work was supported by PHS Grant Number NS-10072, by ERDA Contract No. EY-76-C-02-3077*000, and NSF Grant No. DCR75-09218.

Chapter 1

Methods of System Design

Experienced people agree that in specifying a large software system, the use of structured programming can produce a clear modular design. Here, we will present a top-down design procedure that results in a succinct, well-documented specification making implementation of the desired system easier. This procedure necessitates the use of a language which is more powerful and of higher level than most of the languages used for structured programming. We will use the language to specify components of the intelligent satellite graphics system. The resulting high level modular specification presents a system overview of blocks of highly related modules. The overview will provide a basis for minimizing the intercomputer communications of an implementation.

1.1 Language Hierarchies

One means of encouraging (coercing) designers to use a top-down development procedure is to provide them with a hierarchy of languages, the highest of which is the language of design (Bauer, 1972; Boukens and Deckers, 1974; Dennis, 1973; Geiselbrechtinger et al., 1974; Ichbiah et al., 1974). In this hierarchy of language levels, the primitives of one

language level are defined in terms of the primitives of the next lower level. Some of the primitives of the higher level may be identical to ones on the lower level of definition. There are three methods of providing a higher level language for the designer (Dennis, 1973). One way is to start with an extensible language and to define new operations and data types so that internally the representation of the higher level and the definition level are the same. A second approach is to write a compiler to translate the new high level language into a lower level language. The third method is to write an interpreter for the higher level so that now the higher level is immediately executable. We are not interested in the particular method of implementation of the language but in the use of the language.

Ideally, a design specified in the highest level language results in an executing implementation. An additional benefit is that if the specification language is at a high enough level then the program will be machine independent and can execute on any other machine that has a language hierarchy supporting the same specification language.

One of the disadvantages of this technique is the temptation for the language designers to allow the use of low level constructs within a specific implementation environment (Geiselbrechtinger et al., 1974; Ichbiah et al., 1974). A language user who takes advantage of these low

level features obscures the general abstract solution to his problem and damages its portability to another operating environment.

1.2 Algorithmic Design Languages

An alternative to using a language which permits implementation specific details to become embedded in the design specification is to use a non-hierarchical algorithmic language with such powerful primitives and control dictions that the programmer will focus his efforts on design. For an algorithmic language effectively to fulfill its purpose, it must have features that promote well structured design and machine independence.

"To achieve true machine independence in a language, the concept of using the language must be kept quite separate from that of executing an algorithm written in the environment of a particular computer system." This may mean "that the language is not a programming language at all in the sense that algorithms written in it can be directly compiled for execution on a computer" (Wells, 1974). The language will be invaluable in providing a clear and succinct means of expressing algorithms and in providing good documentation for use throughout the implementation.

Two very high level languages created for the purpose of algorithm design and specification are Madcap 6 and SETL. Both languages have very powerful operators and use concepts

of computational set theory to permit the building of very elaborate data structures (Morris, 1973; Schwartz, 1973).

1.3 SETL

SETL has atoms and sets as its two basic data types. Atoms are integers, reals, boolean elements, bit strings, character strings, labels, blank atoms, subroutines, functions, and tuples. Sets consist of unordered collections of atoms and other sets. To make things even easier for the designer SETL has no type declarations because a global analysis by the compiler can determine which types of values a variable takes on. For example, it can determine that the value of a variable is a "set of tuples, each having an integer first component, a character string second component, and a third component which is a set of integers" (Schwartz, 1975). So a programmer does not need to worry about specific data characteristics while he is in the design process.

The definition of function has been expanded to include both computational functions and set theoretic relations. So a function can be a subprogram or a set of tuples. As a set of tuples, a function can be multivalued. For example, writing "descendants{a}" will access all those tuples in the relation "descendants" which have a first component of "a". Besides acting as a retrieval operator, "descendants"

can also act as a storage operator. For instance

```
descendants(a) = x;
```

adds the tuple <a,x> to the relation "descendants" and removes any other tuples whose first component is "a" from "descendants" thus making "x" the only "descendant" of "a". On the other hand,

```
descendants = descendants + {<a,x>};,
```

where "+" indicates set union, also adds the tuple <a,x> to "descendants" but does not affect any of the other tuples in the relation. These set mapping mechanisms provide powerful tools for structuring data in a manner which is both painless to the programmer and intelligible at the same time because of the elegant, concise syntax derived from elementary set theory.

Besides the fact that all functions and subroutines are recursive which is especially useful for operations on a tree-like or hierarchical data structure, the programmer can also define new infix and prefix functions and subroutines if he wishes. For example, he may define the function

```
definef a factorof b;
     return ((b//a) eq 0); /* b//a gives the remainder
                                of b/a */
end factorof;
```

and invoke it as x factorof y with the result being t (true) if x|y and f (false) otherwise.

One feature that used to be part of the SETL specification language and was useful in this work was the flow block. The flow block provided a pictorial description of the flow of program control through a decision tree and simplified considerably the problem of tracing through several nested if-then-else clauses. The general structure of a flow block is

```
flow          question?
    yesbranch          nobranch
```

The question represents a boolean expression. The two branches can represent either more boolean expressions or labels of statement blocks to be executed if program control takes that branch. Here is an example of a simple flow block that might represent a decision tree for light pen input processing.

```
flow          lightpenenabled?
    menuselection?          ignore,
inputmenuitem,     tracking?
        inputcoordinates, inputpictureitem;
```

Associated with a flow block is the concept of subflow block which is internal to a flow block. Its use is to represent multi-choice decision nodes and to serve the purpose of a case statement.

1.4 Implementing the Specification

Very high level specification languages do present drawbacks to someone who is more interested in the quick production of an "efficient" system. One major problem is that real compilers for these languages are rare. A second problem is that these compilers still produce relatively poor code because compilation techniques have not advanced as quickly as language design.

The design and specification of correct algorithms really is what we are talking about when we are planning any type of software system. The efficiency of the lower level code generated by a computer for a very high level language is a function of how well the lower level language reflects the ideas and power of the upper level, and of how well the primitives of the lower level language map into machine primitives.

A compiler must be able to handle general cases, but a designer after completing this specification will have a good idea of how he wants and expects the program to execute and how to handle data in the most efficient way. The designer may then resort to hand translation of the specification into a lower level language because the only way to alter the object code produced by the high level language's compiler is to change the original algorithmic design. Although the need to recode is a disadvantage, an advantage is that during the implementation phase of the project the

programmer has available a complete well-documented design. To facilitate the implementation, the programmer should use a lower level language not radically different from the design language in terms of features like control dictions in order to maintain a clear parallelism between the specification and the implementation. Within this framework a programmer will find implementation to be a much easier task. However, not having a good implementation language does not preclude using a very high level language like SETL. In fact, a set of SETL algorithms becomes more important as a design document which provides an implementation scheme for what otherwise is the horrendous task of having to plan and implement simultaneously in a poor language, the approach generally followed.

1.5 Unfeasibility of Using a Lower Level Language for Design

We have discussed two alternatives to system design. One alternative is to use a hierarchy of nested language levels. This presents the danger of producing a massive conglomeration of a design and implementation specific details with the details obscuring the ideas motivating the design. The other alternative is possibly to write the design in a very high level language which will protect the design from low level details and preserve its machine independence, and then to translate by hand the specification into a low level implementation.

One may ask if a compromise between the two alternatives exists, that is, whether there is a high level language which is machine independent and also for which a good compiler can be produced? Why not use a language like Algol 68?

Algol 68 is a very powerful language (Lindsey, 1972), but it is at a lower level than a language like SETL. To use Algol 68, one must have a detailed low level design of data formats and any possible operators before programming. A very good example of these considerations is a paper on extension of Algol 68 to produce a graphics language (Denert et al., 1975). This paper presents a small but detailed sample of definitions and necessary redefinitions of new modes and operators. The illustrated complications indicate that a designer should be wary of using Algol 68 for system specification because he might lull himself into believing that he is closer to an implementation than he really is. On the other hand, "A SETL program may be considered to represent an algorithm as it exists before the detailed choices which govern algorithm realization in a language of lower level (such as PL/I or Algol 68) have been made" (Schwartz, 1975). The preceding quotation really summarizes what we have been trying to say in terms of top-down design in a very high level language and the stepwise refinement procedure which leads to a lower level efficient implementation.

A very high level, powerful language like SETL provides the best medium for a flexible, top-down design of a software system. SETL, being derived from mathematical set theory, allows the clear, succinct and precise display of algorithmic ideas uncluttered by machine dependent details, and thus makes analysis and discussion of the algorithms easier. Not having to worry about type declaration, the designer may use sets and set-theoretic mapping mechanisms in system development to produce a general specification of a data structure with which he may experiment. After experimentation, the designer more readily can construct and specify the interfaces of the application data structure and define specific relationships between elements in sets.

Chapter 2

Distributed Graphics Systems

In the design of an interactive computer graphics system using a refresh crt, one's goals are to provide fast responses to user commands and to provide as much computation power to the user as necessary. The "ideal" situation then is for each user to have his own dedicated high powered computer providing graphics support. Unfortunately this approach is highly impractical for a number of reasons, the most obvious of which is economics. Since most of the work done in a graphics system is essentially input/output, the dedication of a large computer to the support of one graphics terminal, or even several graphics terminals, will be terribly wasteful of processing power. On the other hand, if a minicomputer were dedicated to the support of a graphics terminal, the user would be losing a significant amount of computation power when he needed it. A compromise between these extremes is to use a satellite configuration.

In a satellite system, the graphics hardware is connected to a minicomputer which is in turn connected to a remote, shared main computer. In this type of configuration, the basic purpose of the minicomputer is to provide graphics support for the display terminal, and the purpose of the remote computer is to provide the heavy computational support

of the applications using the graphics system. The question which naturally arises is how much of the work should each machine do?

The more work that the minicomputer can do, the more economical is the use of the remote machine. However one has to be careful not to degrade the system response time since the minicomputer's resources, such as its memory size and speed and its instruction set, are limited generally and placing a heavy burden on its computational abilities may impede its efforts at maintaining the display. If too many or the wrong tasks are assigned to the remote machine then the performance of the system may deteriorate because of the amount of time needed to transmit information and results between the two computers. The problem is to determine which types of routines and combinations of them can be distributed between the two machines.

Besides the basic routines necessarily resident in the satellite computer to maintain a picture on the crt face, to service interrupt conditions from the display processor hardware, and to handle communications with the remote machine, one has to be concerned with routines to convert typed input for screen display, or to convert light pen input into an internal structure, or to respond in an appropriate manner to a set function switch. One has to determine the location of the basic command processor for the system and where routines for doing things like

translation, rotation, scaling, windowing, and hidden line calculations will be. Factors influencing the placement of these routines are their lengths, their execution times, and the size and types of the data structures on which they operate.

The placement of routines to do simple mathematical calculations for such things as graph fitting or the initial calculations of surfaces for display is another problem which is a function of the amount of data to be processed, the arithmetic power of the minicomputer, and the transmission speed of the data link.

In the past people have approached the problem of division of software labor in essentially three ways (Foley, 1973). In these three methods, people have used the minicomputer as (1) a fixed function satellite, (2) a variable function satellite, and (3) an intelligent satellite. Foley proposed three criteria for evaluating these systems. These criteria are

(1) How well is the main CPU relieved of trivial processing tasks and how capable is the system of providing a fast response to trivial user interactions

(2) How much flexibility the application programmer has in dividing processing and data between the two computers and in using special display processing unit (DPU) capabilities

(3) How easily a programmer can learn to use the system.

These three conditions are sound guides for system design except that one probably should replace "special display processing unit (DPU) capabilities" with "all system provided capabilities" because the latter is a more machine independent condition than the former. One system may perform in software the exact function that another implements in hardware. The programmer should be able to make use of it in either case.

A fixed function satellite satisfies the needs of a user who is mostly interested in previewing the results of calculations before committing them to a hard copy. He uses the satellite simply as a faster input/output device. The remote machine does most of the processing.

A variable function satellite system allows a user to affect the satellite's processing by subroutine calls. This type of system can provide a user with a faster response time for trivial picture manipulations by permitting him to assign through the application program on the remote machine several of the minor housekeeping procedures to the satellite. The satellite is still being used as a fancy input/output device.

The purpose of an intelligent satellite system is to try to satisfy all three of Foley's criteria, but frequently designers only attempt to satisfy the first two conditions. We will now review some previous and ongoing efforts.

2.1 Fixed Function Satellites

A system which uses a fixed function satellite is one in which the applications programmer treats the minicomputer as a "black box" of whose internal properties and operations he remains blissfully ignorant. The application programmer writes one program to be executed in the remote computer and treats the satellite and its graphics terminal as any other input/output device. In this type of system, the application programmer has no say in the division of software labor because the tasks of the satellite are preordained. In general, the programmer should have an easy time learning to use the system since he really is working with only one computer.

Kilgour (Kilgour, 1971; Foley, 1973) produced a fixed function satellite system using a PDP-7 and an ICL 4120 (later an ICL 4130). The PDP-7 contained only three things. It held the display file, a communications package to handle data transfers to and from the high speed data link, and an executive to handle interrupts from devices attached to the PDP-7 such as the display, the light pen, and the teletype. The ICL computer was dedicated to graphics and did all the other processing such as display file creation and modification, rubberbanding, and light pen dragging of an entity. The system was able to function because of the dedication of the ICL to graphics and the use of a high speed

data link between the computers. The application programmers used FORTRAN and treated the satellite as an input/output device. The satellite was an input/output terminal whose status could be read. The user's application program was never interrupted. It had to request data from the satellite. A justification for using such a rigid interface was that any future modifications could be made with minimal disturbance to working programs. The user could not affect how the data gathering parts of the system worked.

The system was definitely easy to learn because the only programming a user could do was in FORTRAN. The system did not free the remote machine from handling trivial housekeeping work and the programmer had no flexibility at all in dividing software between the two machines.

The National Center for Atmospheric Research (NCAR) produced a fixed function satellite system in which the minicomputer looked like a card reader/line printer to a CDC 6600 (Gammil and Robertson, 1973). The objective here was to maintain the efficiency of the batch system of the CDC 6600 during and after the development of the graphics system. The basic design goals were simplicity, compatibility, usability, and device independence in order to take advantage of existing software for other devices and to allow the use of the new software with other devices when the interactive system was down. The design specified a high degree of modularity and required the generation of graphics by translation of commands for COM (Computer Output Microfilm Recorder,

a noninteractive device). NCAR also decided to handle all interactions through line by line transmission of card image text by an extension of namelist input/output in FORTRAN. They built a hierarchical system in which the user program was on the highest level and the satellite and the display were several levels away. The user was not supposed to know anything about the satellite software.

The software in the satellite here can do some editing of text and light pen inputs before transmitting the information to the CDC 6600, so the 6600 is spared a few trivial tasks. The user has no ability at all to assign software to the satellite processor because of its treatment as a well protected (through the system's hierarchical structure) black box. The system must be very easy to learn because of the use of FORTRAN and its designed similarity to non-interactive graphics packages already available at the site. The authors also noted that they were investigating the possibility of replacing the refresh tube with a storage display due to the nature of most of the work at NCAR.

One other system that falls into the fixed function satellite category is one produced at the Research Laboratories of General Motors (Dill and Thomas, 1975). The functions of the satellite computer are to build and modify display files on command from the host, to accept light pen, function key and other inputs for sending to the host, and communication with the host by teletype simulation. In this system a

DEC GT40 is connected to an IBM 370/168 with a TSS operating system.

The host is relieved of some trivial functions plus the building and modification of display files. The application programmer though has no way of assigning more work to the satellite. The system is simple and the authors claim success in the tasks it was used for. They also feel that it could serve as a basis for an upgraded system. They suggested that, in the future, control logic codes could be added to the information transmitted by the host to the minicomputer for handling events occurring in the terminal. They also suggested implementing a means of saving once transmitted instructions from the host to eliminate retransmission of complicated menus.

2.2 Variable Function Satellites

In a variable function satellite system, the user still does not program the minicomputer, but he can control the amount of work it does by subroutine calls. For example, the user's program in the remote machine can cause the satellite to take certain actions such as turning picture segments on and off when it is better for the remote computer to have control over such actions than for the satellite. The GINO system is a variable function satellite (Woodsford, 1971; Foley, 1973).

In GINO the satellite processor performs many basic display services such as light pen detection of entities and correlation of the entities to user names, varying the intensity of subpicture entities, and the addition and deletion of subpicture parts. It can also do rubberbanding, draw horizontal or vertical lines, and supply extra material upon detection of an entity, if the application program has enabled these functions. The satellite does not immediately transmit every user action to the remote machine for servicing. It queues actions until a "terminal" action such as a light button detect or the entrance of a special keyboard character occurs. Thus the remote computer is saved a lot of trivial overhead work. The satellite may interrupt the user's program with a request to transfer data, but the user's program initiates data transfers to and from the satellite. Since users write their programs in FORTRAN and the satellite acts like an input/output device, the system is easy to learn.

With almost all the systems we have discussed so far, the principal user language for application processing is FORTRAN. The universality of the language means that most users of these systems will already be acquainted with it and will have to learn only the interfaces with the graphics subroutine library. However, these interfaces are rigid and FORTRAN is awkward so that an application programmer's usage of graphics facilities will generally be limited to

the more elementary ones or he will be forced into designing contorted extensions to satisfy his needs. He also will not be able to extend through FORTRAN the graphics capabilities of the satellite to ease the burdens of system control upon the main computer.

Work at the University of Michigan has produced proposals for hardware/software designs for so called high speed/low cost interactive graphic systems (Boardman, 1974). These proposals require very elaborate hardware configuration schemes to handle high speed transmissions for many terminals. Hardware interfaces to data concentrators (minicomputers made to look like disk units to a remote host) will do most of the work such as data transfer, character conversion, and data formatting to make the concentrator look like a real disk. A data concentrator can transfer information it receives from the mainframe to several minicomputers servicing graphics terminals. It can also transfer information from a terminal to the remote computer.

Boardman does suggest using a single language like FORTRAN in order to program the system (mainframe, data concentrator, and terminal minicomputer) as if it were one computer. He wants to minimize mainframe overhead in order to service many terminals, so he speaks of passing subroutine parameters via the data concentrators to the minicomputers from the mainframe to free the mainframe from image processing tasks. He comments that intermachine

program execution is not necessary but that the minicomputers should be able to do some image manipulations and to handle local input/output either by request from the mainframe or from the terminal.

The hardware here is very sophisticated and expensive, despite decreasing technology costs, and the software in the satellite computers do more work than that of a fixed function satellite but it is not clear how easy it would be for an applications programmer to redistribute more of the software to the minicomputer because of the elaborate and well defined interfaces necessary to handle several terminals through the data concentrators. Since a single programming language will be used, one imagines that the system will be easy to learn.

Wycherley describes a system which appears to be partially in the category of variable function satellites and partially in the next category of programmable satellite systems (Wycherley, 1972). The system is supposed to allow for three types of users. The first type is the user who is simply interested in a "black box" to handle graphics input/output. The second type of user is one who will approach the system as a variable function satellite system. The third type of user is one who will write special programs for execution by the satellite. Wycherley does not specify the type of programming language, formula oriented or assembly, used to program the satellite.

The satellite processor can service three terminals so its software is reentrant and each terminal gets time slices for processing.

The user writes an application program in a higher level language to reside in the host computer. The program communicates with the operator's console through calls to the Host Graphics Package. The package allows the user to build and modify pictures on the screen and to obtain information about the operator's actions. The Host Graphics Package translates the subroutine calls into commands to the satellite in Director Code format. The Director Codes are commands for picture building, commands to control the method by which the operator's actions are processed, and commands to the satellite to return information on actions having occurred previously.

In order to handle interactions, the user program in the host machine must tell the satellite in advance what options the operator has and how to respond to the input from the user; that is, whether to process the input locally or transmit it to the host for processing. These instructions are put into an Action Table in the satellite. The Action Table entries contain information such as which routine does the next phase of the processing. A "black box" user will use the system supplied Action Tables. A variable function satellite user can define his own Action Table of calls on system supplied processes.

A really sophisticated user can write his own Action subroutines to be called from the Action Table. However, based on this article, one views this as, at best, a variable function satellite system. Even the"black box" type of user must understand the logical structure of the display file because it represents the logical structure of his picture. The writing of new software has to be more difficult. The Action Table entries are very complicated for describing input handling and phase transitions. New Action subroutines for the satellite must be on a low level, which means the user must be bilingual because he writes his host application in a high level language. System modification must be extremely difficult because of all the low level details that have to be considered. The competition among the users of the three terminals attached to the satellite for the satellite's resources such as space and time, would limit the amount of software customization by any of them. The satellite relieves the mainframe of trivial tasks involving the light pen, text editing, and keyboard inputs. It does not transmit operator interactions to the remote machine until a "terminal" action occurs. But the remote machine still does a major share of the work. It initially builds the display files and it initiates processes at the satellite. The ease of learning the system is a function of the extent to which one wishes to use its facilities.

2.3 Programmable Satellite Systems

Finally we come to programmable satellite systems. These are systems in which the user may also program the minicomputer to perform tasks in support of his application besides just providing maintenance of the display and acting as an input/output terminal for the remote machine.

The Graphics-2 system developed at Bell Laboratories was one of the earlier attempts at this type of system (Christensen and Pinson, 1967; Van Dam et al., 1973, 1974; Foley, 1973). The hardware for the proposed system was a Honeywell 645 mainframe connected to an 8K PDP-9 satellite via a 2000 bps communication link. The two CPUs were to work asynchronously, although all data base modifications in the satellite would be done automatically in the remote machine, whether or not the user wished this to occur. The reason was that the programmer wrote a single program in GRIN-2, a graphics interaction language for data structure manipulation and the handling of real-time inputs, for compilations for both machines and the version for the PDP-7 automatically queued all inputs and data base modifications for transmission to the remote machine. This way the two CPUs appeared to be one to the user. The programmer used GRIN-2 to do interaction handling, data structure interrogation, and data structure display. The analysis program for the application ran on the 645 and was written in the language

of one of its compilers. This meant that the programmer had to be bilingual which made the system more difficult to learn. The analysis program interfaced with the GRIN-2 program through subroutine calls for accessing the hierarchical data structure in the main CPU. The application program had to decode requests only for analyses to be performed and not for updating the data base. Apparently the system was not successful enough so that the satellite was being used in a stand alone mode.

A system similar to Graphics-2 was developed by UNIVAC for an 1108 linked to a 1557 display controller acting as the satellite computer. A 1556 display console was connected to the satellite computer (Cotton and Greatorex, 1968; Van Dam et al., 1973, 1974; Foley, 1973). The user programs the satellite with an interpretive language called ICT (Interactive Control Tables). It is an assembly-like language and can be used to provide user feedback and any kind of interaction. In ICT one can modify the data structure, perform computations, call subroutines, and format messages for transmission to a remote computer. Because the language is so powerful a lot of work can be done on the satellite's data structure without any help from the remote machine. The application program on the 1108 is written in FORTRAN. A larger copy of the hierarchical data structure, called the Entity Tables, is kept in the 1108. It, or subparts of it, can be easily transformed

and transmitted to the satellite by subroutine calls. The application program decodes messages from the satellite to determine if it is to transform the data base or to perform an analysis. The satellite can continue serving the user while the 1108 transforms its data base because the satellite has already performed the transformation on its own copy of the data base.

Among the advantages of the system are that the ability to dynamically load new satellite programs allows for run-time changing of the system environment and that the amount of work the satellite can do allows fast user response. The two major disadvantages of this system are that the programmer must be bilingual and that unlike the Graphics-2 system he must always be aware that he is dealing with two computers.

Van Dam reports very briefly on Kulick's THEMIS system which is a distributed processor graphics system (Van Dam et al., 1974). The system allows the applications programmer to vary the work load nondynamically between the processors through the use of three languages, a host interaction language, a host data base language, and a terminal interaction language. The review comments that the run-time system is quite complex due to the three languages. The need to know three languages makes use of the system more difficult.

Two much more ambitious efforts at producing systems

which are flexible in solving the division of labor problem and also easy to program have proceeded at Brown University and at the University of North Carolina (Van Dam, 1971; Van Dam et al., 1973, 1974; Michel and Van Dam, 1976; Foley, 1973; Hamlin and Foley, 1975; Hamlin, 1976). Both universities have tried to reduce the programming problems through the use of a single language for the remote and the satellite systems. Brown initially used the Language for System Development (LSD) but later switched to a version of Algol-W, and North Carolina used a subset of PL/1 with a preprocessor to handle added statements for specifying configuration information.

At Brown, the hardware of the system is more elaborate than elsewhere with a great deal of emphasis on microprogramming. The belief here is that the satellite system should possess "critical intelligence". This means that the satellite should have enough computational power to be able to handle more than 50% of the processing required for graphics and to be able to act as a general purpose computer for any user. This is essentially the argument that Brown employs to justify the use of microprogramming as a means of compensating for insufficiencies in the architecture of the minicomputer. These insufficiencies include minimal core and/or an inadequate instruction set. Frequently used blocks of code or procedures can be microprogrammed and made part of the instruction set. The hardware for the Brown University Graphics System (BUGS) consists of a local processor, a

display processor, a Super Integral Microprogrammed Arithmetic Logic Expeditor (SIMALE), and an interface to an IBM 360/67 under CP67/CMS. The local processor and the display processor replace the single satellite computer. Both of these processors are microprogrammable Digital Scientific Meta 4 computers. The local processor has been given a 360-like instruction set through microprogramming. The display processor is devoted to servicing a Vector General terminal. The purpose of the SIMALE unit is to do homogeneous transformations, windowing, and clipping to improve the performance of the satellite system. Communications between the remote machine and the satellite system are handled through high level subprogram calls processed by the operating system so that the application programmer does not have to worry about protocols, timing, or interface characteristics. The most ideal feature in terms of the flexibility in dividing the software labor is that the system allows interprocessor communication and real-time reconfiguration of the software. The operating systems allow the calling of routines residing in the other machine and the passing of parameters between machines.

Thus the Brown system was deliberately designed and configured in order that the satellite components of the system would do most of the processing such as real-time transformations and clipping, providing prompts and feedback, local attention handling, and data base editing. It has

great flexibility in dividing software labor because of the ability to reconfigure software while the system is operating. Allocation of procedures in the application is delayed until execution time. The system is easy to learn because the programming for the host and the local processor is done now in Algol W, although the first application, HUGS -- a two dimensional drafting package, was written in a macro assembler language common to the host and the local processor. The people at Brown found from their studies of the application that groups of procedures with a lot of interaction migrated from the host to the satellite as host availability decreased. They were studying how to automate the initial configuration of software.

A major disadvantage in applying the methodology developed at Brown to other sites is the economic cost of configuring a hardware system similar to the one there. In his 1971 article on microprogramming, Van Dam estimated the cost of the Brown system at $145000. Not many places could afford a hardware investment of that size.

People at the University of North Carolina produced a system for Configurable Applications for Graphics Employing Satellites (CAGES). Although Foley had previously done modeling studies of computer graphics systems in which performance was a function of the capabilities of the graphics hardware and the computational requirements of the application (Foley, 1971), the hardware was not as

extraordinary as that assembled at Brown. The satellite computer is a PDP 11/45 and the host computer is an IBM 360/75. The emphasis at North Carolina has been to develop a PL/1 based compiler system that will produce configurable software and to produce an operating system that allows either computer to call a subroutine resident in the other, to pass parameters to the other machine, and to access global variables currently residing in the other machine. In order for these things to occur efficiently, the preprocessor to the PL/1 compiler has to analyze the whole system of programs to determine when any sort of intercomputer communication will take place. One type of optimization is to obtain access to global variables at the beginning of a subprogram to assure their availability rather than have several intercomputer transmissions during the execution of the subprogram. The system also uses prepaging to take advantage of the time spent in accessing information in the other machine and thus reduces the amount of useless idle time.

Among the goals was the desire to permit an application programmer to write a program capable of running on one machine, thus any programs which might run on the satellite computer had to be written in a subset of PL/1 called PLCD. Also a goal was the ability to "tune" the division of software labor based on the performance of and use of facilities by programs. Originally it was thought that the

programmer should write his software as if it were to run on one machine, but results indicated that applications performed better when initially designed with distributed execution in mind. Another goal was that the programmer should not have to worry about intercomputer communication. Finally, by using a language like PL/1, possibilities for system portability were improved.

An interesting feature of this system is that global variables belong to no particular computer but move back and forth as they are needed. This means that before accessing a global variable both computers have to check for its presence.

North Carolina implemented two applications and studied them in different configurations and applied a network flow analysis algorithm to them. One application was an interactive curve fitting program to fit a function to a given set of input data points. The other application was a program for creating and manipulating block diagrams. They found that an application should be analyzed during the design phase so that its structure could be modified to accommodate likely configurations. Certain optimal configurations were not possible because of hardware restrictions. Also the application programmer should design a modular structure carefully and choose an initial configuration.

The major drawback with the system is that in order to reconfigure the software between computers all the programs have to be recompiled. This is necessary because of the analysis done by the preprocessor to optimize inter-computer communication. Another problem is the syntax differences between the PL/1 compilers for the two machines. The compiler for the satellite is more restricted than that for the host. One of the conclusions of the North Carolina study was that a single language with distinct code generators would simplify the task.

One feature common to both the Brown system and the North Carolina system was the ability to redistribute particular routines instead of whole processes consisting of related routines. Both studies verified the obvious conclusion that a group of highly connected routines would tend to move instead of individual routines during reconfiguration.

A major drawback in using the work at either Brown or North Carolina as a model for development at other sites is the extensive amount of time, effort and financial investment needed for the development of new operating systems to handle intercomputer communications. To start from the beginning in developing these systems requires an enormous design and programming effort outside of that of the graphics system. A further drawback of the Brown method is the large initial hardware investment to give the satellite processor enough computational power to achieve "critical intelligence".

Chapter 3

Specification of an Intelligent Satellite Graphics System

We will use SETL to provide a high level design of components of an intelligent satellite graphics system. We will apply SETL to describing command processing, display image maintenance, and asynchronous interrupt processing. Later we will also use it for data structuring,picture construction, and segmented display file generation.

3.1 System Component Specification

One of the advantages of this specification is that it provides a means of describing the blocks of routines which make up the components of the system. An early knowledge of these blocks makes it possible to implement an intelligent satellite system which has a simplified, minimal form of intercomputer communications. A block consists of all those procedures that should be in the same computer to execute efficiently. Later, implementation details may bind certain processes to a particular machine out of necessity but in the design phase, these blocks are machine independent and are capable of migrating between computers. In the implementation, we avoided the highly complex operating systems of the Brown University

and University of North Carolina systems (Michel and Van Dam, 1976; Hamlin, 1976) which had to take care of intercomputer procedure invocation and the associated parameter description and data transmissions. Having component blocks made it possible to restrict intercomputer communications to statements and requests for service from the satellite to the remote host and to answers from the host to the satellite.

The system consists of three main processors. One is the interrupt processor, the second is the display maintenance processor, and the third is the command processor. The first two are bound to the satellite and the third is divided between the two computers although most of its work is performed in the satellite. On the satellite, all three processors function as coprocessors in order to maintain a picture for the user while a command is processed. The display maintenance routine keeps track of time for refreshing the picture and queues graphic segments for output to the display terminal. The command processor naturally processes commands but it contains components to add to, modify, extract information from and to save the description in the graphics data structure that describes the current display. Other components handle system testing, computer communications, and the application programs. Some of these routines will be described in SETL algorithms later. The interrupt processor

operates asynchronously handling interrupts from the display terminal and input/output between the computer and the terminal.

The total length of these algorithms as specified in SETL is approximately one-third the total length of the corresponding routines written in the language of implementation. The obvious advantage to a reader is the relative ease of understanding the programs without the need of setting up a mental filter to block out the low level detail "noise" and housekeeping procedures necessary in an actual implementation.

The main routine of the command processor is *commandprocedure*. It transfers control to subprograms which process the input from the keyboard. These subprograms will then, in turn, call on routines to build and modify the data structure which describes the display. After regaining control from these routines, the command subprocessors will call another program to create graphics segments corresponding to the changes in the data structure.

Now we will describe the three processors and present the SETL specifications of their algorithms.

3.1.1 The Command Processor

The satellite's executive routine will regularly call *commandprocedure* to determine if any keyboard input from the user is waiting for processing. If input is waiting,

the routine will determine based on a previously set flag whether to treat the input as an attempt by the user to enter a new command or as a continuation of processing for the last command entered.

If the input is a new command attempt, *commandprocedure* checks *subprocess*, a function which maps commands into the names of the routines which process them. If the command is illegal, an undefined mapping, an error message is displayed to the user and the routine returns control to the executive procedure.

If the command is legal or the input is a continuation of the previous command then the map *machineof* is checked to determine whether the satellite computer or the remote host processes the command. If the satellite processes the command, the routine *kbinputhandler* will call the appropriate processor.

When *commandprocedure* regains control from *kbinput-handler*, it can reset its parameters to allow a new command to be entered; or it can output a message, possibly an error message, and reset its parameters for more processing; or it can output a cue to the user to help him continue entering information for the command. For example, the command "SCALE" will produce the cue "SCALE=", and the command "TRANSLATE" will produce successively the cues "DX=", "DY=", and "DZ=".

The algorithm which follows uses a flow block to describe the control logic of the procedure.

The definition of a token representing a decision node in a flow block is of the form

question := expression

The final value of the expression is boolean although it may have other effects. The node

legalcommand: subprocess(command) ne Ω

checks to determine if the mapping *subprocess* is defined for the "command".

The other tokens in the flow diagram are labels for statements or groups of statements to be executed if a particular decision is made. These labels are defined as

label: {<statement>}$_1^n$

If a "+" follows a label then after execution of the label's statements, program control passes to the decision node or statement group below it. If a "," follows the label then the program exits the flow block after executing the label's statements and processing continues with the statements following the flow block unless a "return" is executed within the block.

The subprogram *kbinputhandler* demonstrates how SETL's mapping mechanism can be used to invoke subroutines. In a single statement, we can execute a subprogram which is itself the value of a function of a parameter. The designer can easily change the capabilities of the command processor by simply modifying the *subprocess* without recoding

commandprocedure or *kbinpathhandler*. An implementation language with this feature would be very powerful and well adapted for system tasks.

```
define commandprocedure;                    /* a subroutine which processes crt keyboard
                                                                              commands */
   if kbmsgpr lt 0
      then                                  /* there is no input ready for processing */
      return;
   end if;
flow                                    newcommand?
                                              /* multiline  input command */
                        incommand+          machinechoice?
                        legaltoken?       sat,       remote,
                legalcommand?   comerror,
      machinechoice?  comerror,
     sat,      remote;
newcommand := comflg ne multilinegood       /* are we currently processing a multiline
                                                                    input command? */
   or comflg ne multilinebad
legaltoken := type eq alpha                 /* is the token that represents the command
                                                                  alpha-numeric?  */
legalcommand := subprocess(command) ne Ω    /* is the command defined? */
machinechoice := machineof(command)         /* is this command processed by the satellite*/
                  eq 'sat'                  /* or the remote machine */
```

```
incommand : comptr = 1;                    /* set pointer to the first position in the
                                              keyboard line buffer, kblinbf */
   nexttoken(kblinbf, comptr, command,     /* pick up the first token of the line, it */
      comlnth, type);                      /* should be the command with number of charac-
                                              acters equal comlnth and type equal alpha*/
sat : kbinputhandler(command);             /* call the keyboard input handler to call the
                                              processing routine corresponding to the
                                                                          command */
remote : transcommand(command,kblinbf);    /* transmit the input line to the remote
                                              machine for processing */
comerror: mvhresp(illegal  command');
   comflg = badcommand;
   return;

                                           end flow;
```

```
if comflg eq badcommand                    /* determine response to user after command
                                                has been processed */
   then
   mvhresp('illegal parameter');
end if;
if comflg ne multilinegood
      or comflg ne multilinebad
         then
         kblinit;                          /* initialize the keyboard input line to accept
                                                more input */
         kbmsgpr = -1;                     /* reset message processing flag */
         return;
         else                              /* more information to be input */
         cueget;                           /* pick up the user cue */
         kbmsgpr = -1;                     /* reset message processing flag */
         kbin = -1;                        /* reenable  keyboard input */
         return;

end if;
end commandprocedure;
```

```
define kbinputhandler(command);
    subprocess(command)( );

    return;
end kbinputhandler;
```

```
/* this routine calls subprocess(command) */
/* subprocess is a set of ordered pairs where
   the second element is a character string
   which is the name of a command subprocess.
   Subprocess(command) returns the string and
   the second pair of parentheses forces SETL
   to treat the string as a subroutine call.
   this formulation allows easy addition and
   deletion of subprocessors to the system */
```

3.1.2 Display Image Maintenance

The display maintenance processor's task is to check the display controller's frame clock counter, incremented by the interrupt routine, to determine if the display needs refreshing. If the display needs refreshing, then the counter is reset and all the graphics segments are queued up for output. If the system has a response message to a user's input, the message is also added to the queue for outputting. Finally, the input line also has to be refreshed so it is added to the queue. The routine *clock* represents this processor in a simplified form.

The following program illustrates the use of a tuple as a first-in-first-out (fifo) queue.

```
define clock;                  /* this subprogram checks the frame
                                  clock counter set by the interrupt
                                  routine to determine whether the
                                  crt display needs refreshing. if
                                  it does, the clock counter is
                                  reset and the display and trans-
                                  formation segments are queued for
                                  output */
   if fclock lt 0              /* is it time to output  */
      then return;             /* no */
   end if;
   fclock = -interval;         /* reset the clock */
   (3<=∀s<=#gstr)
      seg = gstr(s);
      if seg ne nℓ
         then                  /* segment has not been deleted */
         queue=queue+seg;      /* add it to the queue */
      end if;
   end ∀s;
   if userresponseseg          /* is there a response message to */
         ne nℓ                 /* a user command  */
      then queue = queue       /* yes */
        + userresponseseg;
   end if;
   queue = queue+kblinbf;      /* refresh the user  input line */
      return;
   end clock;
```

3.1.3 Asynchronous Interrupt Processor

The interrupt processor runs asynchronously and steals time from either the command processor or the display maintenance processor whenever necessary to service interrupts from the display terminal. The processor handles signals indicating the end of transmission of a graphics segment, the display's frame clock cycle, light pen input, function switch status, and a keyboard character waiting to be input. The processor acknowledges any interrupts and does any appropriate processing. If graphics segments are in the output queue and the display terminal is not busy then the processor will transmit a segment to the terminal. The processor is described by the routine *interrupt*.

Here we have specified the functions of a low level routine in a very high level language.

The expression of an interrupt processor at such a high level has a couple of benefits even if the actual implementation is significantly different. First of all, it illustrates the type of processing that is necessary and does so in a machine independent manner. It is also a good teaching tool because it does not burden students with the need to know particular bit configurations or special hardware instructions. It also demonstrates that an interrupt processor is a special subroutine which first must recognize the state of the hardware before proceeding with its work

and afterwards restore that state before relinquishing control.

```
define interrupt;              /*this routine is called by the
                                 hardware. it checks for hardware
                                 interrupts caused by the sending
                                 of the last data item of a
                                 display or transformation segment;
                                 the graphics terminal frame clock;
                                 or the input of a keyboard charac-
                                 ter. it also outputs a display or
                                 transformation segment waiting in
                                 an output queue if the graphics
                                 terminal is ready to accept output*/
  savestatus;                  /*save hardware status and then
                                   determine source(s) of interrupts*/
  if displayinterrupt eq t     /*did last item of segment go out */
    then                       /*yes */
      acknowledge(display);    /*signal hardware an acknowledgment*/
      displayinterrupt=f;
  end if;
  if frameclockinterrupt eq t /*frame clock */
    then
      acknowledge(frameclock);
      fclock = fclock+1;       /*increment clock */
      frameclockinterrupt = f;
  end if;
  if lightpeninterrupt eq t    /*beam output detected by pen */
    then
      acknowledge(lightpen);
      <ℓpx,ℓpy>=<beamx,beamy>;/*read beam coordinates*/
      lightpeninterrupt=f;
  end if;
```

```
      if keyboardinterrupt eq t   /* keyboard char for input*/
        then
          acknowledge(keyboard);
          keyb=keyboardcharacter; /* save the character */
          keyboardinterrupt=f;
      end if;
      if functionswitchinterrupt eq t /* function switch(es)
                                              depressed */
        then
          acknowledge(functionswitch);
          switches=switchstatus;      /* read switch status */
          functionswitchinterrupt=f;
      end if;
                                  /* if the display is not busy,
                                     check to see if a segment is
                                     waiting for outputting */
      if not displaybusy          /* hardware status flag check */
        then
          if queue ne nult
            then                  /* queue is not the null tuple so
                                           a segment is waiting*/
            segment=hd queue;     /* dequeue the segment */
            queue=tl queue;
            transmit(segment); /* transmit segment to display
                                  terminal -- hardware sets
                                  displaybusy status flag to t*/
          end if queue;
      end if;
restorestatus;
return;
end interrupt;
```

3.2 System Implementation

Once the high level specification of the system is complete, the designer has to worry about its implementation. If space and time considerations are not important and the only interest is in studying the algorithms, he can compile the specification to get an object program to run. If efficiency also matters and the language of specification is on the same level as SETL then the designer must do a hand translation of the design into a lower level language. In the case of the satellite graphics system just described, we have a real-time application which is to operate mostly in a minicomputer so we definitely are interested in an efficient implementation.

Besides having an awareness of the operating environment, a programmer can do several other things to contribute to the efficiency of an implementation. For example, he can design interfaces carefully and consistently so that any future modifications will not cause havoc and a major recoding effort or the appearance of a "kluge". He can also specify data structures and formats early in a detailed but flexible way as an aid to interfacing. Still another contributing factor to efficiency is to write modular well-structured code, but this should be relatively easy if the original high level specification was done in that way.

One means of promoting these efficiency contributing events to occur is to select a low level language which has

primitives that can implement several of the primitives of the design language and which has a subset of the control structures of that language. We also want to preserve the machine independence and portability of the design so this suggests using a language that is machine oriented but also on a high level.

For our work, we used the system implementation language called LITTLE. This language, like SETL, has developed under the supervision of Professor Jacob T. Schwartz. It produces efficient, machine independent code which is then translated into the assembly language of the target machine. The machine independence results because the language is designed for an unknown machine and therefore cannot permit references to registers or machine addresses nor allow embedded assembly language to appear as do minicomputer languages like BLISS-11, PL516, or ALIAS. The language has macros and conditional compilation directives which make coding easier and also provide a high level way of allowing for machine dependent characteristics such as word size and character size. Defining new sets of macros or compiling a different group of statements make the portability of programs to other machines much easier. LITTLE has a wide range of control dictions which encourages structured programming. It has bit, field and character operators so that a shift operator is not needed. Its only data types are bit and character strings and real numbers so that any operation can

be performed on any quantity (Stuart, 1975).

At the start of the LITTLE implementation we produced a detailed but flexible macro specification of the data formats and structures. This produced well defined interfaces and permitted us to avoid many problems during the writing of subprograms or modules. Although at one point we needed to significantly revise part of the data structure design, it did not require much of a reprogramming effort because the design was very modular and the macro facility provided a very easy means of redefining formats.

One of the important advantages of writing a very high level specification is that the binding time of particular implementation dependent interfaces is put off. The basic algorithm is always available during experimentation with different strategies.

Chapter 4

Process and Communication Structures in Experimental Graphics Systems

4.1 Overview

At the Courant Institute of Mathematical Sciences at New York University (NYU) we have also attempted to produce a configurable graphics system in which the division of labor problem is more easily solved, but we have used a slightly different approach and philosophy than those of the Brown and North Carolina groups. We agreed with Van Dam that the processing power of a minicomputer was a function of its instruction set and architecture, but, perhaps out of the realization that we had no funds to replace or upgrade the minicomputer available to us, we determined to take advantage of the minimal processing power available. We emphasized the principle that software should be written as if it were going to run on only one machine by assuming, whether rightly or wrongly, that all the software would run on the satellite computer. In spite of the truly limited computational power of our satellite computer, this philosophy proved successful as new software was added.

The hardware we have consists of a Honeywell 316 computer, a teletypewriter, a disk, a link to a remote CDC 6600 and a Vector General display with a keyboard.

The Honeywell 316 computer used as our satellite does not meet the standards of the "critical intelligence" definition (Van Dam, 1973). It has almost no intelligence at all. It has a single accumulator and one index register. Also, it lacks hardware instructions for both multiplication and division and this significantly increases the cost in terms of time and space of many programs such as image transformation routines and application programs doing real arithmetic.

Even though the satellite lacked "critical intelligence", we wanted it to do more than just act as an input/output device between the user and the remote machine. We wanted it to do nontrivial local processing such as performing image transformations, providing feedback and prompts to the user, and modifying the data base. We have succeeded in having it do image transformations such as rotation about the coordinate axis, translation, scaling and real-time continuous rotation. The terminal user receives text cues and messages and he can create and save new pictures and thus modify the data base. He can retrieve previously created display files from the remote machine. We also want the terminal user to be able to run as much of his application as possible on the satellite. He has the ability to experiment with shifting the application processes during real-time without the need for recompilation.

Our success in satisfying these requirements with an extremely primitive machine suggest that Van Dam's definition of "critical intelligence" puts too much emphasis on

the basic computational power of the hardware. However, our results show that a lack of hardware "intelligence" can be compensated for by software "intelligence". The Van Dam philosophy might be labeled "genetic" intelligence whereas we demonstrated that "environmental factors" (software) can be an effective substitute.

One of the major goals of the effort is to be able to implement new software modules as easily as possible. The LITTLE compiler permits this. Using one set of macro definitions for the satellite computer and another set for the remote computer allows the same LITTLE program to execute on both computers. Hence we were able to use the CDC 6600 to check out almost completely the correctness of the software modules to run on both machines without having to worry about real-time complications. With most of the errors removed during debugging runs on the CDC 6600, it took only four compilations for the satellite computer to have a major part of a stand alone system operating successfully in less than two days. The plug-in capability due to the machine independence of the LITTLE source code facilitates the determination of the best location for a new module relative to the rest of the software.

Unlike the system designs at Brown and North Carolina which permit explicit intercomputer subroutine calls, our design does not permit this amount of flexibility which makes intercomputer communication much more complicated than necessary and either as in the case of North Carolina

requires a compiler preprocessor to optimize the input/output between computers or as in the case of Brown requires the run-time monitor to do procedure call resolution and maintain symbol table information for parameter formatting. Another design difference is that instead of permitting either the terminal user or the remote machine to initiate an operation at the satellite, our design takes the view that the terminal user initiates operations, and if necessary the satellite will initiate an operation at the remote computer to provide some computational support to the satellite or to execute some analysis for the application. This view is consistent with the goal of writing all the software as if it were to run on one machine, the satellite computer. The software which can move between machines is the command subprocessors of the application and their related support routines. The very high level modular design described earlier in this paper makes it easier to conceive of the blocks of processes that may migrate between machines.

No software modules are actually transmitted between machines. Compilations are done for both the satellite and the remote machine on the 6600 and those command subprocessors which can execute in either machine appear in both compilations. If a user decides a particular command ought to be processed in a particular machine, he can enter a command at the keyboard terminal to assign that command's processor to a particular machine. He simply can enter

ASSIGN commandname: SAT

or

ASSIGN commandname: MAIN

where the first statement assigns "commandname" for processing on the satellite and the second statement assigns it to the remote machine. The net effect of one of these commands as far as the system is concerned is that one bit in the command tables is either reset or set. The system does not have to reset any linkages nor does it need recompilation.

As in all intelligent satellite systems, a main goal is to minimize communication transmissions between the two computers and to minimize the amount of work done by the remote computer in support of the display terminal. We feel that maintaining the center of system control in the satellite is the best method of minimizing the demand on the remote computer. One of the users of the Brown system has made the point that a system should be inexpensive to use so that the user does not feel that he is working under pressure (Strauss, 1974). Obviously, the remote machine is the more expensive one to use because of its power and peripheral utilities, and its more elaborate accounting procedure for billing its customers. Hence one would want to use it the least and only when necessary which means that as many command subprocessors as possible should execute on the minicomputer.

Although intercomputer subroutine calls are not allowed, the user will not have to worry about elaborate

protocols or any of the other hassles associated with intercomputer communication. The transmission monitor routine will take care of the formatting or interpreting the data. The current implementation calls for all transmissions to the CDC 6600 to be in the form of batch "JOB" streams. One reason for this was that it permitted us to take advantage of software already written which allowed the Honeywell 316 to be used as a remote job entry terminal. Second, since we wanted to minimize use of the CDC 6600 and because in most cases we did not need to use the full graphics system to do processing at the remote site we hoped to get better responses by submitting short jobs for processing than by trying to keep a cooperating program executing while the graphics terminal was in use. The CDC 6600 is too busy with other tasks and the operating system too complicated to be modified while the current work was in progress to allow for special graphics handling for the terminal. Another advantage was that by sending a job control stream we could easily use the file system of the 6600. Later, the system must be modified so that the satellite would look like a terminal in the time-sharing system as a means of improving performance.

All transmissions from the CDC 6600 to the Honeywell 316 are binary blocks with six word headers. The transmission monitor in the satellite will interpret the information in the headers to handle the block appropriately. A description of the communication formats appears in Appendix E.

More block types can be defined if the need arises. At the moment, only blocks of type 0 and type 4 are implemented. These are the block types which represent display files and messages to the user. Users will not need to know anything about communication formats.

The disk attached to the satellite computer permits the creation of an overlay library and serves as a storage device for display files corresponding to defined pictures and for other information. The overlay library enlarges the number of routines which can execute on the satellite. The overlays for the basic graphics system are in five groups: programs used for creating and displaying pictures; programs used for saving picture definitions; programs used for doing subpicture transformations; programs used by the system programmer for examining and testing the system; and programs used for communication with the remote computer. The application programs require one or more additional overlays.

Since one of the main goals of any graphics system is to be as efficient as possible, we made a strong and reasonably successful effort to eliminate redundant or similar code sequences and to make single routines serve many purposes. Because LITTLE does not permit multiple entry points in its subroutines, many routines became finite state automatons with a state indicator and/or a switch setting describing the path of its processing. This means that the overlays become libraries containing related

routines and that will help to reduce the amount of disk input/output. For example, the use of one transformation processor implies the use of another transformation processor. Both of these processors use many of the same subprograms so that all transformation related routines can be on the same overlay. Another example is the building of pictures from primitive elements and previously defined subpictures. Picture building routines form another overlay. Those routines common to both picture building and picture transformation appear in the graphics system root.

The addition of new command processing software to the system is fairly simple. New compilations contain the new software with a new entry describing the command added to the command table. The first routine of the overlay of which this processor becomes a part has an invocation statement for the processor added to it. The object code of the compilation then replaces the old overlay.

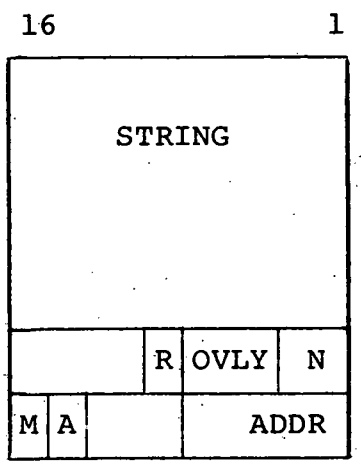
As mentioned earlier, one of the purposes in developing a top-down very high level design and using a system implementation language like LITTLE is to write machine independent programs. In reality many programmers face the "problem" of knowing the characteristics of the machines for which they are writing code. This causes them to try to take advantage of certain machine characteristics or to attempt to avoid certain machine insufficiences. For example, the lack of hardware multiplication or division instructions

and poor manufacturer supplied routines for the Honeywell computer forced us to write customized arithmetic routines for our calculations. A nice characteristic of LITTLE is that by using its macro facilities we can eliminate these machine dependent calls by supplying macros which change a subprogram call to an explicit calculation.

Finally, a user can write his whole application in LITTLE because he has available the equivalent of a large portion of the FORTRAN library.

4.2 Implementation Details and Interfacing Considerations

To be more specific about the actual implementation let us consider the procedure for adding subprocessor modules and the means of system interfacing in more detail. First, an entry in the command table has the following format:



where STRING is the command name (left justified and blank filled); N is the number of characters in the name (up to ten Vector General ASCII characters); OVLY is the index

number of the overlay that contains the processor for this command; R is 1 if the satellite does or 0 if it does not expect the remote computer to reply with data when it processes the command; M is 0 if the command is assigned to the satellite or 1 if the command is assigned to the remote computer for processing; A is 1 if the command can be assigned to either machine for processing and 0 otherwise; and ADDR is an index for the command. When a new processor is added to the system, this type of entry has to be included in the command table.

The command subprocessor must be able to read the keyboard input line and also be able to issue messages to the user. Global variables associated with these abilities are kept in LITTLE namesets called COMINF and KBLINE. Namesets in LITTLE are like COMMON areas in FORTRAN. For the Honeywell machine, all global namesets are compiled with absolute origins so that they are uniform for all the overlays. The terminal user's typed input line is in the array KBLINBF in the nameset KBLINE. The scanner *nxttoken* is used for reading the line. It picks up the next available token in the array being scanned. The token may be an identifier, an integer or a real or an octal (O'n...n') number, a string enclosed in quotes, a single character used as a punctuation mark or an operator, or the end of text character. The variable COMPTR in the nameset COMINF indicates the character position in KBLINBF where the next token search will begin.

The variable COMFLG in the nameset COMINF is a state indicator of the command system. Before a subprocessor returns control to the system it sets COMFLG to indicate the status of processing. If processing is complete and no error was detected the value is 1. If one or more errors were found, COMFLG is set to 2. If processing is complete but the subprocessor wants to issue a specific error message, warning, or information message to the user, the indicator is set to 3. For a command which requires multiple lines of input, COMFLG takes on the values 4, 5, and 6 to indicate no errors found, an error detected, and a message issued. When COMFLG takes on one of the last three values, the new line of input will be processed by the same subprocessor.

A subprocessor may issue a message to the user by calling the routine *mvhresp* with a LITTLE character string as the argument. To issue cues to the user for a multiline input command the programmer writes

```
CALL H2CRT(TEXT,CUEOUT,LNCUE);
```

where TEXT is a LITTLE character string and CUEOUT and LNCUE are global variables in the nameset CUEHOLD. COMPTR is set equal to 1 plus the string length of TEXT to indicate the first possible position in the input line where the user may enter information. The routine *h2crt* converts a LITTLE character string on the Honeywell computer to display characters for the Vector General. The system has programs for converting integer, real, and octal number tokens returned

by the scanner to binary formats. It also has programs for performing the inverse operations. All of these programs are available to the application programmer.

The implementation closely follows the high level SETL specification given for *commandprocedure* in Chapter 3. Each computer has a version of the *commandprocedure* and both invoke KBINHNDL (*kbinputhandler* in the SETL specification) as a first step in executing a resident command subprocessor. The two machines have different versions of KBINHNDL.

Initially when the system was small enough to fit entirely within the memory of the satellite both computers had the same KBINHNDL but as the system grew an overlay library had to be created for the satellite and a different mechanism for invoking command subprocessors adopted. In the remote computer, KBINHNDL uses the value in the ADDR field of the command table entry as an index to a list of invocation statements for the subprocessors. In the satellite version the ADDR field is used slightly differently.

In the satellite, all overlays start loading at the same memory location. Each one begins with a routine that, through macro definition, has the name LAYOVER. If an overlay has more than one command subprocessor, LAYOVER has a list of values of the ADDR fields from the command table entries for the subprocessors it can execute. When LAYOVER is called it is sent an ADDR value. The value's position in the LAYOVER list is the index to the subprocessor invocation by LAYOVER.

In the satellite KBINHNDL checks to see if the overlay currently resident in memory is the one which contains the subprocessor. If it is, then KBINHNDL calls LAYOVER. Otherwise the appropriate overlay is read in and then LAYOVER is called.

When a user decides to reconfigure the application software and uses the ASSIGN command, the subprocessor for ASSIGN first examines the A field in the command table entry for the command to be reassigned to determine if the change is permissible. If it is, then the M field is set to 0 if the subprocessor will execute on the satellite or to 1 if the subprocessor will execute on the remote machine.

When the *commandprocedure* in the satellite determines that a subprocessor executes in the remote machine, it checks to see if the communications overlay is in memory. If the overlay is not present then it is read in and LAYOVER is called. LAYOVER then calls MAINTRANS which formats the keyboard input as a card image for transmission to the remote computer. MAINTRANS makes sure the line is up between the two computers and then transmits a batch job for processing by the remote machine. The first three card images involve necessary accounting information. The fourth card is a call on a procedure file at the CDC 6600 which loads and executes the graphics processor. Two procedure files are available. One is used if the satellite expects data to be returned by the remote machine and the other is used if no

data is to be returned. The fifth card image is an END OF RECORD; the sixth is the formatted input line, and the seventh is an END OF FILE. If a response is expected, a message is issued to the user to 'CHECK FOR OUTPUT FROM 6600'. If a response is expected a semaphore is bumped.

To retrieve the data, the user enters

CDC PLT,XXXX

where XXXX is a system supplied user hash code. When the information has been received, the semaphore is decreased and the block is interpreted. If the block is a display list the routine PROC66 issues a message to the user saying so. The user may then type 'add' which will put the display on the screen. If the block is a user message from the remote machine, the message will automatically appear on the screen.

When the remote machine receives a request for processing from the satellite, the card image is input as a LITTLE character string in BCD format. The string is converted to the Vector General ASCII format and placed in KBLINBF so that the same programs which process it on the satellite can process it in the remote machine.

Professor Thomas Stuart was responsible for the software for overlay input, intercomputer communications, and display list storage on and retrieval from the disk. Mr. Peter MacLean was responsible for the low level assembly language disk package which supported the overlay and display list software.

Chapter 5

The Application

The distributed application which was implemented was a simple economics model of a rent distribution function for a city with a central business district (Solow, 1973). The reasons for choosing this particular application were that (1) it was easy to implement quickly; (2) it could serve as a prototype for a more complex economic model; (3) other features of the graphics system such as real-time rotation could be used with it; and (4) its calculations were simple enough so that the intercomputer communications would be the principal system feature tested.

The goal of the model is to maximize, relative to a constraint, a utility function which is an attempt at quantifying the quality of life for a family in the model of the city. The constraint is that the wage less a worker's transportation costs is equal to the sum he pays for consumer goods and housing. One good is produced in the city at a given price and all workers have the same wage. One wants to find the distance from the center of the city beyond which the cost of transporting the good to the city center consumes too much of its price. The distance is also the closest point at which a worker can live without consuming too much of his wage and thus decreasing the quality of his life.

In the application program, the user can plot the

rent function

$$r(x) = c \cdot \gamma^{1/\beta} (W - T(x))^{(\alpha+\beta)/\beta}$$

where

c = a normalization constant

β = elasticity of the utility function with respect to land as a good; it measures the degree of responsiveness of the utility to a percentage change in the amount of land.

α = elasticity of the utility function with respect to the produced product as a good.

W = monthly wage of a worker

$T(x)$ = a worker's total transportation cost between the job and home

$$\gamma = \frac{\alpha^{\alpha}\beta^{\beta}}{(\alpha+\beta)^{\alpha+\beta}}$$

In the program we take $T(x) = Tx^e$ for the transportation function. As parameters the user may vary α, β, W, T, and e. The program provides a quick way of studying the behavior of the function r(x) as the parameters vary. If α and β are chosen such that $\alpha+\beta = 1$ then the utility function is a Cobb-Douglass function.

The basic operations the user can perform are (1) to plot the rent function, display an axis, and output the current parameter list; (2) to display only the rent function; (3) to display several copies of the curve around the rent-axis to produce a surface; and (4) to draw an axis and list

the parameters only.

The first alternative is useful as a previewing method of curve behavior before more calculations are made or a hard copy plot is produced. The second choice allows a user to quickly display several curves produced by different parameter choices in the same area for easy comparison. The third possibility presents a visual idea of what total rents in the city are at particular radii. The fourth alternative allows the user to superimpose an axis on choices two and three as a means of reference.

Once the user has the pictures, he may operate on them as he wishes. He can translate, rotate, or scale them with the surface picture, a real-time rotation will emphasize the three dimensional aspects of the picture. The pictures can also be saved on disk.

The problem of whether the application shall execute on the satellite or the remote computer is a function of the work load of the remote computer and the type of work the user wants to perform. The satellite can do all the calculations involved in the application and can present the results to the user in less time than the user can retrieve the results from the remote machine. The average time for the satellite to calculate a function and display it with an axis is 4.2 seconds. The time between the entering of a command and the display of only the curve is about 2.6 seconds. The time needed to produce a surface

is about 2.9 seconds. The remote machine can of course do the calculations much more quickly: .086, .067, and .074 seconds respectively. But among the problems with using the remote machine are the transmission times to send the request for service and to retrieve the results; the fact that the current communications software treats a request as a batch job which means that it must spend time in an input queue before being serviced and that it may be rolled out while executing; and the current communications software requires that the user enter another command to retrieve the results from the remote machine.

The communications software has the above drawbacks because it was much easier and faster to modify existing remote job entry terminal software than to produce a new communications package at this time. The user receives a message from the system that he must enter a retrieval command to obtain the results. This is a relatively minor inconvenience. The major problem occurs when the remote machine has a moderate to heavy work load and the request for service is not processed immediately. While waiting, the user can do other work. In a situation in which the remote machine has long delays in servicing, the user can still effectively use it if he has many different plots he wants to make by entering the commands one after the other. Before he has entered the last request, the first set of results should be ready for retrieval.

In the event that the user has only a few examples to work out, he will get much quicker results if he uses the satellite for the calculations.

The particular model used here is fairly simple but it illustrates the potential usefulness of a distributed graphics system for economic model analysis. More parameters might be added to make the model more complex. Changes in the various parameters would reflect changes in social policies and the changes in the rent pattern could be studied.

Chapter 6

The Data Structure

In designing a graphics system, one has to determine fairly precisely what the requirements of the system are. Also useful is the ability to anticipate what future requirements may be in order to allow for them in the design. One advantage of using a very high level specification language for design is that implementation details can wait. However one cannot ignore the decisions concerning the basic information to be found in the data structure.

If the sole use of a system is by a particular application, then the data structure will be very application oriented and can be part of the application data base. This type of structure, however, is generally not suitable for use by other applications. If the graphics system is to be for general use then the data structure must be independent of any particular application. Interfacing routines between the application and the graphics system will hide this independence from the user.

In the implementation of the data structure, several other factors need consideration. A specific structure topology has to be chosen. The SETL programs which we will present to describe the data structure manipulation algorithms use sets but for an implementation the data relationships

have to be precisely defined. The data base may consist of bits, rings, trees, hierarchical structures, or a hybrid combination of two or more of these.

Another factor is the language of implementation. As already mentioned, one of the goals in designing a graphics system and its data structure is to allow enough flexibility in the initial design so that satisfaction of future requirements is not difficult. To use assembly language would make this goal impossible. A major advantage in using the system implementation language LITTLE for our intelligent satellite graphics system is that a routine that may run on either the satellite minicomputer or the remote host computer has to be coded only once. Another advantage is the portability of the graphics system software to another computer system with a LITTLE compiler. Languages such as LSD, BLISS (Bergeron et al., 1972), SYDEL (Garwick, 1972), PL516 (Wichman, 1972), SPL (Klunder, 1972), and XPL (Horning, 1972) permit the embedding of assembly language instructions which can prevent eventual software portability.

Also important in data structure design is the hardware available. A bulky data structure in a small minicomputer will be a disaster because it will consume valuable memory that can store useful processing programs. Even in a virtual memory system, the time needed to page programs and data back and forth between core and auxiliary storage degrades

the performance of the display processing routines. Our hardware configuration consists of a Honeywell 316 computer with 32768 16 bit words, a teletypewriter, a disk, and a link to a remote CDC 6600. The graphics display hardware consists of a cathode ray tube, a keyboard, and a display controller with hardware registers to perform subpicture scaling, translation, and three dimensional rotations.

Finally, based on the original requirements of the system, we have to determine what data has to appear in the structure and the related lists, tables, and data bases so that it is reasonably easy to generate a display file. In an interactive system, we must be sure that data is easily and quickly retrievable from and storable into our data structure.

To make the system appropriate for more than one type of application, we must use a virtual graphics data structure that is both application and display processor independent. Interfacing routines permit the user to relate his data to the graphics objects described in the virtual structure. The independence of the virtual structure from the display processor means that from the data of the structure one can generate display commands for different graphics devices without reprogramming the whole system. To achieve this independence we consider our drawings or

pictures in terms of three basic primitive objects, lines, arcs, and text strings.

A fourth type of image that we can display is that of a "special object". A "special object" may be a point list, an edge list, a polygon list, or a subpicture which is never going to be modified and thus can be saved in its display file form only. The purpose of having special objects is to control the size of the VIRTUAL structure and the table of subpicture definitions, DEFTAB. If each subpicture had to have an entry in the VIRTUAL table for every line in its image, then complicated pictures would soon fill up computer memory. The special object category helps to restrict core usage.

The next decision is the choice of the topology of the structure. Many topologies are possible but the one chosen for the implementation is a tree-like hierarchical structure. For the purposes of the SETL algorithms, the structure is a tree (Figure 1). Each node represents a subpicture. Each subpicture is defined in terms of the nodes and leaves below it. A leaf has no nodes or other leaves below it and represents a primitive object such as a line, an arc, a text string, or a special object.

The basic reason for choosing the lower structure is that the header describing the subpicture relationships of each node and leaf will have a fixed format which will

save both calculation time for accessing information and compute time for moving data around when subpicture modification occurs and for doing garbage collection on holes. Concern over excessive bookkeeping calculations is justified if we want to assign to a satellite mini-computer with a limited instruction set and a slow core memory as many of the software tasks as possible in order to optimize the performance and response time of the graphics system.

Let us consider the data necessary for each element of the VIRTUAL structure. First if we examine the lower structure of Figure 1, we see that a node needs

1. a pointer to the node's "left" or "first" son,
2. a pointer to the node's parent, and
3. a pointer to the node's brother.

In order to define a picture in terms of its sub-pictures, the son field in the node that represents the picture points to a chain of nodes and leaves representing the subpictures.

In a leaf we replace the son pointer with a field that describes the leaf type. For a special object this field points to a table containing more information about the object.

In order to quickly propagate an indication of a modification made on a subpicture and thus on all the subpictures it helps define and whose nodes are higher

in the VIRTUAL structure, each node and leaf has a pointer to its parent.

The brother pointer is a link in the chain of subpictures which define the next higher level subpicture. This linking method makes memory management and structure handling easier through the use of fixed size node and leaf entries.

We have defined the mechanisms through which one element of the VIRTUAL structure is related to another element. We can specify now the data associated with this element and the subpicture which it defines.

For a node, the data is

1. its position or displacement in the main picture,
2. its angular orientation in the main picture, and
3. its scale.

In Figure 2, the illustration of the fields of a node for the SETL algorithms, "vcenter", "vrotmat", and "vscale" represent this data. The field "vcenter" is really a triple consisting of x, y, and z coordinates specifying the center of the subpicture. These coordinates are functions of the coordinates, angular orientation, and scale of the parent of this node and the original definition of this subpicture.

The field "vrotmat" is a nine element rotation matrix which defines the angular orientation of the subpictures that define this subpicture. The rotation matrix is a function

of the rotation matrix of the node's parent and its own original definition.

The field "vscale" is the scale of the subpictures that are descendants of this node. The scale is a function of the scale of the parent node and the scale in the node's original definition.

Since each subpicture in the VIRTUAL structure has its location, orientation, and scale specified in the coordinates of the main picture, we need a way of referring back to the original definition of the subpicture in its own coordinates. The "pictureid" field permits referral to the original definition of the subpicture in the DEFTAB. This definition can be considered the master instance (Newman and Sproull, 1973) of the subpicture in its own local coordinates (Figure 3).

The leaves of primitive objects are represented in the same type of element. The contents of some of the fields will be different or undefined. For a line, the data will be the coordinates of the two endpoints. For an arc, the start and stop angles and major and minor radii replace the rotation matrix and the scale. A text string will have information on character orientation (vertical or horizontal), character size, the number of characters in the string, the location of the string in the computer, and the location of the string in auxiliary storage. Figures 4, 5, and 6 illustrate these leaves.

The leaf type for a special object will point to an array which describes the object and points to more information about it. The rest of the information for this leaf will be like that of a node.

Chapter 7

Picture Development and Data Structure Manipulation

In this section, we will present some simple examples of picture development and modification and the SETL algorithms which describe the more complex data structure manipulations. The basic routines which operate on the data structure are

A.	*addprimitive*	adds primtiive graphics objects to the virtual structure
B.	*definitionsave*	creates a master instance of a picture from the virtual structure
C.	*picturebuild*	adds a specific instance of a previously defined subpicture to the virtual structure
D.	*modifypicture*	performs transformations on the virtual structure
E.	*virtualtographics*	maps the leaves of the virtual structure into graphics segments

7.1 Building a Picture from Primitive Objects

The top node of the VIRTUAL structure represents the picture on display and is defined in terms of the nodes and leaves on the second level of the structure. If the screen

is blank, the top node will have no descendants. The addition of a new object or subpicture to the display should affect only the definition of the top node and therefore the representation of the new addition as a node or a leaf should be added only at the second level of the structure and not elsewhere.

Suppose we start with a blank screen and wish to build up a picture from primitive elements. The subprogram which makes the corresponding additions to the VIRTUAL structure is *addprimitive*. For lines and text the routine creates leaves which define the objects. For an arc it adds a node with position, angular orientation, and scale information; and a leaf entry specifying start and stop angles and minor and major axis sizes. It also adds each leaf to a list to facilitate the generation of graphics segments by *virtualtographics*. The SETL specification of this program is simple and appears in Appendix B.

If we entered at the keyboard the command

```
SHOW LINE  -.5, .5,0  .5,.5,0
```

a line with end points (-.5,.5,0) and (.5,.5,0) would appear on the screen and a leaf would be added to the VIRTUAL structure.

Similarly, the commands

```
SHOW LINE  .5 .5 : .5 -.5
SHOW LINE  .5 -.5  0  -.5  -.5   and
SHOW LINE  -.5  -.5 : -.5 .5 0
```

will add the lines [(.5,.5,0), (.5,-.5,0)],[(.5,-.5,0), (-.5,-.5,0)], and [(-.5,-.5,0), (-.5,.5,0)] to the screen to produce a square. At the same time, the VIRTUAL structure gets three additional entries.

To inscribe a circle inside the square, we use the command

```
SHOW CIRCLE 0,0  RADIUS  .5
```

which places a circle at (0,0,0) with radius of .5. The VIRTUAL structure gets an entry describing the circle.

In order to add a character string to the display, we first type

```
STRING -.1,0  H  3
```

which says that beginning at (-.1,0,0) we want a horizontal string of character size 3. The system will then respond by providing a blank line for entering the text. We can type

```
HOLE
```

and it will appear on the screen and an entry will be added to the VIRTUAL structure.

Each of the entries in the VIRTUAL structure is a new item so its "pictureid" field is set to zero to indicate that the item is not defined in the definition table, DEFTAB. The rest of the node or leaf has the appropriate information for the subpicture it represents. Figure 7 illustrates the effects the above sequence of commands has on the display and the VIRTUAL structure. Figure 8 illustrates the data in

each of the entries. The nodes and leaves are numbered in the order in which they are created.

7.2 Naming the Picture and Saving It

If we decide we may want to refer to the picture we just created at a later time without reconstructing it step by step, we can name the picture and save a master instance of it in the definition table, DEFTAB. The procedure for saving a picture by transforming its VIRTUAL structure into a master instance is *definitionsave*. This procedure takes the absolute values which are in the fields of the VIRTUAL structure and for each item in the structure converts them to values relative to those of the item's "parent". This method allows the same subset of nodes and leaves to define several instances of a subpicture. In order to understand the procedure for picture definition, we must define three tables.

The first table is DEFTAB. An entry here contains data which corresponds to that in either a node or a leaf in the VIRTUAL structure. In DEFTAB however, information is in terms of local coordinates. The examples in Figures 9, 10, 11, and 12 indicate the elements are similar to those of the VIRTUAL structure. The first field in a DEFTAB entry for a node points to a chain in DEFTAB which defines the subpicture. A primitive element has its leaf type in this field. The last field links an element to a brother which contributes to the definition of the next higher level subpicture.

The second table is a hash table for hashing a picture's name, and the third table holds the picture names, pointers to definitions in DEFTAB and links for picture name hash collisions.

The definition of a VIRTUAL node is saved by making entries for its sons in DEFTAB, making an entry for it, linking the sons together through their DEFLINK fields and then setting its entry to point to this definition chain.

The command

SAVE PICT1

will save the picture under the name 'PICT1'. If we assume that the picture we have constructed is the first one we want to save then the definition will be formed as illustrated in Figure 13. The SETL specification of *definitionsave* appears in Appendix C.

When we wish to refer to a subpicture and retrieve the master instance to build a picture, we go through a pointer chasing sequence as illustrated by Figure 14.

7.3 Adding Previously Defined Subpictures to the Structure

Suppose we decided to add another copy of the picture to the display at a different position with a different scale. We may type the command

SHOW PICT1 .25 .25 SCALE .75

which will cause the creation of a node in the VIRTUAL

structure. In the node, the position, the angular orientation, and the scale of the subpicture are functions of the values input with the command and the values in the DEFTAB entry referred to by the table containing the picture's name. The "parent" of the node is the highest node in the VIRTUAL structure. Next the linked list of entries in DEFTAB which defines the subpicture is used to build nodes and leaves on the next lower level in the structure. Each node is defined in terms of leaves and lower level nodes. The structure of the subpicture is complete when leaves are the only free elements without descendants in the VIRTUAL structure. Figure 15 illustrates the addition of a subpicture to the VIRTUAL structure. As each leaf is generated it is added to a list for easy reference by *virtualtographics*.

The program whose purpose is to add a specific instance of a previously defined subpicture to the VIRTUAL structure is *picturebuild*. Its SETL specification follows. (Note: in a flow block, a single statement may replace a block label.)

```
define picturebuild;                    /* a routine to add a previously defined subpic-
                                           ture to the virtual graphics structure */
                                        /* check to see if the   picture has been defined */
     picsearch(ppic,eflg);              /* if the picture is defined, ppic has a positive
                                           value and eflg is zero */
     if elfg ne 0                       /* picture undefined */
        then mvhresp('undefined picture');  /* message to user */
           return;
     end if eflg;
                                        /* initialize the orientation parameters of the new
                                                                                  picture */
     xc = deltax;
     yc = deltay;                       /* displacement of the picture */
     zc = deltaz;
     alpha = deltaalpha;
     beta  = deltabeta;                 /* angles of rotation */
     gamma = deltagamma;
     tau   = deltatau;
```

```
        scale = deltascale;                    /* scale of subpicture */
        vptr = max virtual + 1;
        rotation(top)=rotmap(alpha,beta,gamma,tau);  /* the rotation matrix for the
                                                   given angles is calculated by rotmap*/
componentid = vptr;                            /* componentid is a global variable whose
                                                   value the user associates with the subpicture
                                                   he is adding */
                                               /* now build the virtual structure by referenc-
                                                   ing the picture's definition in the deftab
                                                   set and using the recursive routine
                                                   buildvirtual */
buildvirtual(idptr(ppic),top);                 /* idptr(ppic) is the singleton subset of deftab
                                                   which contains the first level of definition
                                                   of the subpicture */
rotation = nl;                                 /* release the space required for the rotation
                                                                          matrices */
return;
end picturebuild;
```

```
define buildvirtual(definition,vnode);  /* definition is a subset of entries in deftab
                                           which provide the 1st level of definition of vnode*/
(∀defptr ∈ definition)
flow                                                isprimitive?
                    vnodeinit+                                          makenodeentry+
              whichprimitive? 4                          (buildvirtual(idpt(defptr),vnextnode);),
        pline, parc,pmess,  pspec;
whichprimitive:  subflow (linelt, arcelt, messelt, specelt)
                                                    isline?
                              linelt,                       isarc?
                                                  arcelt,     ismess?
                                                        messelt,  specelt;

isprimitive := type idpt(defptr) ne set
isline      := idpt(defptr) eq lline
isarc       := idpt(defptr) eq arc
ismess      := idpt(defptr) eq message
makenodeentry: vnextnode = max virtual + 1;
     virtual = virtual with vnextnode;
     if vnode eq top                          /* is the picture at 2nd level of structure */
        then pictureid(vnextnode)             /* indicate instance of a previously defined */
                = <nl, defptr>;                                          /* picture */
        else pictureid(vnextnode)
                = defptr;                     /* reference to definition */
     end if;
     son(vnextnode) = nl;
```

```
     parent(vnextnode) = vnode:
     son(vnode) = son(vnode) with                  /* calculate the vcenter(vnextnode) from */
                       vnextnode;                  /* vcenter(vnode); local center, dcenter(defptr),
                                                      in the deftab entry; the rotation matrix,
                                                      rotation(vnode); and vscale(vnode) */
tcenter = trot(rotation(vnode),
                       dcenter(defptr));           /* trot is a matrix-vector multiplication function*/
vcenter(vnextnode) = vcenter(vnode) tadd
     (tcenter vescal vscale(vnode));
vrotmat(vnextnode) = matmul(rotation(vnode), /* calculate the rotation matrix for this */
     drotmat(defptr));                                                    /* entry */
rotation(vnextnode) = vrotmat(vnextnode);     /* matmul is a matrix multiplication function*/
vscale(vnextnode) =dscale(defptr)*vscale(vnode);
                                              /* initialize a leaf entry in the virtual
     vnodeinit:                                                           structure*/
       vleaf = max virtual + 1;
       virtual = virtual with vleaf;
       pictureid(vleaf) = defptr;
       parent(vleaf) = vnode;
       leaftype(vleaf) = idpt(defptr);
     pline:                                   /* calculate the end points */
       tend = trot(vrotmat(vnode),
                dendl(defptr));
       vendl(vleaf)=vcenter(vnode) tadd
        (tend vescal  vscale(vnode));
```

```
    tend = trot(vrotmat(vnode),
           dend2(defptr));
    vend2(vleaf) = vcenter(vnode)
      tadd(tend vescal vscale(vnode));
    tlist = tlist + <vleaf>;          /* add this leaf to the list of leaves of the
                                         structure for quick reference */
parc:
    vcenter(vleaf)=vcenter(vnode);
    vstart(vleaf) = dstart(defptr);   /* start angle of arc */
    vstop(vleaf)  = dstop(defptr);    /* stop angle of arc */
    vminor(vleaf) = dminor(defptr);   /* minor radius */
    vmajor(vleaf) = dmajor(defptr);   /* major radius */
    tlist = tlist + <vleaf>;
pmess:                                /* calculate starting position of text */
    tstart = trot(vrotmat(vnode),
      dcenter(defptr));
    vcenter(vleaf) = vcenter(vnode)
      tadd(tstart vescal vscale(vnode));
    vtext(vleaf) = dtext(vleaf);      /* copy text */
    tlist = tlist+<vleaf>;            /* add to leaf list */
pspec:                                /* this leaf represents some type of optimized
                                         display file. first calculate the center
                                         coordinates */
```

```
        tcenter = trot(rotation(vnode),
                  dcenter(defptr));
        vcenter(vleaf)=vcenter(vnode) tadd
         (tcenter vescal vscale(vnode));
                                        /* calculate the rotation matrix for the display
        vrotmat(vleaf) = matmul(rotation(vnode),                              file */
                 drotmat(defptr));
        vscale(vleaf) = dscale(defptr)*
                        vscale(vnode);
        tlist = tlist + <vleaf>;        /* add to leaf list */
        if vnode eq top then
           pictureid(vleaf)=nl;;
                          end flow;
end Vdefptr;
return;
end buildvirtual;

definef a vescal b;                     /* this function multiplies the vector a by the
  if #a eq 1                                                          scalar b */
     then return <hd a * b>;
     else return <hd a * b, tl a vescal b>;
  end if;
end vescal;
```

7.4 Modification of the Picture

Suppose that we have defined a picture and established its data structure (Figure 16a). If we wanted to transform the subpicture represented by node 5 and its leaves 6 and 7 by a rotation, translation or scaling then we have to modify the VIRTUAL structure to reflect the change. The parameters in the node and leaves have to be recomputed but also the relationships among nodes has changed. The subpicture described by node 3 no longer includes node 5 as part of its description. The "pictureid" field of node 3 is modified to indicate it no longer represents a previously defined picture but is now a new subpicture. Node 1 is now defined in terms of leaves 2 and 10 and nodes 3 and 5. Leaves 6 and 7 have not changed relative to node 5 so its definition remains the same. The new structure appears in Figure 16b. The program *modifypicture* performs the modifications on the VIRTUAL structure that reflect the user's desired changes to the picture. The SETL specification for the program is in Appendix D.

7.5 Displaying the Picture

Once we have a VIRTUAL structure, we can generate display segments from its leaves. From the way we build the structure, the leaves contain or point to all the information necessary to build a display list containing

the actual draw, positioning, and transformation instructions necessary to generate an image on the display. As an example, consider a leaf that represents a previously defined picture. The leaf has all the rotation, displacement, and scaling information necessary for displaying the instance of the picture.

To speed up access to the information in the leaves, we maintain a list of pointers to them as we build the VIRTUAL structure. This list saves us the bother of traversing the structure again to find the leaves.

A segmented display list whose structure is parallel to the leaves makes reflection of a change in the VIRTUAL structure easy and avoids recomputation of the whole list.

The purpose of *virtualtographics* is to convert the VIRTUAL structure into a segmented display file. It uses the data in the leaves to produce graphics transformation and display segments.

virtualtographics produces five basic types of segments. The first type is a transformation segment. This segment contains information on the coordinate scale, the displacement, and the rotation of any subpictures produced by display segments which follow these transformation values. Only another transformation segment will change the transformation applied to succeeding display segments.

The second type of segment is a display segment which produces a single line between two points. The last

transformation segment preceding this display segment must be an identity transformation segment, that is, a segment with the coordinate scale set to full, the displacement set to zero, and the rotation matrix set to the identity.

The third type of segment is a display segment which produces an arc by approximating it with short straight lines. A transformation segment with the arc's scale, center, and rotation precedes the display segment.

The fourth type of segment is one which displays text characters. Like the line segment, the text segment must follow an identity transformation segment.

The final type of segment is the special object segment which has four subtypes. The first three subtypes constructed from point lists, edge lists, and polygon lists, require a preceding transformation segment while the fourth subtype is an optimized display list consisting of a mixture of transformation and display segments. This fourth subtype is built when a user decides to save a constructed picture whose subparts are never going to be modified in the future. These segments are kept in auxiliary memory until needed.

The SETL specification of *virtualtographics* follows.

```
define virtualtographics;      /* after the virtual structure has had a
                                  subpicture added to it, this routine
                                  converts the data in the leaves to
                                  graphics display segments. if no graphics
                                  segment is added to the display list
                                  then action=0 is returned, otherwise
                                  action = 1 */
regsw = 1;                     /* this flag will force a transformation
                                  segment setting the coordinate scale
                                  register (CSR) and the rotation registers
                                  of the hardware controller to full scale
                                  and the identity matrix if a single line
                                  or text segment is  to be displayed after
                                  a previous segment has changed the
                                  registers */
if tlist eq nl
     then                      /* nothing to display */
     action=0;  return;
end if;
if tcnt eq 0                   /* is this a completely new set of segments*/
     then                      /* yes, initialize graphics tuple */
     gstr = <nl,nl>;
     gstr(1) = #gstr           /* first component is length, second will be
                                  name if tuple is saved later */
     gsptr = #gstr + 1;        /* pts to end+1 of current tuple */
end if;
```

```
(tcnt+1 ≤ ∀tindx ≤ #tlist)      /* the next three statements transform the
                                   leaves of virtual to display segments */
tnvptr = tlist(tindx);          /* get leaf in virtual structure */
seglist = seglist + gsptr;      /* add to the segment list a pointer to the
                                   beginning of the new segment */
flow                    whichleaf?   4
      checkorient+   garc,        checkorient+   gspec,
      gline,                      gmess;
whichleaf: subflow(linelt,arcelt,messelt,specelt)
                              isline?
           linelt,                      isarc?
                                 arcelt,   ismess?
                                       messelt, specelt;
isline  :=  leaftype(tnvptr) eq lline
isarc   :=  leaftype(tnvptr) eq arc
ismess  :=  leaftype(tnvptr) eq message
checkorient:                    /* do we need a segment to reset the scale
                                   to full, the displacements to zero and
                                   the rotation matrix to the identity? */
    if regsw eq 1
        then                    /* yes */
        gstr=gstr+createtrans(identitytrans);  /* add a segment respesent-
                        ing the identity transformation to the string */
        regsw = 0;              /* reset switch */
        seglist(tindx)= (#gstr+1) is gsptr; /* correct segment list */
        gstr(1) = #gstr;
```

```
gline:                          /* create a segment to draw a line */

   gstr=gstr+createcommand(tindx,
        drawline,endpoints(tnvptr));

   gstr(1) = #gstr;

   gsptr = #gstr + 1;

garc:                           /* create a transformation matrix */

   k=parent(tnvptr);

   gstr=gstr+createtrans(vscale(k),
     vcenter(k),vrotmat(k));
   gsptr = #gstr+1;

   seglist(tindx) = gsptr;

   gstr=gstr+createcommand(tindx,
       drawarc,vstart(tnvptr),
       vstop(tnvptr),vminor(tnvptr),
       vmajor(tnvptr));
   gstr(1) = #gstr;  regsw=1;
   gsptr = #gstr + 1;

gmess:                          /*  generate text command segment */

   gstr=gstr+createcommand(tindx,text,
       vcenter(tnvptr),vtext(tnvptr));

   gstr(1) = #gstr;

   gsptr = #gstr+1;

gspec:                          /*  first determine what type of special
                                                  segment this is */
   specindex=message-leaftype(tnvptr); /*calculate  index of descriptor
                                                  of segment */
   if sptype(specindex)=sdfile /*  is this a special optimized file */

      then                      /*  yes */

      dfretrieve(specara,specindex); /* copy the segment from auxiliary
                                 memory and leave it in specbuf */
      gstr=gstr+specbuf;        /* add the segment to the list of segments*/

      gstr(1) = #gstr;

      gsptr = #gstr+1;
```

```
      else                          /* generate a transformation segment */

      gstr=gstr+createtrans(vscale(tnvptr),
           vcenter(tnvptr),
           vrotmat(tnvptr));
                                    /* copy  the segment next */

      dfretrieve(specara,specindex);

      gstr = gstr + specbuf;

      gstr(1) = #gstr+1;

      gsptr = #gstr + 1;

   end if;

   regsw = 1;

                              end flow;

end ∀tindx;
return;
end virtualtographics;
```

7.6 Multiple VIRTUAL Structures for the Same Picture

Earlier we made a drawing of a square enclosing a circle and a text string by using the primitive geometric objects of the system. Figures 7 and 8 illustrate the properties of the VIRTUAL structure for the commands which built this picture.

If we had decided beforehand to define a picture called "square" as an unalterable special object, we could have constructed an exactly similar appearing picture with the commands

```
SHOW SQUARE  SCALE  1.
SHOW CIRCLE RADIUS   .5
STRING    -.1, 0   H   3
HOLE
```

which would have produced the VIRTUAL structure illustrated in Figure 17. The leftmost son of the top node is a special object leaf which represents the square. Next comes the node and leaf containing information about the circle, and finally comes the leaf for the text string "HOLD".

Other ways of producing different VIRTUAL structures representing the same picture exist.

Chapter 8

The VIRTUAL Structure and Other Data Structures

In this section we will discuss the advantages of the VIRTUAL structure and a possible extension of it for hidden line computations. We will also briefly survey previous data structure design efforts.

8.1 Advantages of the Data Structure

The VIRTUAL graphics structure has several basic advantages. The first good point is that it is relatively compact which is important when a minicomputer uses it. The structure contains only the information necessary to generate a display and has a hierarchy to allow the user to refer to subpictures of the display. It is not bogged down with a huge core consuming ring structure and auxiliary tables make reference of subpictures easy and quick.

Modification of subpictures and changing the topology of the structure is very easy. Referring directly to a node or leaf makes reference of subpictures fast, and the modification of the VIRTUAL substructure representing the subpicture is straightforward. A pointer to any leaf that is modified in the VIRTUAL structure is added to a list of leaves which must have their corresponding display segments altered. A list of pointers to the display segments for the leaves makes quick modification of these segments possible.

A third advantage is that the VIRTUAL structure is easy to traverse. Each node and leaf has a pointer to its parent. All nodes and leaves on the same level of a subpicture structure are linked together through the brother pointer field. Each node has a pointer to a first son which is at the head of the brother chain. This system of pointers permits changes to a subpicture to propagate downwards in the structure and to be indicated in higher levels of the structure quickly.

A fourth point is that the VIRTUAL graphics structure can be used in other graphics systems also. It can be used to describe pictures for plasma screens, direct view storage tables, or cathode ray tubes. It serves the purpose of a pseudo display file (Newman and Sproull, 1973). A segmented or non-segmented display file can be generated from its leaves, or a series of subroutine calls with the information in the leaves as arguments can be used to generate a display on a storage tube or plasma panel. The structure does not have to be traversed to produce a picture. In a bigger computer more information can be added to a node to make processing easier. The macro facility of LITTLE makes it easy to define more fields of information or to redefine old fields.

A fifth advantage is that all elements in the VIRTUAL data structure are the same size and therefore bookkeeping is simpler.

The only real disadvantage to this structure is the

searching that occasionally may be necessary along a brother chain. This occurs when a subpicture is being modified and its level in structure is changed to level two as a reflection that the system is dealing with a new picture. The brother chain of which the picture is currently a member must be searched to find the picture's immediate predecessor so that the brother pointer of the predecessor may be changed. In general this will not happen too often. Most changes occur at the second level of the structure where no structural modifications are required. Otherwise the search is usually a short one.

8.2 An Extension for Hidden Line Removal

Besides the information necessary to display a picture, more information can be added to the structure. Information that might be added in the remote machine could be data for doing hidden line calculations. This data could be the maximum and minimum x and y coordinates of a subpicture. These coordinates would give window boundaries around subpictures and can be calculated at the same time the VIRTUAL structure is built.

Hidden line algorithms can be divided into two general categories. One category includes algorithms like those of Roberts and Loutrel in which each object is compared to every other object. The second category includes algorithms like that of Warnock in which an object is compared to a

screen window and if the window contains no visible object it is discarded; otherwise the window is subdivided until something can be displayed. The design of the VIRTUAL structure suggests a modification of the Warnock algorithm.

If two subpictures at the same level in the data structure are surrounded by two non-intersecting windows, then we know that their subpictures do not have to be compared because the parents do not overlap. If an overlap occurs, the subpictures at the next level will have to be put into disjoint sets if possible. This process continues until all that is left is a collection of disjoint sets containing leaves which have to be compared for intersecting or hidden images. Only leaves in the same set have to be compared.

Thus the idea of the algorithm is to do object comparisons only in the screen areas where windows intersect. On any other part of the screen no conflict exists and neither does a need for any comparisons. The windows described here seem to be analogous to the instance rectangles used in two dimensional transformation systems (Newman, 1975).

After the disjoint sets of leaves are formed, a modified Warnock algorithm can then be applied to the objects represented by the leaves to generate a display file with the hidden lines removed.

```
define  hidden;  /* a routine to eliminate hidden lines by
                    means of a modified Warnock algorithm */
  leafconflicts = nℓ;  /* this is a global tuple whose
                    components represent the sets of those leaves
                    which need an application of the Warnock algo-
                    rithm to eliminate lines */
  conflictfinder({top});  /* find the conflicts in the
                    subpictures of the top node */
  maxorder(leafconflicts);  /* order the tuple components in
                    decreasing order of number of elements in each
                    component */
  maxsubsets(leafconflicts); /* remove from leaf conflicts
                    those components which are subsets of other
                    components */
  dispfile = <2,nℓ>;   /* initialize the display file */
    /* apply the modified Warnock algorithm to each set of
       leafconflicts and generate the appropriate display
       segment */
  (1 <= ∀i <= #leafconflicts) Warnock(leafconflicts(i));;
  return;
  end hidden;
```

```
define conflictfinder(conflictset); /* this routine determines
      the  conflicts among the subpictures of the elements
      in the conflictset, a high level set of pictures
      having hidden line problems */
 allpossibleconflicts = {descendant∈son[conflictset]}
                        + {leaf∈conflictset|leaftype(leaf)ne Ω};
                  /* build a set of all possible conflicts in
                     the substructure under the conflictset */
 if allpossibleconflicts  eq conflictset
      then /* this is the lowest level of resolution for
                          this routine */
      leafconflicts = leafconflicts + <conflictset>;
      return;
 end if;
 /* for each node  in allpossibleconflicts form a set of
    potential conflicts and initialize a set of actual
    conflicts to nℓ */
 (∀node ∈ allpossibleconflicts)
      potentialconflicts(node) = allpossibleconflicts less node;
      conflicts(node) = nℓ;
 end ∀node;
 /* now for each node in allpossibleconflicts, determine
    its actual conflicts from its set of potential conflicts */
```

```
  (∀node ∈ allpossibleconflicts)
     (∀v ∈ potentialconflicts(node))
        if node overlaps v /* overlaps is an infix function
                    which returns a value of true if the windows
                    of node and v intersect. If the windows are
                    disjoint then it returns a value of false */
           then
           conflicts(node) = conflicts(node) with v;
        end if;
                              /* remove a redundant check */
     potentialconflicts(v) = potentialconflicts(v) less node;
     end ∀v;
  conflictfinder(conflicts(node) with node);
end ∀node;
return;
end conflictfinder;
```

8.3 Other Data Structures

We will now examine some other possible data structures. We will discuss three of the main examples surveyed in (Williams, 1971) plus another structure which are comparable to the one we presented.

Graphics 2, a graphics system developed at Bell Labs. (Christensen and Pinson, 1967; Williams, 1971) has a hierarchical data structure consisting of node blocks, leaf blocks, and branch blocks. The node blocks represent subpictures; the leaf blocks hold graphical information; and the branch blocks contain picture transformation information. A ring connects all the branches into a node and another ring connects all the branches out of a node. The system uses a PDP-9 computer as a terminal connected to a large time-shared Honeywell 645 computer. The whole data structure is kept in the 645 while a version with fewer pointers is kept in the PDP-9 to save on storage. The storage saving has its price though because unless the user adds his own pointers and rings, searching the data structure for objects is very time consuming.

In the General Motors Graphics System (Joyce and Cianciola, 1967; Williams, 1971) the data structure consists of three sets of rings for data elements, entity blocks, and display buffer data. The rings of the data elements and the entities are bidirectional and there are two-way connec-

tions between a data element and its corresponding entity block and between an entity block and its display buffer data. A time-shared IBM 360/67 is connected to a regeneration buffer, IBM 2840-II. The display console, an IBM 2250-III, has some display controller logic. The entity blocks contain graphics display property information such as object intensity and light pen selectability. All the pointers make response time very good, but they also make the data structure very bulky and only suitable for a large machine. Also some information in the entity block seems to be too display dependent and perhaps should have been kept in the display data buffer.

In the Univac Graphics system (Cotton and Greatorex, 1968; Williams, 1971), a hierarchical data structure is used. Copies of this structure exist in both the large remote computer and the satellite computer which acts as the terminal. The main data base is called the Entity Table which has all the data within the system. A Directory in the form of a hash coded reference table allows rapid access to major structural entities in the table, and an external directory relates user supplied names to the internal codes in the Directory and Entity Table.

The basic structural element in the Entity Table is the *group*. A group is made up of *items* and *uses* of other groups. This organization allows for subroutining in the data structure. Because a group definition can be used

repeatedly to define other groups there is a danger of recursive definition which will not be caught until the structure is processed for transmission to the satellite.

An item contains the graphical information necessary for display. An item is a logical combination of points, lines, and text which are called components. An item cannot be used more than once but a single item group can be.

In the Entity Table, a group has a pointer to a chain of uses and items, pointers to associated data, and a pointer to the uses of this group. A use entry has a pointer to the next entry within the structure of its parent group, a pointer to the next use of the group, a pointer to the component ring (contains information like a positioning vector and transformation matrix) for this use, and a pointer to associated managerial data. An item entry has a pointer to a component ring having information about points, vectors, and text. The Entities also have attribute information about displayability, external names, and light pen detectability.

The satellite computer has three data structures. It has a copy of the Entity Table for the picture being displayed. It has a display file consisting of *item blocks* that are really graphic subroutines. Finally it has an ordered display list which contains positioning and light pen detectability information and the order in which the item blocks will be executed. The display decoding hardware

must execute a complete pass through the picture organization on each refresh cycle.

The drawbacks in the Univac system are the very large data elements for the satellite computer and the apparent need for display hardware that can process the item blocks as subroutines and execute jump instructions between blocks.

In the Pictorial ENCoding Language, PENCIL (Van Dam and Evans, 1967), the data structure is of a hierarchical tree form. Pictures consist of subpictures, points, and lines. In the structure an item contains linkages to all its subpictures and lines, a transformation for each subpicture, and a transformation matrix for the whole picture. Lines and points are the terminal nodes of the tree of a subpicture. A drawback to this system is that the picture tree must be walked each time the picture is output. For the implementation described in the paper, this is not a problem since output is to a printer, but it is a hindrance to a refresh device. Also, items in the structure are of variable size with many pointers, and so a well-thought out memory management scheme is necessary.

A reiteration of the advantages of the VIRTUAL structure is

(1) it is compact which is beneficial for use in a minicomputer;

(2) fixed format nodes make memory management easier;

(3) it is extensible in a computer with greater processing power;

(4) information in a node is machine independent;

(5) a separate leaf list eliminates structure traversal for picture display;

(6) the data in a node or leaf is absolute so it reduces the need for extra sophisticated display hardware to perform transformation concatenation and graphics subroutining on the fly;

(7) referencing a displayed subpicture is easily done by referring to its corresponding node in the structure.

Chapter 9

Results and Conclusions

This work has demonstrated that in a distributed graphics system the satellite computer can serve as the center of system control and do a significant amount of the nontrivial processing required by an application without recourse to the facilities of the remote computer. When interaction between computers occurs, it is initiated by the satellite unlike other distributed systems in which the remote machine is the master. In addition, the user through commands to the satellite computer easily can reconfigure the system during real-time and transfer applications processing capabilities between machines. Also we have showed that the satellite computer does not have to be a machine with an extensive instruction repertoire.

This was accomplished by beginning design work from a very high level. We were able to envision the basic processor blocks more easily which helped to keep intercomputer communications simple. The top-down design method strengthened the assumption that all software should be able to execute in the satellite and that the satellite would be the center of system control. A useful by-product of the design was a machine independent virtual graphics structure which could be used with other display hardware and in non-distributed systems.

The use of the systems implementation language LITTLE permitted the writing of a single set of programs capable of running on both computers, but perhaps more importantly in our own particular situation, it allowed the lowering of the level of "critical intelligence" defined by Van Dam. Through software, we "educated" or "trained" the primitive Honeywell machine so it could perform most of the same tasks as better endowed, in terms of hardware, minicomputers. Its level of intelligence is high enough now that through an additional, finite programming effort its level of intelligence can be further lifted so that almost anything that can be done on the remote CDC 6600 can be done on the Honeywell satellite. The disadvantage is that as the layers of software increase, the execution time for a task lengthens so that processing by the remote machine, the CDC 6600, may still be preferable.

We did a linear programming analysis of the implemented rent distribution function processor and it gave the obvious answer for machine location. It said the processor should be located on the machine that had the higher ratio of (commands processed per unit time/cost of processing one command). We found that although the CDC 6600 could process a command forty to fifty times faster than the Honeywell satellite, it was in general still better to do the calculations on the satellite in order to provide results to the user more quickly. The reason for this was that our approach to intercomputer

communications was inadequate and this increased the cost of processing and lowered the above ratio for the CDC 6600.

One problem with the communication software in the satellite was that it was originally written to make the Honeywell a remote job entry terminal for the CDC 6600. Originally it was poorly written and it became a morass as it was modified. Unfortunately, a lack of person power, funds, and time required that it be modified again for our purposes instead of thrown out and rewritten explicitly for our project.

A related problem was the excessive transmission time for sending data between the two computers. There is a 9600 bits/second line between the satellite and a similar machine serving as a front end for the CDC 6600. Unfortunately, the effective transfer rate is only one-third to one-half the line rate. A major bottleneck is the CDC 6600's having to send the data through the front end instead of directly to the satellite. Tests indicated that the average transmission time for returning a display file to the satellite was 4 seconds with a best time of 2.1 seconds and a worst time of 6 seconds. These times alone are of the same magnitude as those for doing all the calculations and displaying the results on the satellite.

A third significant problem was our having to submit requests for processing by the remote machine as batch jobs. This meant that we were running graphics at a lower priority

than a standard time sharing job! One of the consequences of running batch jobs is that they had to pass through an input queue before processing could begin. In one set of tests, the average stay in the input queue was 26.8 seconds. The range went from 0 seconds to 136 seconds. In a second series of tests we improved on this by changing the time and initial memory size parameters to give the requests higher priority. The average time was .48 seconds and the worst time was 3 seconds.

After escape from the input queue, the next problem was the amount of time between the beginning of processing and the point when the display file has been routed back to the satellite. Under very light CDC 6600 usage, the situation was barely tolerable. The average duration of processing was 11.8 seconds with a best time of 10 seconds and a worst time of 17 seconds. In the medium and heavy usage periods the results were ridiculous. Under medium 6600 usage, the average processing time was 51.9 seconds in a range of 12 to 126 seconds. Under heavy usage, the average processing time was 297 seconds in a range of 21 to 706 seconds. We examined these times some more and found that the major time loss occurred between the point when actual program execution began and when it ended. For CPU execution times of .06 to .09 seconds, the shortest time was 3 seconds and the longest was 697 seconds. These times indicate that under batch processing, operating system requests increase

processing times by many orders of magnitude over CPU execution times and that under heavy CDC 6600 usage, the short graphics jobs can be rolled out of memory almost indefinitely.

A further problem caused by the present communications software is that the user in addition to the original command requesting application processing by the remote machine must enter a command requesting the return of the resulting display list. Again, the reason for this is that the original communications software is so poorly written that the system needs to bring the package into memory as an overlay because of its size. It is too large to keep resident while the user is doing other processing, so a new request for display list retrieval is necessary to return the package to memory.

There are several possible future paths for upgrading the intercomputer communications in order that the graphics system be used to its full potential. If we were to obtain more memory for the Honeywell 316 we could keep the entire communications package resident all the time. A small background loop could check for data waiting at the remote computer and initiate its transmission so the user would be free of that responsibility. In practical terms though, providing the Honeywell with more memory is impossible. It would have to be replaced by a better computer. This step is not as drastic as it sounds because LITTLE compilers have been written for other machines such as the PDP-11 and

most of the software would be transferable.

An alternative that is also a part of almost any other change is to completely rewrite the communications software with the sole purpose of satisfying only distributed graphics needs. This would make the package smaller and more efficient. It would be part of the satellite system root and could work in the background.

A third approach is to reprogram communications from the front end to the CDC 6600 so that it could read and write files at the CDC 6600 directly in order to shorten transmission times.

Another consideration would be to produce new systems programs for the CDC 6600 which would give very high priority to graphics servicing.

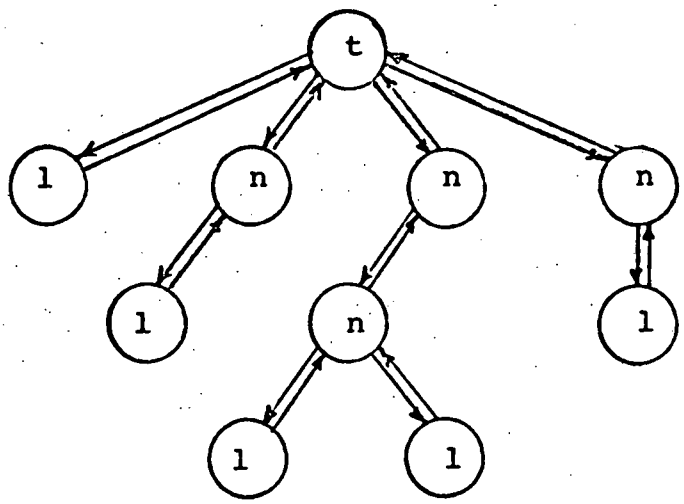
The most likely approach to produce the quickest results and improvements is to provide a new hardwired line between the satellite and the front end of the remote machine and handle graphics as part of the time-sharing system. A cooperating program in the remote machine would execute when a graphics request was received from the satellite and the results would immediately be returned.

One problem that did not arise with the particular application implemented is a shortage of buffer area in the satellite computer. Because the Honeywell 316 is such a slow processor, most system data concerning picture definition and creation is kept in buffers in memory so that it

is quickly retrievable. The rent application does not fill up these buffers because the pictures it generates are comparatively simple. A more complex application with more structured pictures could rapidly fill these data areas if the user was not careful. A more reliable disk system would be an aid in preventing this problem. The major part of the structure information for defined pictures could then be saved on disk and thus eliminate not only fear of buffer overflow but the buffers themselves.

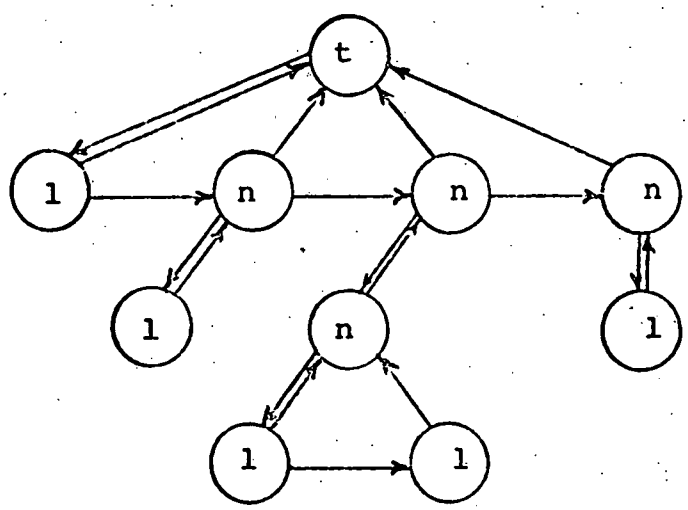
A program that did lengthy, complex calculations and generated a lot of data would be impractical to run on the satellite at any time because the system would lose its interactive characteristics. The computing power of the remote machine would be better suited for this type of problem. An example of such a program is one that generates many different views of a complex object. The user could run his view generator at the CDC 6600 and save each of the scenes on disk at the satellite as they were returned. A similar program might generate a sequence of movie frames. Again the major bottleneck for these types of applications would be intercomputer communications. Communications via the time-sharing system would not be feasible for such large amounts of data. A faster communications path such as a hard-wired channel to a peripheral processor would be necessary. Otherwise, for short streams of data the time-sharing system would not produce any appreciable delays through transmissions.

In conclusion, we have presented an approach to distributed graphics systems that produces an easily programmable and real-time reconfigurable intelligent satellite graphics system. Its high level design is clear and its actual processor interfaces and boundaries are well defined. The main drawback of the current implementation is the method of intercomputer communications due to a lack of sufficient personnel and funds to insure parallel and complete development of all system components. We have shown though that a functioning distributed graphics system can be configured from available, primitive components. A major investment in new computing power is unnecessary. A large software investment is needed but this is practical to provide because computing facilities already have software support staff members. Finally, an upgrading of the intercomputer communications will provide a system of very high potential.



SETL Version

t = top node
n = node
l = leaf



LITTLE Version

Figure 1. Virtual Structure.

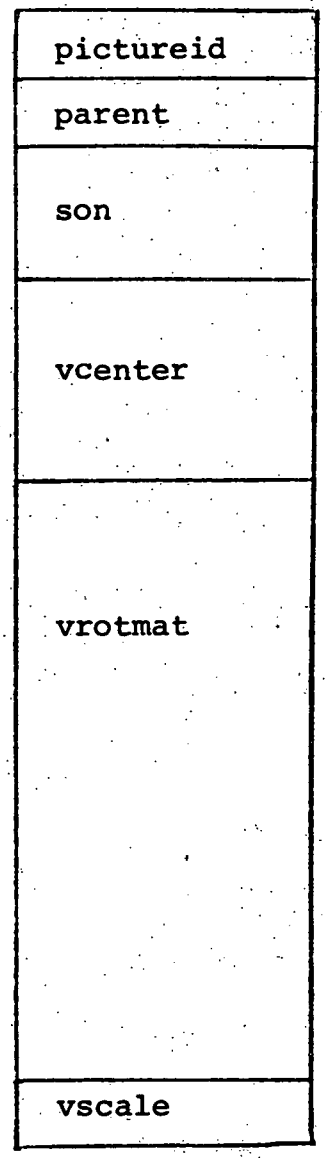


Figure 2. Fields of a Node.

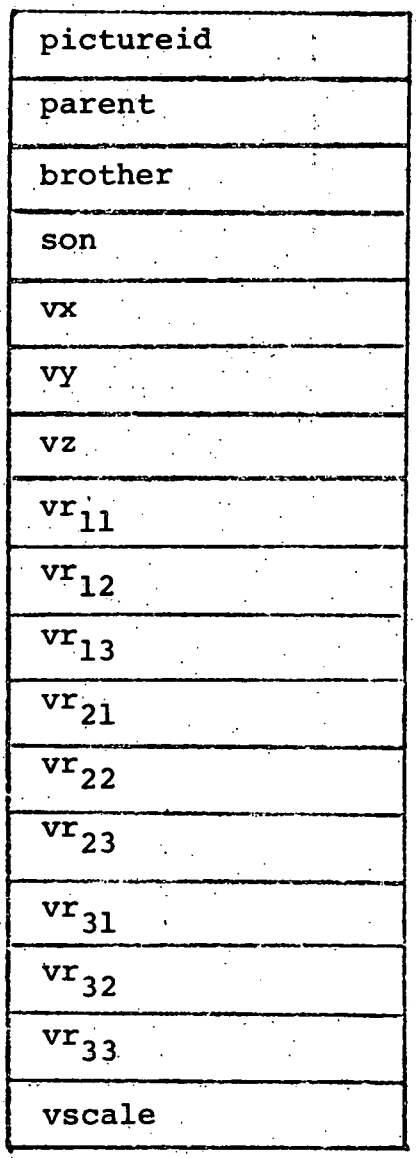


Figure 3. Virtual Node Entry.

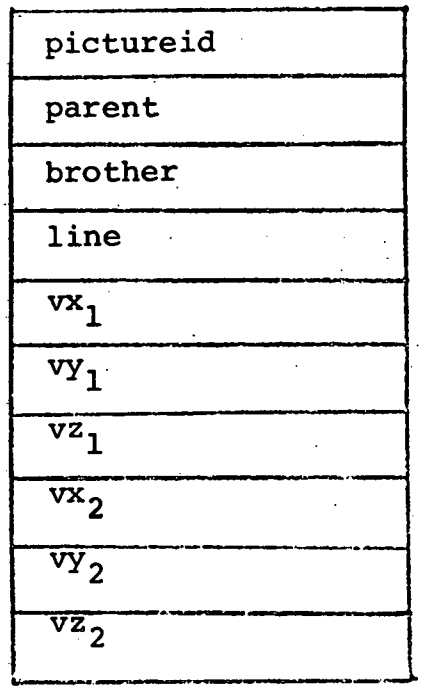


Figure 4. Line Leaf.

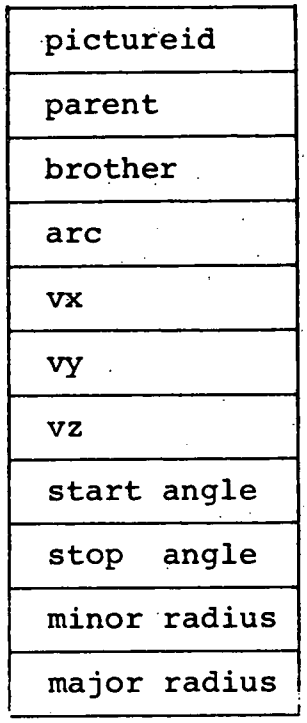


Figure 5. ARC LEAF

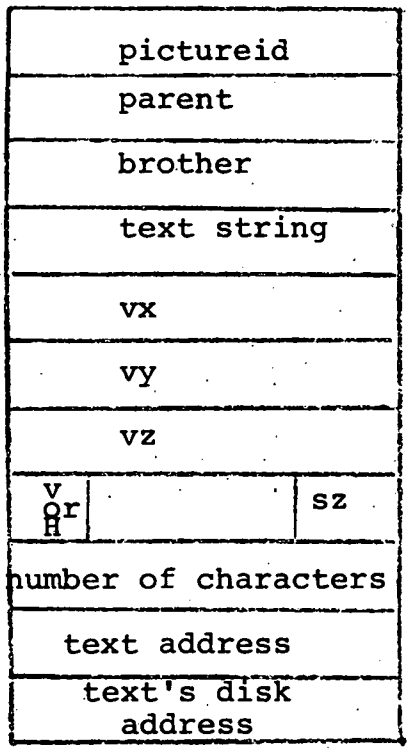
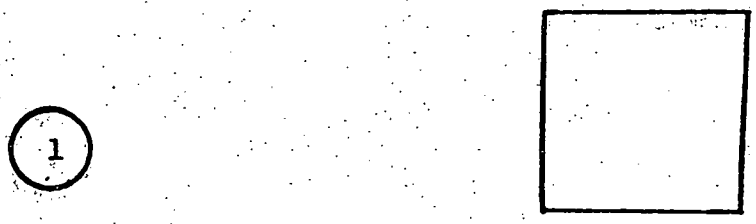
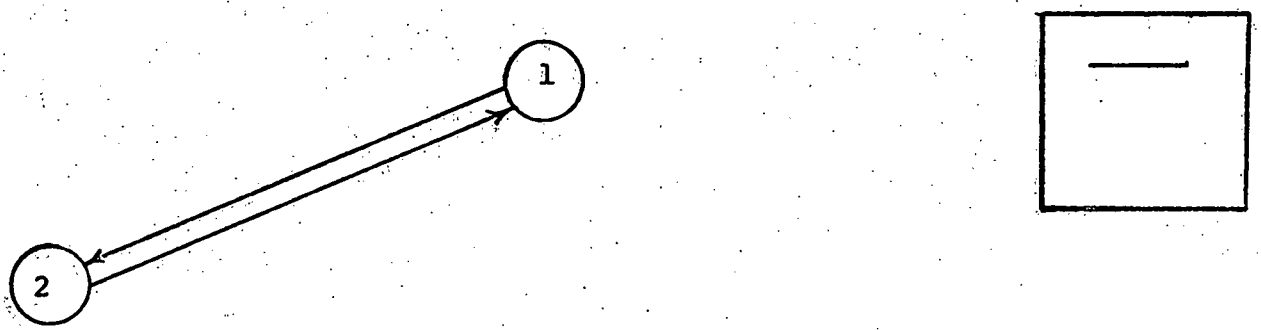


Figure 6. Text String Leaf.

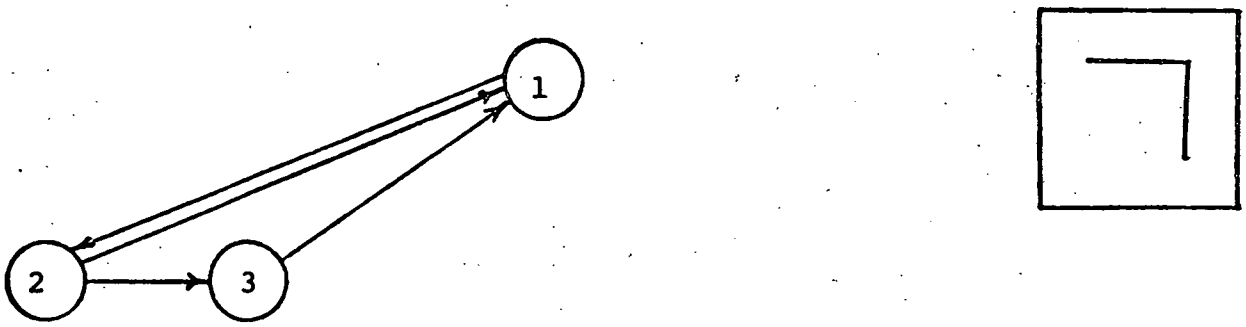


(a) Blank Screen.

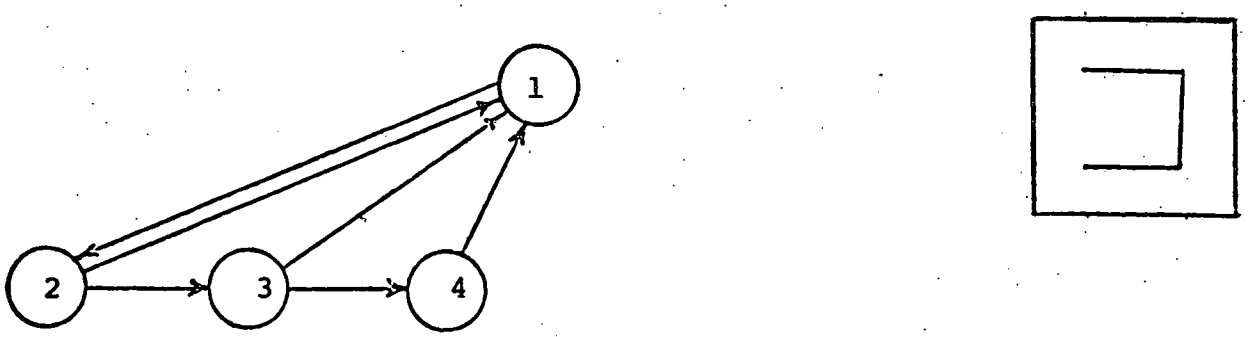


(b) Show Line -.5,.5,0: .5,.5,0

Figure 7. Building a Picture.

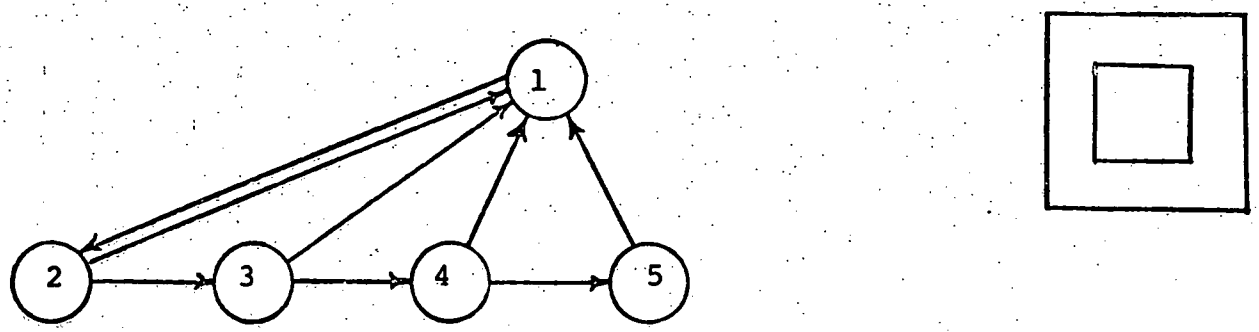


(c) Show Line .5,.5: .5, -.5

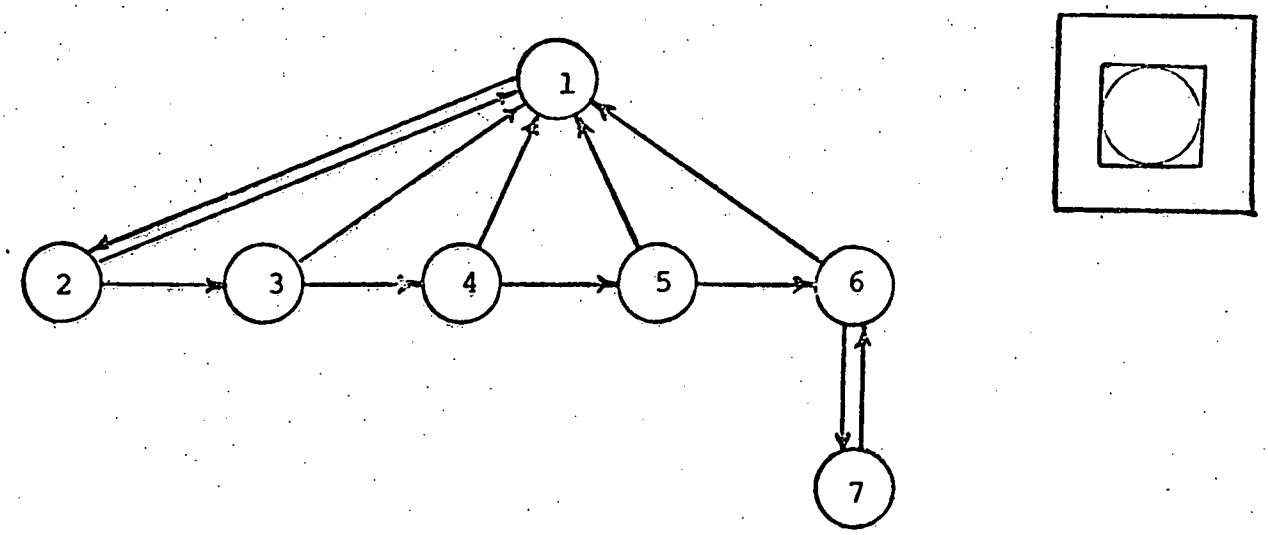


(d) Show Line .5,-.5,0: -.5,-.5

Figure 7 (Continued)

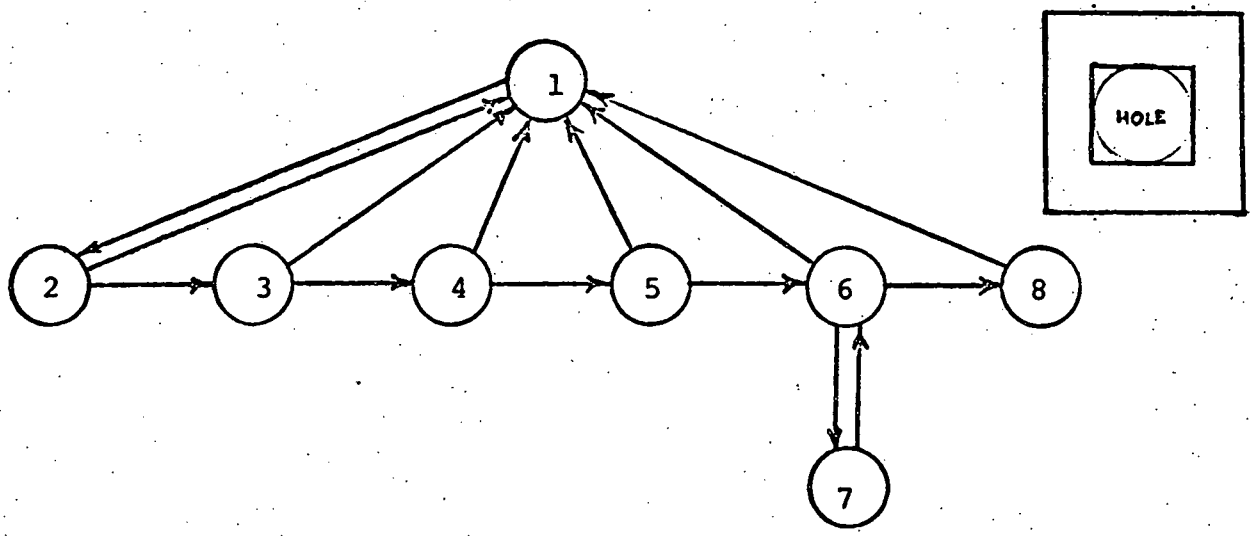


(e) Show Line -.5,-.5: -.5,.5,0



(f) Show Circle 0,0, Radius .5

Figure 7 (Continued)



(g) String -1, 0 H 3

HOLE

Figure 7 (Continued)

0
0
0
2
0.
0.
0.
1.
0.
0.
0.
1.
0.
0.
0.
1.
1.

Figure 8a. Node 1.

0
1
3
LINE
-.5
.5
0
.5
.5
0

Figure 8b. Line Leaf 2.

0
1
4
LINE
.5
.5
0
.5
-.5
0

Figure 8c. Line Leaf 3.

0
1
5
LINE
.5
-.5
0
-.5
-.5
0

Figure 8d. Line Leaf 4.

0
1
6
LINE
-.5
-.5
0.
-.5
.5
0.

Figure 8e. Line Leaf 5.

0
6
0
ARC
0.
0.
0.
0.
360.
.5
.5

Figure 8g. Arc Leaf 7.

0
1
8
7
0.
0.
0.
1.
0.
0.
0.
1.
0.
0.
0.
1.
1.

Figure 8f. Node 6.

0		
1		
0		
TEXT STRING		
-.1		
0.		
0.		
H		3
4		
text address		

Figure 8h. Text String Leaf 8.

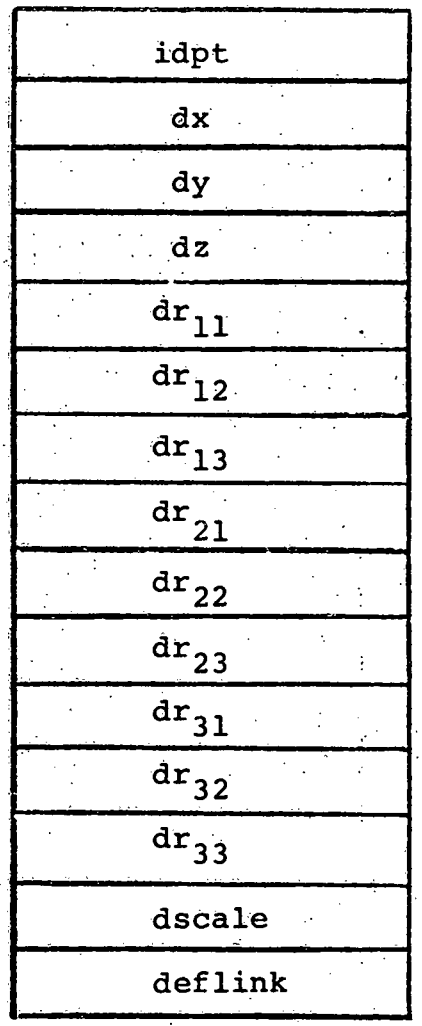


Figure 9. Node Entry in DEFTAB.

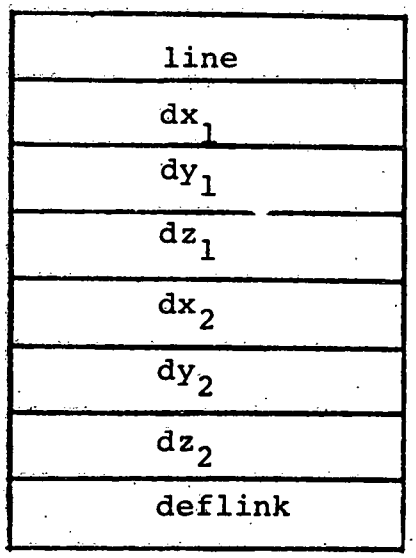


Figure 10. Line Entry in DEFTAB.

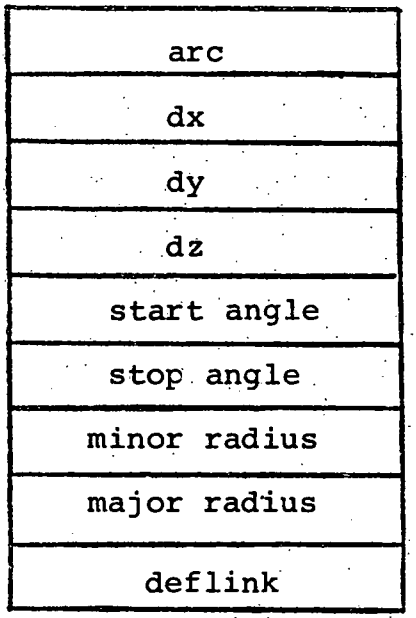


Figure 11. Arc Entry in DEFTAB.

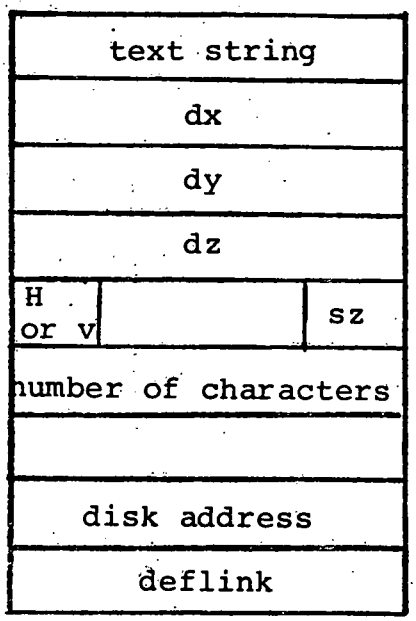


Figure 12. Text Entry in DEFTAB.

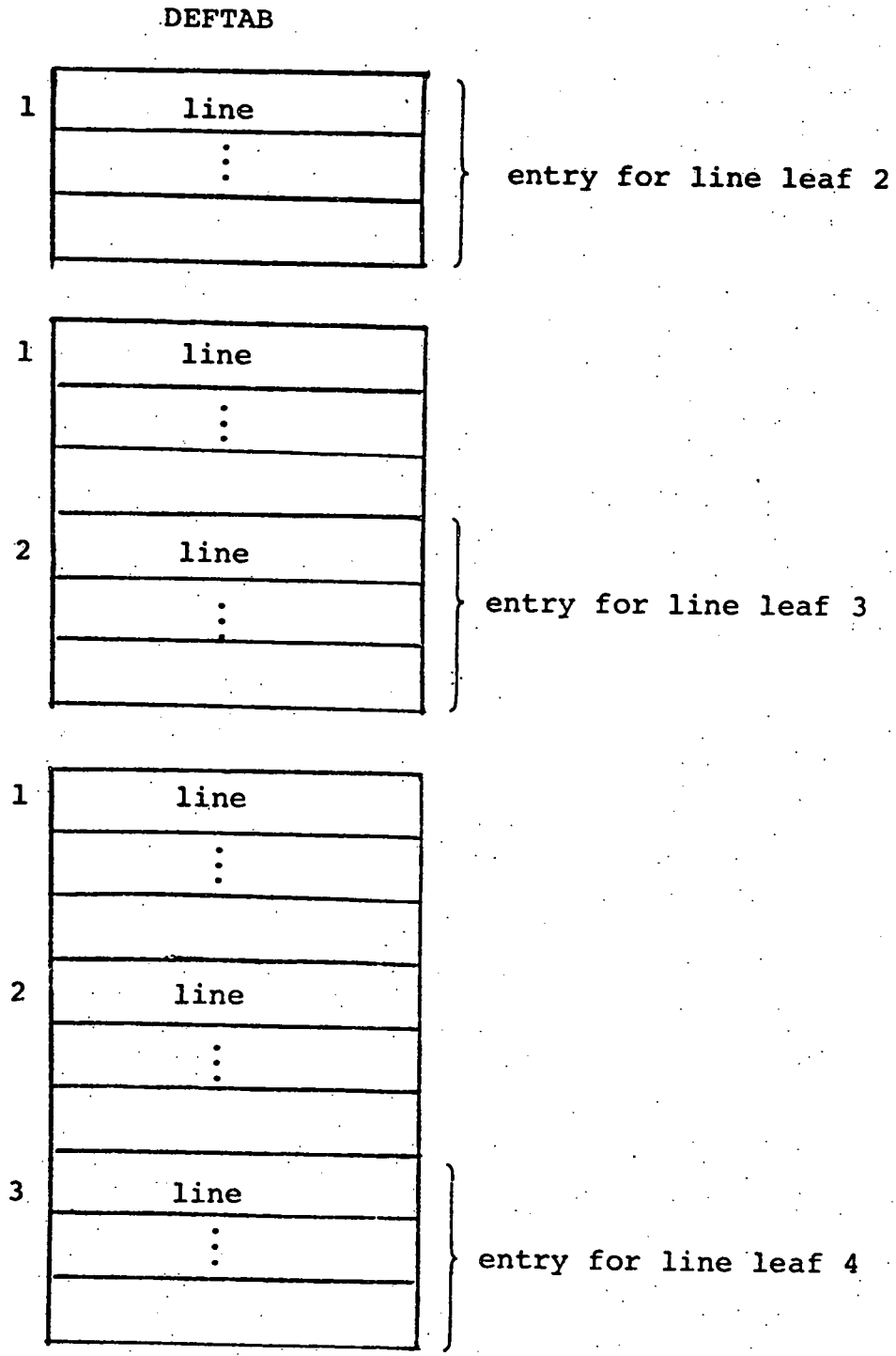


Figure 13. Steps for Saving a Picture in DEFTAB.

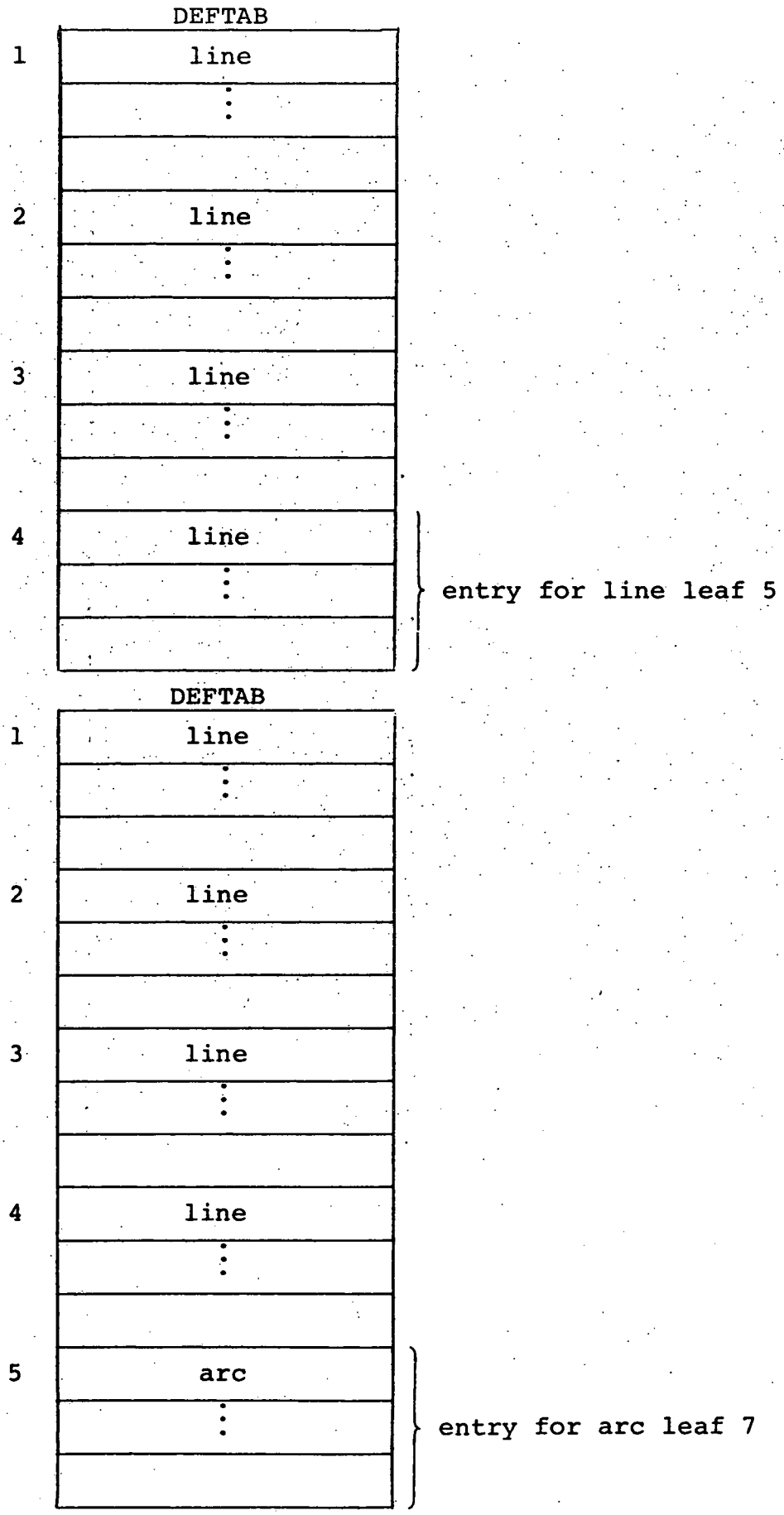


Figure 13 (Continued)

DEFTAB

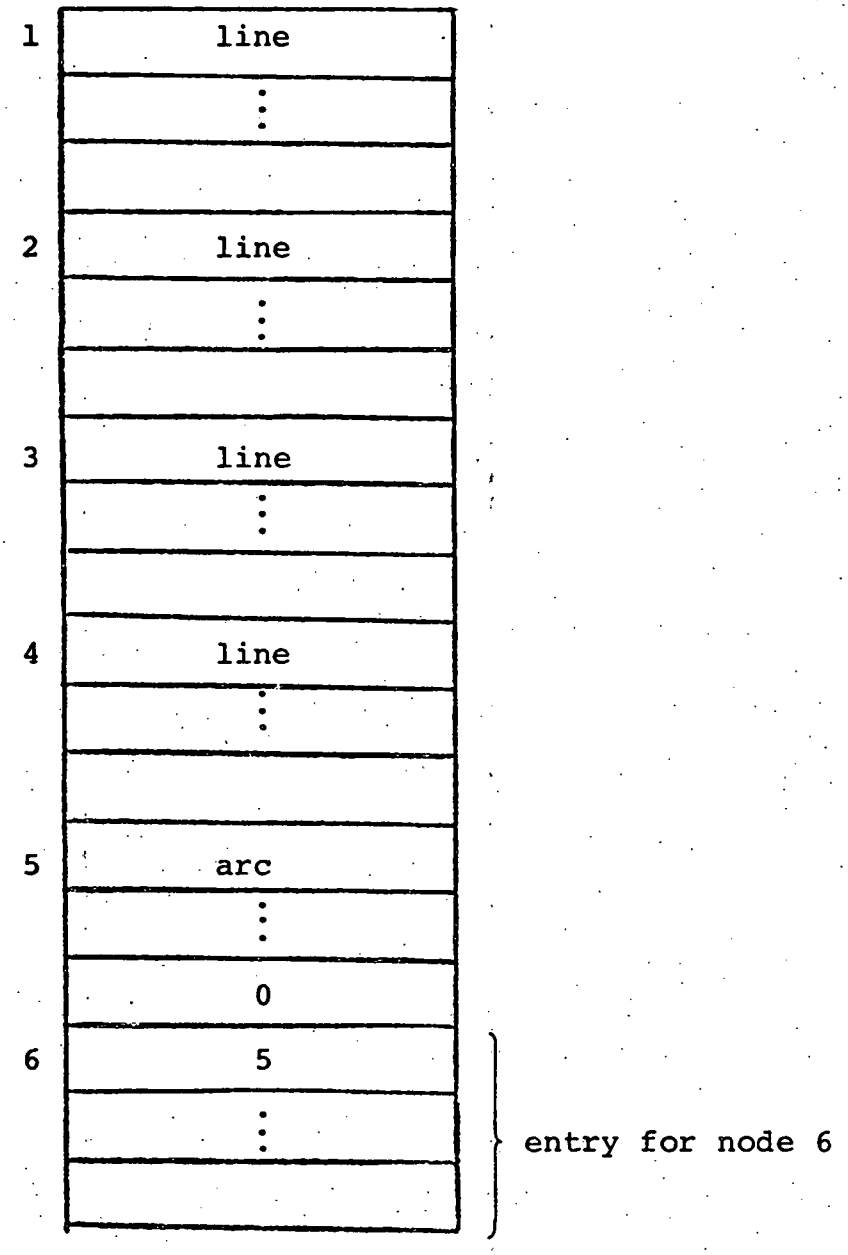


Figure 13 (Continued)

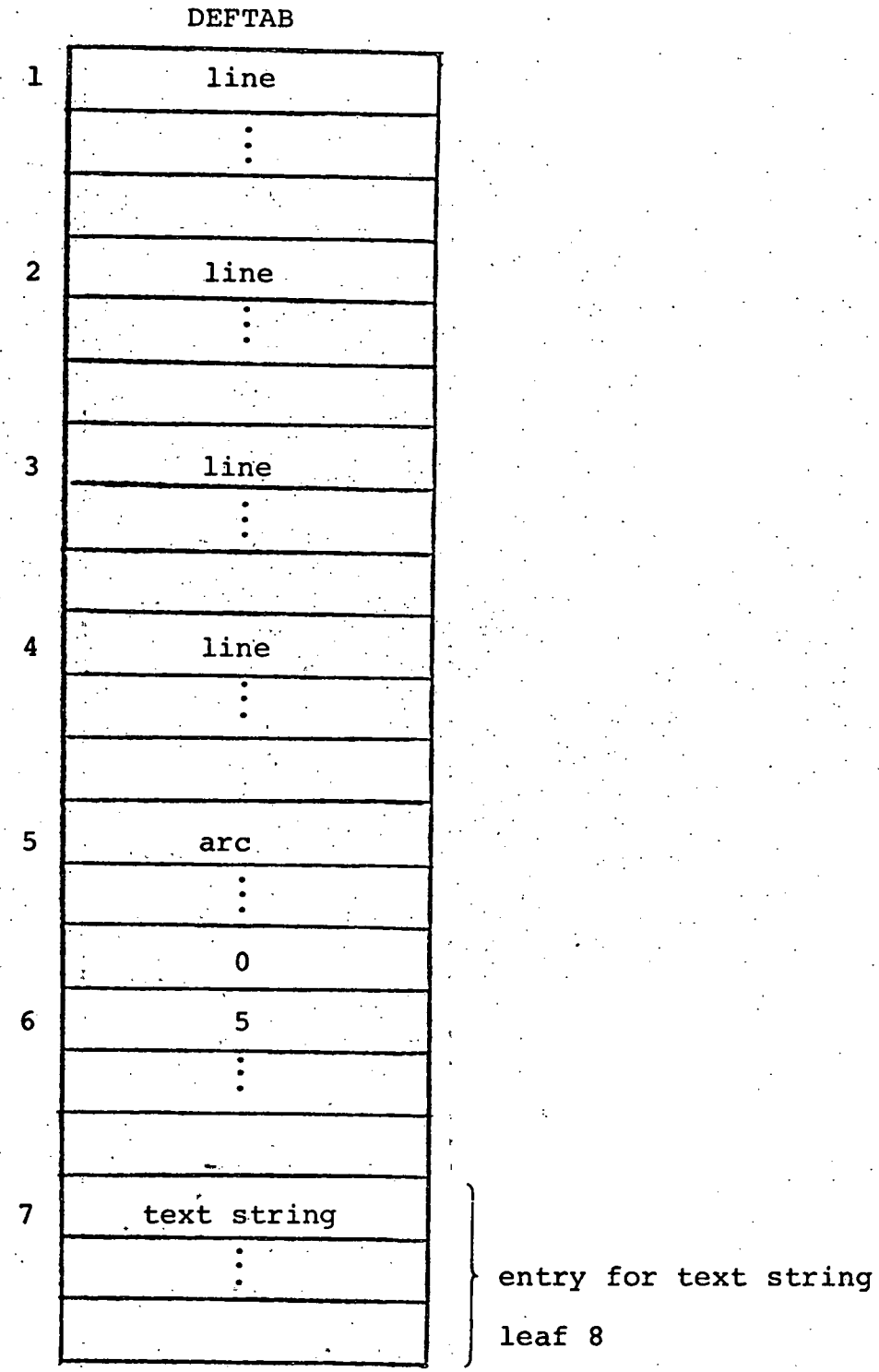


Figure 13 (Continued)

DEFTAB

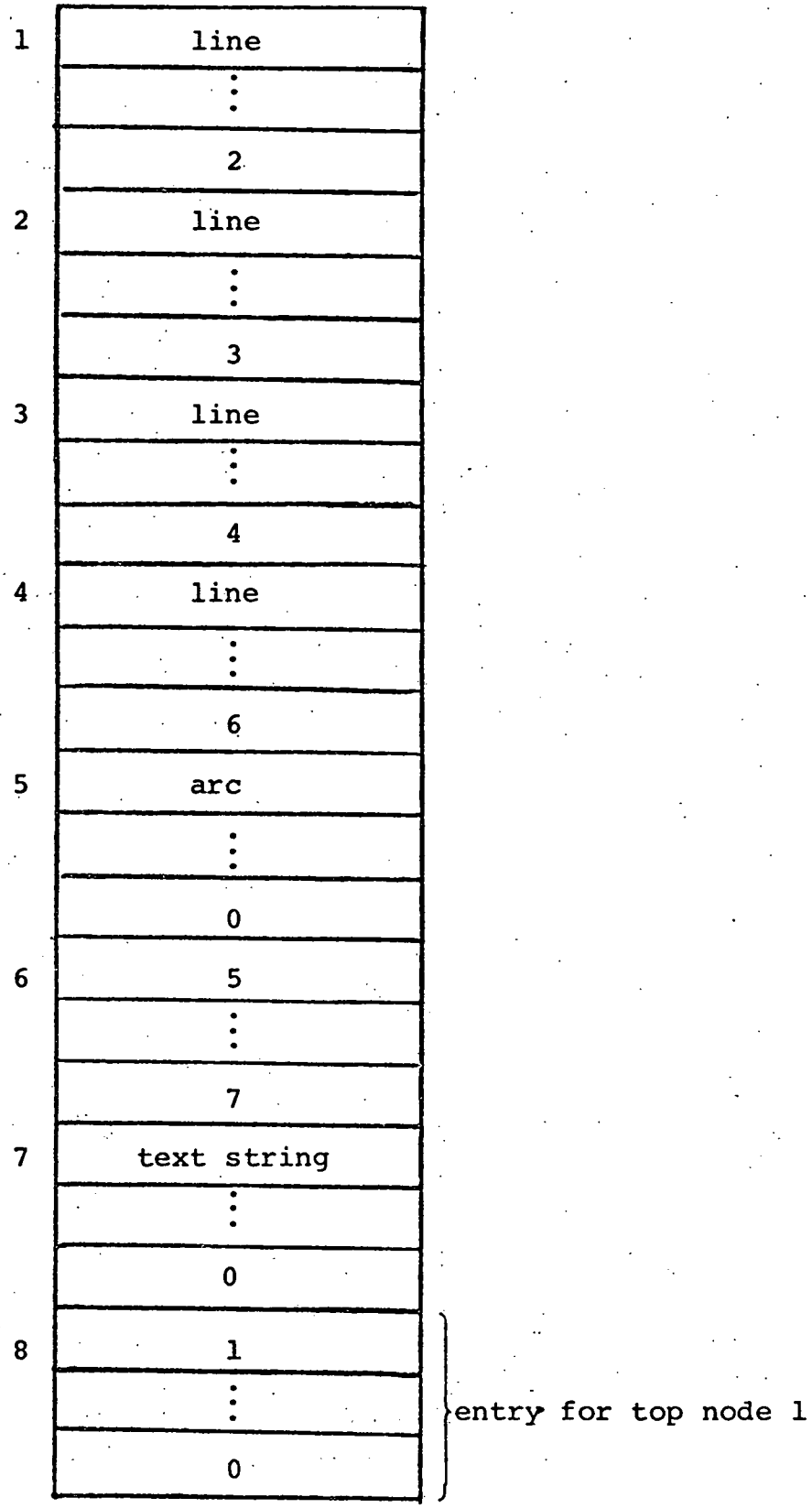


Figure 13 (Continued)

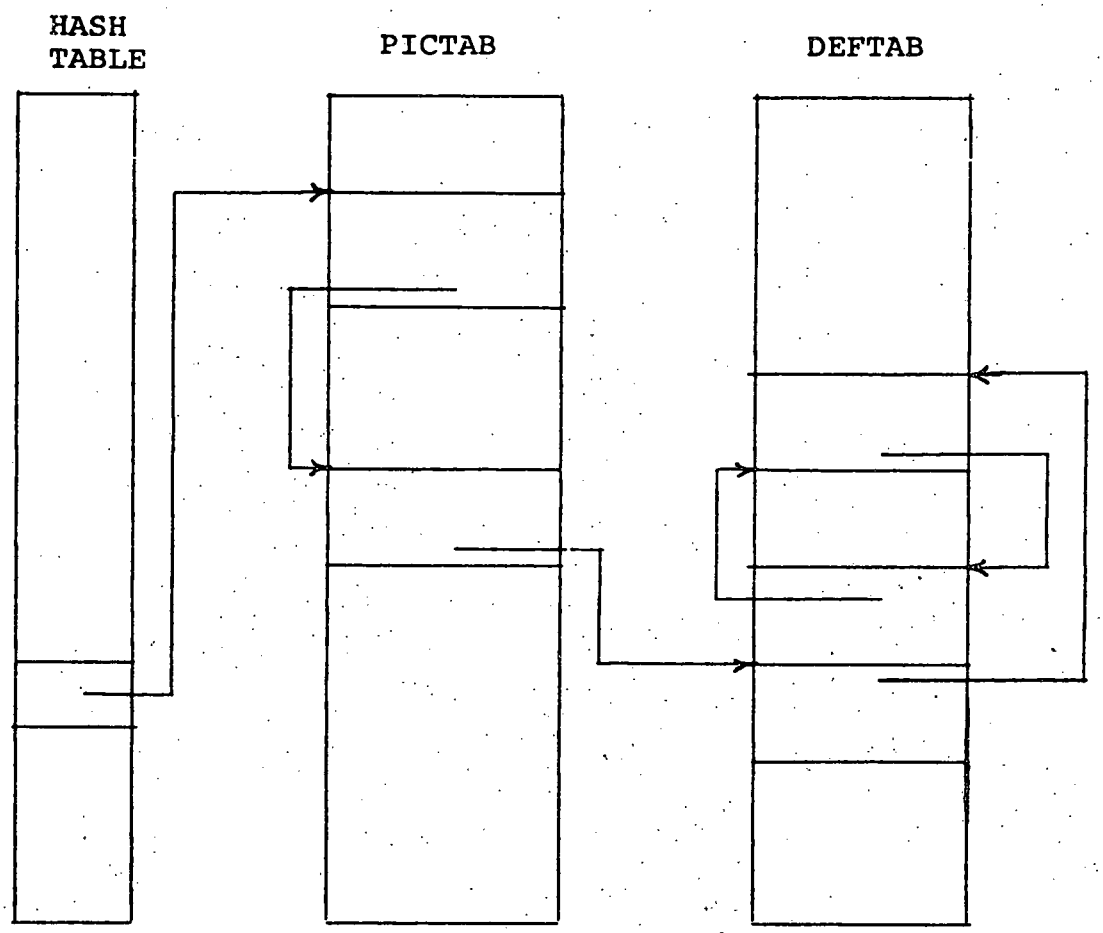
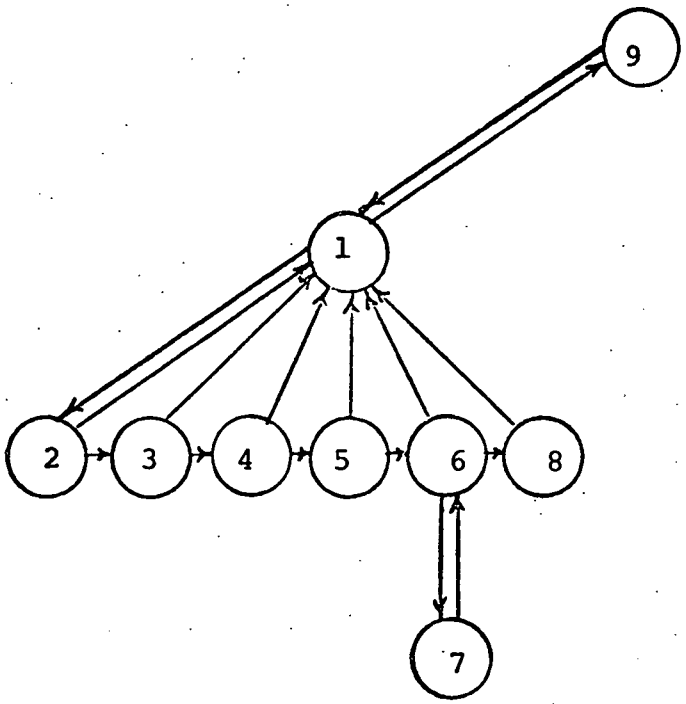
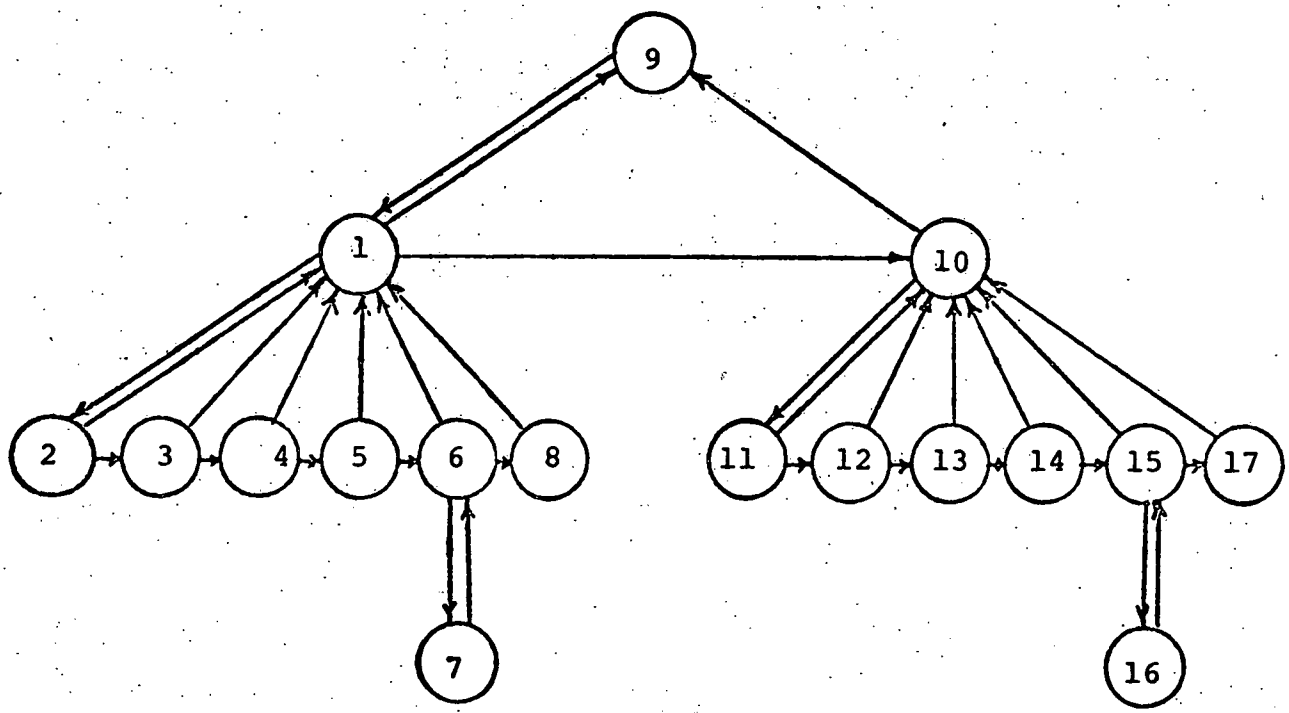


Figure 14. Pointer Chasing in Retrieving a Picture's Definition from DEFTAB.



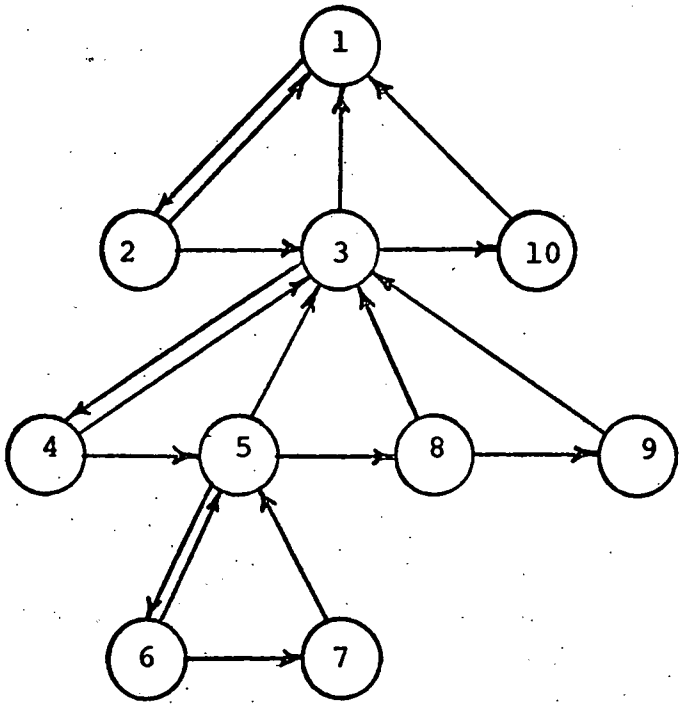
(a) Current Structure.

Figure 15. Adding a Subpicture to the Virtual Structure.

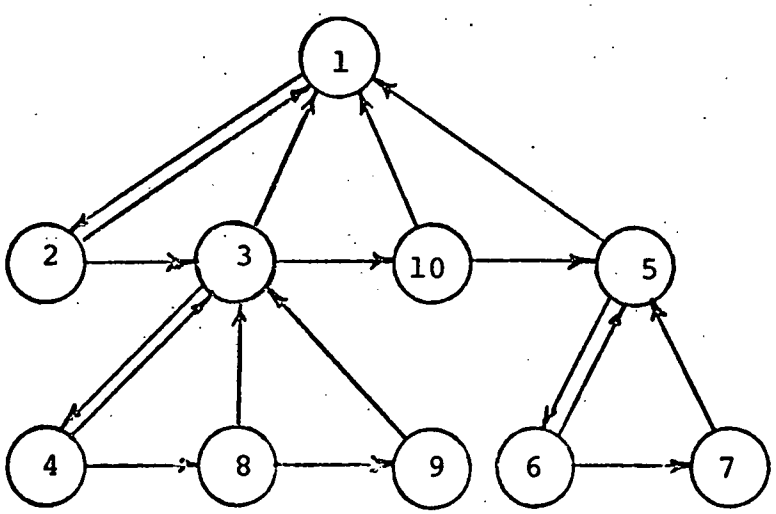


(b) New Structure

Figure 15 (Continued)



(a)



(b)

Figure 16. Modification of a Picture's Virtual Structure.

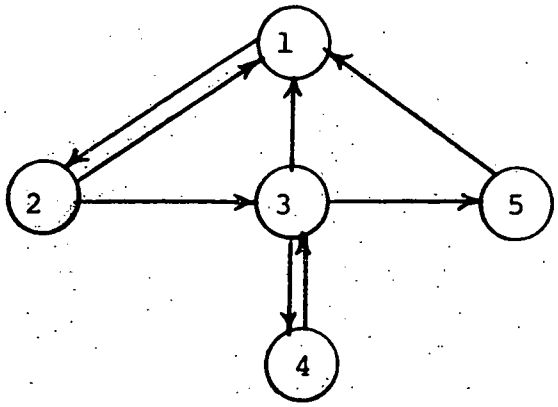


Figure 17. Virtual Structure in which Leaf 2 Represents the Square.

0
0
0
2
0.
0.
0.
1.
0.
0.
0.
1.
0.
0.
0.
1.
1.

Figure 17(a). Top Node Entry.

k (deftab entry ptr)
1
3
-n (index of square)
0.
0.
0.
1.
0.
0.
0.
1.
0.
0.
0.
1.
.5

Figure 17(b). Special Leaf for Square.

0
1
5
4
0.
0.
0.
1.
0.
0.
0.
1.
0.
0.
0.
1.
1.

Figure 17(c). Node for Circle

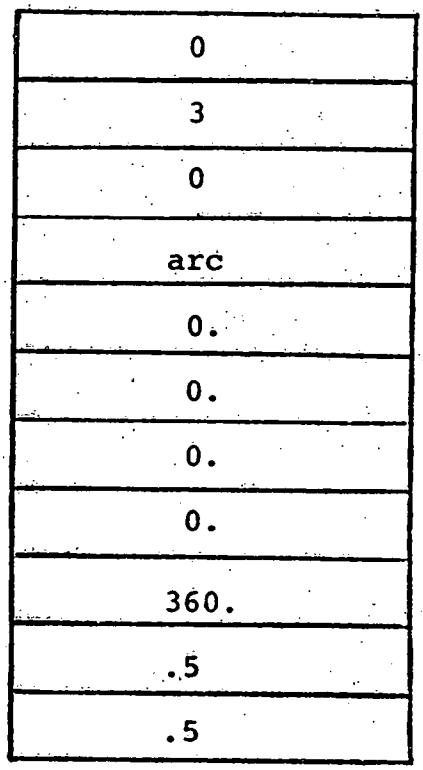


Figure 17(d). Leaf for Circle.

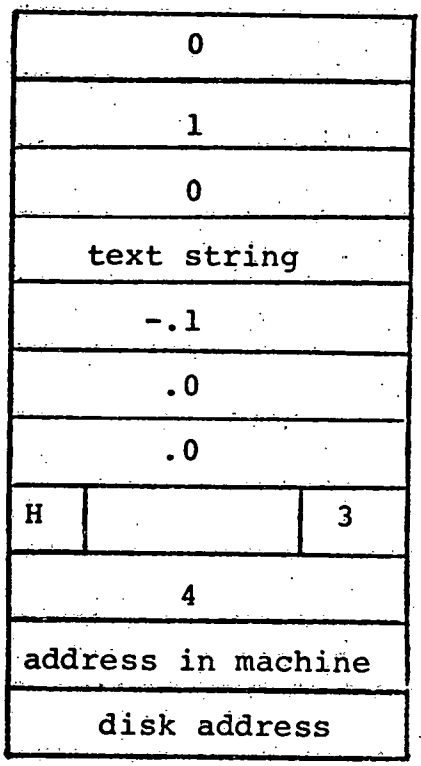


Figure 17(e). Leaf for Text String.

Appendix A

Simple SETL Primer

Atoms

integer	3, -71
real	2.0, -3.1E-14
character strings	'aeiou', nulc (null string)
logical constants	t (true), f (false)
label (of statements)	label: , <label:>
blank (created by function newat)	
Ω	undefined
subroutine	
function	

Operations for Atoms

Integers:	arithmetic:	+, -, *, / , // (remainder)
	comparison:	eq, ne, lt, gt, ge, le
	other:	max, min, abs
Reals:	above arithmetic operations except //	
	plus exponential, log, and trig functions	
Booleans:	logical:	and (or a), or, exor,
		implies (or imp), not (or n)
Strings:		+ (concatenation), # (size)

Set Operations

∈ (membership test); nl (empty set); ∋ (arbitrary element);
(number of elements); eq, ne (equality tests);
with, less (addition and deletion of elements)
e.g. {a,b} with c is {a,b,c}; {a,b,c} less b is {a,c}
{a,b} less c is {a,b}
+ (set union), * (intersection).

Tuples

tuple former: if x,y,...,z are n SETL objects

t =<x,y,...,z> is an n-tuple

#t is the number of components of t

t(k) is the kth component of t

t(i:j) is the tuple whose components, for $1 \leq k \leq j$, are t(i+k-1)

hd t is t(1)

tl t is t(2:)

+ is the concatenation operator for tuples

Set Definition

by enumeration: {a,b,...,c}

set former: $\{e(x_1,\ldots,x_n),\ x_1 \in e_1,\ x_2 \in e_2(x_1),\ldots,$
$x_n \in e_n(x_1,\ldots,x_{n-1}) \mid C(x_1,\ldots,x_n)\}$

Loop Control

(while cond) block;

(while cond doing blocka) blockb; is the same as

(while cond) blockb blocka;

($\forall x \in s$) block; means for all elements of the set s

execute the block

($i \leq \forall x \leq n$) block; means for all integers between i and n

execute block (equivalent to a DO-loop in PL/1)

Existential Quantifier

$\exists x \in s \mid c(x)$ means pick the first x in the set s such that c(x) is t

$max \geq \exists x \geq min \mid b(x)$ means pick the largest x such that b(x) is true

Similarly for $min \leq \exists x \leq max$, $a \leq \exists x \leq b$ etc.

If there exist no x then the value of the expression is Ω.

Subroutines and Functions (always recursive)

to call:

sub(param$_1$,...,param$_n$);

or

infix: p_1 sub p_2

or

prefix: sub p_1

to define:

subroutine:

define sub(p_1,p_2,...,p_n); text end sub;

return; to return from subroutine

or

define p_1 insub p_2; text end insub;

or

define presub p; text end presub;

function:

definef fun(p_1,...,p_n); text end fun;

return val; to return from a function

or

definef p_1 infun p_2; text end infun;

or

definef prefun p; text end prefun;

Operator with Special Side Effects

expn is x has same value as expn and assigns this value to x.

Appendix B

addprimitive

```
define addprimitive(entry);  /* a routine to add a previously
     undefined line, arc, or text entry to the virtual set */
  vptr = max virtual + 1;
  virtual = virtual with vptr;
  son(top) = son(top) with vptr;  /* add to set of first
     generation descendants of main node of structure */
  parent(vptr) = top;   pictureid(vptr) = nl;
  flow                  isarc?
            enterarc,         (tlist = tlist+<vptr>;)+
                                    isline?
                              enterline,    entermessage;
isarc    := leaftype(entry) eq nl
isline   := leaftype(entry) eq lline
enterline: vendl(vptr) = endl(entry);
           vend2(vptr) = end2(entry);
entermessage: vcenter(vptr) = center(entry);
              vtext(vptr)   = text(entry);
enterarc:  vcenter(vptr)  = center(entry);
           vrotmat(vptr)  = rotmat(entry);  /* ang. orientation*/
           vscale(vptr)   = ascale(entry);
           vleaf          = max virtual + 1;
           virtual        = virtual with vleaf;
           son(vptr)      = vleaf;
           vcenter(vleaf) = vcenter(vptr);
           vstart(vleaf)  = start(entry);
           vstop(vleaf)   = stp(entry);
           vmajor(vleaf)  = mjr(entry);
           vminor(vleaf)  = mnr(entry);
           parent(vleaf)  = vptr;
           pictureid(vleaf)=nl;
           tlist          = tlist + <vleaf>;
                                                  end flow;
return;
end addprimitive;
```

Appendix C

definitionsave

```
define definitionsave;  /* a routine to  name and save the
     virtual structure that represents the current picture
     display. the definition is added to deftab. the defining
     positional values of each node are relative to the node's
     parent. in many cases the same subset of nodes can define
     several instances of the subpicture in this way */
picsearch(ppic, eflg);  /* is the   picture already defined */
if eflg eq 0
/* yes */ then mvhresp('illegal picture name, possible
                                          double definition');
        return;
end if eflg;
if delsw eq 0 and #son(top) eq 1
  /* if the current picture has not been modified and it
     consists of only one subpicture then just record the new
  then namerec(pictureid(son(top)));                    name*/
        return;
end if;
definition = nl;
(∀vptr ∈ son(top)) builddef(top,definition,vptr);;
  /* build the definition of top in deftab recursively */
  deftop = max deftab+1;  deftab = deftab with deftab;
  dcenter(deftop) = vcenter(top);  /* center of picture */
  drotmat(deftop) = vrotmat(top);  /* rotation matrix */
  dscale(deftop) =  vscale(top);   /* scale */
  idpt(deftop) = definition;
  namerec(deftop); /* record name and definition in pictab */
  pictureid(top) = deftop;
  /* now make this newly defined picture a subpicture
     and create a new top node */
  new = max virtual + 1;   virtual = virtual with new;
  pictureid(new) = nl;
  parent(new)    = nl;
```

```
    son(new) = {top};
    vcenter(new) =<<0,0,0>;
    vrotmat(new) = identitymatrix;
    vscale(new) = 1;
    parent(top) = new;
    top = new;
    return;
    end definitionsave;

define builddef(father,definition,vptr);
    /* a recursive routine to translate the virtual structure
       into a subpicture definition in deftab */
    /* check to see if this node or leaf, vptr, is defined or
       whether it or one or more of its descendants has been
       modified */
    if pictureid(vptr) ne nl
       then /* it is defined so add to definition */
       definition = definition with pictureid(vptr);
       return;
    end if;
    /* this node or leaf is undefined, so it must be defined
                                                    first */
    flow              isprimitive?
        whichprimitive? 4          definenode,
    dline, darc, dmess, dspec;
whichprimitive:  subflow(linelt, arcelt, messelt, specelt)
                      isline?
                 linelt,     isarc?
                        arcelt,  ismess?
                           messelt, specelt;
isprimitive := leaftype(vptr) ne Ω
isline      := leaftype(vptr) eq lline
isarc       := leaftype(vptr) eq arc
ismess      := leaftype(vptr) eq message
```

```
definenode: defvptr = nl;  /* initialize definition set for vptr*/
      (∀v ∈ son(vptr))
      builddef(vptr,defvptr,v);;
   /* calculate relative values for definition */
   <tx,ty,tz> = (vcenter(vptr) tsub vcenter(father))
                              vescal(1/vscale(father));
   defnode = max deftab + 1;  deftab=deftab with defnode;
   drotmat(defnode) = matmul(matinv(vrotmat(father)),vrotmat
                                                  (vptr));
   dcenter(defnode) = trot(matinv(vrotmat(father)),<tx,ty,tz>);
   dscale(defnode)  = vscale(vptr)/vscale(father);
   idpt(defnode)    = defvptr;
   pictureid(vptr)  = defnode;
dline:  /* calculate relative values for definition */
   defleaf = max deftab + 1;
   idpt(defleaf) = lline;
   <tx,ty,tz> = (vendl(vptr) tsub vcenter(father))
                              vescal (1/vscale(father));
   dendl(defleaf) = trot(matinv(vrotmat(father)),<tx,ty,tz>);
   <tx,ty,tz> = (vend2(vptr) tsub vcenter(father))
                              vescal (1/vscale(father));
   dend2(defleaf) = trot(matinv(vrotmat(father)),<tx,ty,tz>);
   pictureid(vptr) = defleaf;
   deftab = deftab with defleaf;
darc: defleaf = max deftab+1;
      idpt(defleaf) = arc;
      dcenter(defleaf) = <0,0,0>;
      dstart(defleaf) = vstart(vptr);
      dstop(defleaf) = vstop(vptr);
      dminor(defleaf) = vminor(vptr);
      dmajor(defleaf) = vmajor(vptr);
      pictureid(vptr) = defleaf;
      deftab = deftab with defleaf;
dmess:defleaf = max deftab+1;
      idpt(defleaf) = message;
```

```
    /* calculate relative start of message */
    <tx,ty,tz> = (vcenter(vptr) tsub vcenter(father))
                              vescal(1/vscale(father));
    dcenter(defleaf) = trot(matinv(vrotmat(father)),<tx,ty,tz>);
    pictureid(vptr) = defleaf;
    dtext(defleaf) = vtext(vptr);
    deftab = deftab with defleaf;
dspec: defleaf = max deftab+1;
       idpt(defleaf) = son(vptr);
    /* calculate relative values */
    <tx,ty,tz> = (vcenter(vptr) tsub vcenter(father))
                              vescal(1/vscale(father));
    dcenter(defleaf) = trot(matinv(vrotmat(father),<tx,ty,tz>);
    drotmat(defleaf) = matmul(matinv(vrotmat(father)),
                                    vrotmat(vptr));
    dscale(defleaf) = vscale(vptr) /vscale(father);
    pictureid(vptr) = defleaf;
    deftab = deftab with defleaf;
                                                    end flow;
    definition = definition with pictureid(vptr);
    return;
    end builddef;
```

Appendix D

modifypicture

```
define modifypicture(changes); /* a routine to reflect in
     the "virtual" set changes desired by the user in the
     picture display.  "changes" is the set of elements in
     "virtual" which are to be modified  */
  /* calculate a rotation matrix rmat based on the direction
     angles deltaalpha, deltabeta, deltagamma, and angle of
     rotation deltatau */
  rmat = rotmap(deltaalpha, deltabeta, deltagamma, deltatau);
  (∀vptr ∈ changes)
     if vptr ne top  /* are we modifying the whole picture */
       then          /* no */
       i = modvlinks(vptr);
       if i eq 'no'  /* was this node already in son(top) */
          then       /* no, mark it and its ancestors as
                        no longer being defined in deftab */
          tnvptr = parent(vptr);
          pictureid(vptr) = nl;
          parent(vptr) = top;
          (while tnvptr ¬= top doing tnvptr=parent(tnvptr);)
               pictureid(tnvptr) = nl;
          end while;
          son(top) = son(top) with vptr;
       end if i eq 'no';
  end if vptr;
  /* now modify the components of the node */
  modvirtual(vptr);
  end ∀vptr;
  return;
  end modifypicture;
```

```
definef modvlinks(node); /* a function which checks to see
    if parent(node) eq top. if so, it returns a value of
    'yes' indicating that node is already an immediate
    descendant of top. if parent(node) ne top then node
    is removed from set of descendants of its parent and
    the value 'no' is returned */
  cvptr = node;
  (while parent(node) ne top)
     tnvptr = parent(node);
     if #son(tnvptr) gt 1  /* more than 1 son  */
       then /* remove from set */
       son(tnvptr) = son(tnvptr) less node;
       if leaftype(cvptr) ne arc
          then node = cvptr;
          else node = parent(cvptr);
                 /* no need for undefined ancestors */
       end if;
       return 'no';
       else
       node = tnvptr;
     end if;
  end while;
  return 'yes';
end modvlinks;
```

```
define modvirtual(vptr);  /* a subprogram which modifies the
     components of the vptr entry of the virtual set
     according to values in the deltaset.  if vptr has any
     descendants, modvirtual recursively modifies them */
  flow                  isprimitive?
    (tchng=tchng with vptr;)+  modnodeentry+
     whichprimitive?  4      ((∀v∈son(vptr))modvirtual(v);;),
   mline, marc, mtext, mspec;
  whichprimitive: subflow (linelt, arcelt, messelt, specelt)
                        isline?
              linelt,                  isarc?
                                arcelt,   ismess?
                                      messelt, specelt;
isprimitive:= leaftype(vptr) ne Ω
isline     := leaftype(vptr) eq lline
isarc      := leaftype(vptr) eq  arc
ismess     := leaftype(vptr) eq message
modnodeentry:   /* calculate rotated displacement */
     <tx,ty,tz> = vcenter(vptr) tadd <-rx,-ry,-rz>;
/* <rx,ry,rz> is the origin of the axis about which
                          the rotation occurs */
   <xℓ,yℓ,zℓ>= trot(rmat,<tx,ty,tz>)
               tadd <rx,ry,rz>;
/* now scale and add in displacement */
  vcenter(vptr) = (<xℓ,yℓ,zℓ> vescal deltascale) tadd
                          <deltax,deltay,deltaz>;
  /* modify rotation matrix */
  vrotmat(vptr) = matmul(rmat, vrotmat(vptr));
mline:   /* change line endpoints */
  <tx,ty,tz> = vendl(vptr) tadd <-rx,-ry,-rz>;
  <xℓ,yℓ,zℓ> = trot(rmat,<tx,ty,tz>)
               tadd <rx,ry,rz>;
  vendl(vptr) = (<xℓ,yℓ,zℓ> vescal deltascale) tadd
                              <deltax, deltay, deltaz>;
  <tx,ty,tz>  = vend2(vptr) tadd <-rx,-ry,-rz>;
  <xℓ,yℓ,zℓ>  = trot(rmat,<tx,ty,tz>) tadd <rx,ry,rz>);
```

```
  vend2(vptr) = (<xℓ,yℓ,zℓ> vescal
          deltascale) tadd <deltax,deltay,deltaz>;
marc:  /* an arc has the same transformed center as
                          its parent */
  vcenter(vptr) = vcenter(parent(vptr));
mtext:   /* transform position of start of text */
  <tx,ty,tz> = vcenter(vptr) tadd <-rx,-ry,-rz>;
  <xℓ,yℓ,zℓ> = trot(rmat,<tx,ty,tz>) tadd <rx,ry,rz>;
  vcenter(vptr) = (<xℓ,yℓ,zℓ> vescal deltascale
                          tadd <deltax,deltay,deltaz>;
mspec:   /* transform center, adjust scale, and
                          rotation matrix */
  <tx,ty,tz> = vcenter(vptr) tadd <-rx,-ry,-rz>;
  <xℓ,yℓ,zℓ> = trot(rmat,<tx,ty,tz>) tadd <rx,ry,rz>;
  vcenter(vptr) = (<xℓ,yℓ,zℓ> vescal deltascale)
                          tadd <deltax,deltay,deltaz>;
  vscale(vptr) = vscale(vptr) * deltascale;
  vrotmat(vptr) = matmul(rmat,vrotmat(vptr));
                                                end flow;
return;
end modvirtual;

definef a tadd b;  /* this function does component addition
                          of tuples a and b */
  if #a eq 1
     then return <hd a + hd b>;
     else return <hd a + hd b , tℓ a tadd tℓ b>;
  end if;
  end tadd;

definef a vescal b;  /* this function multiplies the vector a
                                          by the scalar b */
  if #a eq 1
     then return <hd a * b>;
     else return <hd a*b, tℓ a vescal b>;
  end if;
end vescal;
```

Appendix E

Communication Formats for Data between the Honeywell 316 and the CDC 6600

From the CDC 6600 to the Honeywell 316

All blocks from the CDC 6600 will have a six word header which is consistent with the format of a display file. The first word of the header will have the total number of words in the block. The right byte of word 6 will describe the type of the block. At the moment, ten types are assigned.

Type	Data
0	A standard display file
1	A display segment for a point list
2	A display segment for an edge list
3	A display segment for a polygon list
4	A Vector General ASCII message for display to the user
5	A point list
6	An edge list
7	A polygon list
8	A part of the VIRTUAL structure
9	A leaf list and the corresponding segment list. It precedes a block of type 0.

In blocks of types 0 - 3, words 2 - 5 may have a one to eight character name.

In a user message block (type 4), word 7 contains the number of words in the message which begins in word 10.

In a point list block (type 5), word 7 contains the number of points in the list. If word 7 is negative the points have two dimensional coordinates, otherwise they have three dimensional coordinates. After word 7 comes first the list of x coordinates, then y coordinates, and then possibly z coordinates.

In an edge list block (type 6), word 7 contains the number of edges in the list. Next comes the list of points which are the first end points and then the list of points which are the second end points of the edges. Finally comes the point list which is of the form described for block 5.

In a polygon list (type 7), word 7 contains the number of polygons. Next for each polygon comes a word with the number of edges in the polygon and a list of the edges. After all the lists of the polygons' edges, comes an edge list in the form of a list from a type 6 block.

Blocks of types 8 and 9 will not be defined here because there will be no immediate implementation.

From the Honeywell 316 to the CDC 6600

Currently, all transmissions to the CDC 6600 are in the form of batch "JOB" streams or are queries about or requests for data from the CDC 6600. The only data concatenated to a "JOB" stream would be a text string representing the command for processing.

Appendix F

Graphics Library Package

In order to make graphics programming easy for an application designer, a library of graphics subroutines has been provided. The library has routines whose calls are similar to those of the programs provided in a CALCOMP package for a plotter.

Plot Initialization

Before a user can create a display list, he must call the plot initialization routine

```
CALL PLTINIT(FILENAME,ANGLES,USERSCALE,USERORIGIN,PLOTDIMS);
```

where FILENAME is a self-defined string of at most eight characters which will be the name of the file; ANGLES is a real array of four elements representing the four direction angles, α, β, γ, and τ; the USERSCALE is the maximum real value to be displayed; the USERORIGIN is a real array of picture displacements, dx, dy, (and dz); PLOTDIMS is either 2 or 3 to indicate a two or three dimensional drawing. If the FILENAME is the empty string then the file is given the name 'NAMELESS'.

Drawing Routines

In order to draw a two dimensional picture, the user can write a sequence of commands of the form

```
CALL PLT2D(X,Y,BEAM);
```

where X, Y are real user coordinates of the new point and the value of BEAM determines the type of line drawn.

BEAM = 1; the beam draws a solid line as it moves
= 2; the beam draws a dashed line as it moves
= 3; the beam draws a dotted line as it moves
= 4; the beam draws an end point only at the end of a move
< 0; the beam moves without drawing.

In order to draw a three dimensional picture, the programmer uses

```
CALL PLT3D(X,Y,Z,BEAM);
```

where X, Y, Z are the real coordinates of the new point and BEAM takes on the values indicated above.

Positioning the Beam

To position the beam on the Z = 0 plane, the programmer writes

```
CALL MOV2D(X,Y);
```

where X,Y are real user coordinates for the beam position.

To position the beam in three space, one writes

```
CALL MOVB(X,Y,Z);
```

where X,Y,Z are real user coordinates for the beam position.

Establishing a New Origin

To establish a new absolute origin for two dimensional plotting one uses

```
CALL ORIGIN2D(X,Y);
```

where X,Y are the user's real absolute coordinates for the new origin.

In three space, one uses

```
CALL ORGIN(X,Y,Z);
```

where X,Y,Z are real absolute coordinates. In either case the origin is noncumulative. The coordinates are relative to the center of the screen.

Graph Plotting

To draw a graph that plots a list of points, the programmer can use

```
CALL GRAPH(XARRAY,YARRAY,NUM,CYCLESIZE,BEAM,SYMBOL,XSCALE,YSCALE);
```

where XARRAY,YARRAY are real arrays of X and Y coordinates; NUM is the number of points to be plotted; CYCLESIZE is the repeat cycle of points graphed: 1 + I * CYCLESIZE for I = 1 to NUM; BEAM is the mode for the graph; SYMBOL is a special ASCII symbol to mark points; XSCALE is the real scale

factor for the x coordinates; and YSCALE is the real scale factor for the y coordinates. If SYMBOL = 0 then no special symbol marks the points.

Solid of Revolution

To produce a solid of revolution about the Y-axis from a curve in the x,y plane, the user can call the routine MAKESOLID.

```
CALL MAKESOLID(XARRAY,YARRAY,NUM,CYCLESIZE,BEAM,SYMBOL,
                                        XSCALE,YSCALE);
```

where the parameters are the same as those for GRAPH. The original two dimensional curve is repeated at forty degree intervals around the y-axis.

Axis Drawing

To produce a horizontal or vertical axis with labels, the user can call the routine AXIS.

```
CALL AXIS(X,Y,TEXT,CHZ,LENGTH,ANG,ASTART,DELTA,DIV,XSCALE,YSCALE);
```

where X,Y are user coordinates for the start of the axis; TEXT is a self-defined string for describing the axis; CHZ is the character size (1-4) of TEXT; LENGTH is the length of the axis; ANG is 0. or 90. to describe the direction of the axis; ASTART is the real value to be written at the start of the axis; DELTA is the change in value for AXIS numbering per division; DIV is the number of division marks

on the axis; XSCALE is a real scale factor for the x coordinates; and YSCALE is a real scale factor for the y coordinates.

Text

In order to write a string of text on the display, the programmer may write

```
CALL SYMBOL(X,Y,Z,TXT,CHZ,ANG);
```

where X,Y,Z are scaled user coordinates whose magnitudes are less than or equal to the user scale defined in the plot initialization routine; TXT is the character string to be plotted; CHZ is the character size (1-4); and ANG = 0. or 90. for either horizontal or vertical text.

Real Numbers

To format the first N characters of a real number in ASCII for the display device, a programmer uses

```
CALL FORMFLT(VALUE,ASC,N);
```

where VALUE is the real number; ASC is the array that holds the ASCII; and N is the number of characters in ASC.

ASCII CHARACTERS

When a user formats an ASCII character block in an array, he can add it to the display list with

```
CALL CHARBLOCKADD(BLOCK,LENGTH,SIZE,DIR);
```

where BLOCK is the array containing the characters(2 charac-

ters/word); LENGTH is the number of words in the BLOCK; SIZE is the character size (1-4); and DIR is 0. or 90. for a horizontal or vertical direction. The routine adds a character instruction and the character block to the display file and makes sure the block terminates properly.

Two Dimensional Rotation

To change the rotation matrix from the original setting in PLTINIT, the user may write

```
CALL ROTATION(THETA);
```

where THETA is a real number in degrees. The rotation matrix is reinitialized to the value calculated from the THETA about the Z-axis.

Miscellaneous Routines

The routines which follow would probably not be used by most users.

To load any number of display address registers (in the CRT controller)

```
CALL LOADR(DAR,VAL,NVAL);
```

where DAR is the first display address register; VAL is an array of values; and NVAL is the number of values to be loaded.

To determine the current position of the beam in user coordinates unless the previous command produced text, use

```
CALL WHERE(X,Y,Z);
```

where X,Y, and Z contain the last user coordinates.

To decide whether plotting is permissible, one can invoke the function

```
GRPHSTAT(X)
```

where X is a dummy argument to determine the system status. If GRPHSTAT is nonzero then plotting is permissible, otherwise it is not.

Plot Termination

To terminate a plot,

```
CALL PLOTFIN;
```

On the Honeywell 316 this just terminates the last command. On the CDC 6600 it also packs the display list into 60 bit words and then writes a file. This file can be routed to the satellite computer.

Bibliography

Bauer, F. L. (1972). "Software Engineering", Proceedings of the IFIP Congress 71, edited by C. V. Freiman, North-Holland, pp. 530-538.

Bauer, F. L. (ed.) (1973). Advanced Course on Software Engineering, Springer-Verlag.

Bergeron, R. D. et al. (1972). "Systems Programming Languages", Advances in Computers 12, pp. 175-284.

Boardman, T. L., Jr. (1974). "Hardware/Software Design Considerations for High Speed/Low Cost Interactive Graphic Communication Systems", Proceedings AFIPS 1974 National Computer Conference, Vol. 43, AFIPS Press, pp. 273-278.

Boukens, J. and Deckers, F. (1974). "Chief, An Extensible Programming System", in (van der Poel, 1974), pp. 129-148.

Christensen, C. and Pinson, E. N. (1967). "Multifunction Graphics for a Large Computer System", Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, pp. 697-711.

Cotton, I. and Greatorex, F. S., Jr. (1968). "Data Structures and Techniques for Remote Computer Graphics", Proc.AFIPS 1968 FJCC, Vol. 33, Pt. 1, AFIPS Press, pp. 533-544.

Denert, E. et al. (1975). "GRAPHEX68: Graphical Language Features in Algol 68", Computers and Graphics, Vol. 1, Pergamon Press, pp. 195-202.

Dennis, J. B. (1973). "The Design and Construction of Software Systems", in (Bauer, 1973), pp. 12-27.

Dill, J. C. and Thomas, J. J. (1975). "On the Organization of a Remote Low Cost Intelligent Satellite Graphics Terminal", Computer Graphics 9, 1, pp. 1-8.

Foley, J. D. (1971). "An Approach to the Optimum Design of Computer Graphics Systems", CACM 14, 6, pp. 380-390.

Foley, J. D. (1973). "Software for Satellite Graphics Systems", Proceedings ACM 1973 Annual Conference, pp. 76-80.

Gammil, R. C. and Robertson, D. (1973). "Graphics and Interactive Systems-Design Considerations of a Software System", Proc. AFIPS 1973 National Computer Conference, Vol. 42, AFIPS Press, pp. 657-662.

Garwick, J. (1972). "Sydel", in (van der Poel, 1974), pp. 519-524.

Geiselbrechtinger, F. et al. (1974). "Language Layers, Portability, and Program Structuring", in (van der Poel, 1974), pp. 79-99.

Hamlin, G. Jr. and Foley, J. D. (1975). "Configurable Applications for Graphics Employing Satellites (CAGES)", Computer Graphics 9, 1, pp. 9-19.

Hamlin, G. Jr. (1976). "Configurable Applications for Satellite Graphics", Proc. of the Third Annual Conference on Computer Graphics, Interactive Techniques and Image Processing - Siggraph '76, ed. by U. W. Pooch, pp. 196-203.

Horning, J. J. (1972). "XPL", in (van der Poel, 1974), pp. 529-531.

Ichbiah, J. D. et al. (1974). "The Two Level Approach to Data Independent Programming in the LIS System Implementation Language", in (van der Poel, 1974), pp. 161-169.

Joyce, J. D. and Cianciolo, M. J. (1967). "Reactive Displays: Improving Man-Machine Graphical Communication", Proc.AFIPS 1967 FJCC, Vol. 31, AFIPS Press, pp. 713-721.

Kilgour, A. C. (1971). "The Evolution of a Graphics System for Linked Computers", Software-Practice and Experience Vol. 1, pp. 259-269.

Klunder, J. (1974). "Experience with SPL", in (van der Poel, 1974), pp. 385-393.

Lindsey, C. H. (1972). "Algol 68 with Fewer Tears", Comput. J. 15, 2, pp. 176-188.

Michel, J. and van Dam, A. (1976). "Experience with Distributed Processing on a Host/Satellite Graphics System", Proc. of the Third Annual Conf. on Computer Graphics, Interactive Techniques, and Image Processing -- Siggraph '76, ed. by U. D. Pooch, pp. 196-203.

Morris, J. B. (1973). "A Comparison of Madcap and SETL", Los Alamos Scientific Lab., U. of California, Los Alamos, New Mexico.

Newman, W. M. and Sproull, R. F. (1973). Principles of Interactive Computer Graphics, McGraw-Hill Book Co.

Newman, W. M. (1975). "Instance Rectangles and Picture Structure", Proc. of the Conference on Computer Graphics, Pattern Recognition, and Data Structure, May 14-16, 1975, pp. 297-301.

Schwartz, J. T. (1973). On Programming: An Interim Report on the SETL Project, Installment II (now combined with Installment I): The SETL Language and Examples of Its Use, Courant Inst., New York Univ.

Schwartz, J. T. (1975). "Automatic Data Structure Choice in a Language of Very High Level", Comm. ACM 18, 12, pp. 722-728.

Shields, D. (1976). "Guide to the LITTLE Language", Courant Inst., New York Univ.

Solow, R. M. (1973). "On Equilibrium Models of Urban Location", Essays in Modern Economics, ed. by J. M. Parkin, Longmans, pp. 2-16.

Strauss, Charles M. (1974). "Computer-encouraged Serendipity in Pure Mathematics", Proc. of the IEEE, 62, 4, pp. 493-495.

Stuart, T. (1975). "Adapting Large Systems to Small Machines", Courant Inst., New York Univ.

Stuart, T. (1976). "A Minicomputer System Language", Courant Inst., New York Univ.

Van Dam, A. and Evans, D. (1967). "A Compact Data Structure for Storing, Retrieving, and Manipulating Line Drawings", Proc. of SJCC, Vol. 30, Thompson Book. Co., pp. 601-610.

Van Dam, A. (1971). "Microprogramming for Computer Graphics", SIGGRAPH, 5, 4.

Van Dam, A. and Stabler, G. M. (1973). "Intelligent Satellites for Interactive Graphics", Proc. AFIPS National Computer Conference 1973, pp. 229-238.

Van Dam, A. et al., (1974). "Intelligent Satellites for Interactive Graphics", Proc. of the IEEE, 62, 4, pp. 483-492.

van der Poel, W. L. and Maarssen, L. A. (1974). Machine Oriented Higher Level Languages, American Elsevier.

Wells, M. B. (1974). "Algorithmic Languages and Machine Oriented Tasks", in (van der Poel, 1974), pp. 49-65.

Wichman, B. A. and Bell, D. A. (1974). "PL516", in (van der Poel, 1974), pp. 501-508.

Williams, R. (1971). "A Survey of Data Structures for Computer Graphic Systems", Computing Surveys 3, 1, pp. 1-21.

Woodsford, P. A. (1971). "Design and Implementation of the GINO 3D Graphics Software Package", Software-Practice and Experience, 1, 4, pp. 335-366.

Wycherley, R. D. H. (1972). "An Interaction Handling Technique for Satellite Graphics", Proc. of the IFIP Congress 71, C. V. Freiman (ed.), North Holland, pp. 435-439.