INTERFACING A HIGH-LEVEL GRAPHICS LANGUAGE
WITH THE GSPC MACHINE MODEL

by

CHRISTIAN MANNHARDT

B.Sc., The University of Cape Town, 1976

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in
THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
May 1980

© Christian Mannhardt, 1980
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date May 26, 1980
ABSTRACT

Recent efforts in computer graphics to design a graphics standard have contributed much towards the conceptual clarification of basic operations such as modelling, display and transformation functions. This thesis discusses how such concepts are reflected in a high-level graphics language with the aim of providing a tool to write effective and readable application programs. The high-level language is a graphics extension to Pascal. An interface is described how the multi-level graphics modelling system is mapped onto a single-level display system. The particular display system considered is the Core System which is proposed as a graphics standard by the SIGGRAPH-ACM Graphic Standards Planning Committee (GSPC).
CONTENTS

Page

LIST OF FIGURES ................................................................................... vi

ACKNOWLEDGEMENTS ........................................................................... vii

Chapter

1. INTRODUCTION .............................................................................. 1

2. COMPUTER GRAPHICS SOFTWARE AND HARDWARE .................. 5

   2.1 High-level Graphics Languages ............................................. 6

   2.2 LIG, Concepts and Improvements ......................................... 12

   2.3 Portability of Software using the

       Virtual Machine Concept ......................................................... 16

   2.4 Graphics Display Processors ............................................... 22

       2.4.1 Host/Satellite Configurations ....................................... 22

       2.4.2 Hardware of Display Processors ................................. 23

       2.4.3 Microprogrammed Display Processors ......................... 26

   2.5 Achieving Device Independence ............................................ 28

   2.6 Design Criteria for Graphics Packages ............................... 31

       2.6.1 Levels of Picture Naming

           and Modification .............................................................. 32

       2.6.2 Common Graphics Packages ....................................... 34

       2.6.3 Standardization of Graphics Capabilities ..................... 35

       2.6.4 Functions of the Core System ....................................... 36
3. LIG/P LANGUAGE REFERENCE MANUAL

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.2</td>
<td>Terminology, Notation and Vocabulary</td>
</tr>
<tr>
<td>3.3</td>
<td>Program Structure</td>
</tr>
<tr>
<td>3.4</td>
<td>Modelling Functions</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Graphics Types</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Coordinate Systems</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Modelling Transformations and Attributes</td>
</tr>
<tr>
<td>3.4.4</td>
<td>The Synonym Assignment Operator</td>
</tr>
<tr>
<td>3.4.5</td>
<td>The Copy Assignment Operator</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Graphical Functions</td>
</tr>
<tr>
<td>3.4.7</td>
<td>The Superposition Operator</td>
</tr>
<tr>
<td>3.4.8</td>
<td>The Naming Operator</td>
</tr>
<tr>
<td>3.4.9</td>
<td>The Deletion Operator</td>
</tr>
<tr>
<td>3.4.10</td>
<td>The Identification Statements</td>
</tr>
<tr>
<td>3.5</td>
<td>The Display Functions</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Device Definition</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Anonymous and Named Instances</td>
</tr>
<tr>
<td>3.5.3</td>
<td>The Display Statements</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Display Attributes, Viewing and Image Transformations</td>
</tr>
<tr>
<td>3.5.5</td>
<td>The Erase Statement</td>
</tr>
<tr>
<td>3.6</td>
<td>Graphical Input</td>
</tr>
<tr>
<td>3.7</td>
<td>Control Statements</td>
</tr>
<tr>
<td>3.8</td>
<td>Device Setting and Device Inquiry</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Input Device Setting</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 1</td>
<td>29</td>
</tr>
<tr>
<td>Figure 2</td>
<td>51</td>
</tr>
<tr>
<td>Figure 3</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4</td>
<td>54</td>
</tr>
<tr>
<td>Figure 5</td>
<td>54</td>
</tr>
<tr>
<td>Figure 6</td>
<td>65</td>
</tr>
<tr>
<td>Figure 7</td>
<td>65</td>
</tr>
<tr>
<td>Figure 8</td>
<td>66</td>
</tr>
<tr>
<td>Figure 9</td>
<td>72</td>
</tr>
<tr>
<td>Figure 10</td>
<td>72</td>
</tr>
<tr>
<td>Figure 11</td>
<td>88</td>
</tr>
<tr>
<td>Figure 12</td>
<td>88</td>
</tr>
<tr>
<td>Figure 13</td>
<td>90</td>
</tr>
<tr>
<td>Figure 14</td>
<td>91</td>
</tr>
<tr>
<td>Figure 15</td>
<td>91</td>
</tr>
<tr>
<td>Figure 16</td>
<td>95</td>
</tr>
<tr>
<td>Figure 17</td>
<td>96</td>
</tr>
<tr>
<td>Figure 18</td>
<td>115</td>
</tr>
<tr>
<td>Figure 19</td>
<td>116</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to Dr. Guenther Schrack for his valuable help and guidance during the past two years, for allowing me a generous amount of his time, and for his financial support. I would also like to thank Dr. John Feck for his helpful suggestions for improving this thesis. I am grateful to my wife Evelyn for her patience and for her help in typing and formatting this manuscript.
Chapter 1

INTRODUCTION

In the field of computer graphics the design of a Graphics Standard is currently receiving much attention. An attempt is made to find a unifying concept for currently available graphics hardware to facilitate program portability, device independence and "programmer portability". Various proposals have been put forward as a Graphics Standard; examples are the German GKS system, the English GINO-P package, and the Core System proposed by the Graphics Standards Planning Committee (GSPC) of the Special Interest Group SIGGRAPH of the ACM.

This thesis is an investigation of how a mapping can be achieved from a high-level graphics language to the functions of the Core System (GSPC79). The Core System was designed as a subroutine package to act as an interface between an application program and graphics hardware; its output functions provide application-independent facilities to display arbitrary views of two and three-dimensional graphical objects; its input functions support the interaction between the user and an application program. The high-level language that was
chosen is LIG ("Language for Interactive Graphics"); this language is an extension to existing high-level languages which are used as "host languages". LIG was developed at the University of British Columbia and uses FORTRAN as its host language [Schr78a]. It has been used extensively as a research and teaching tool and its basic concepts have been proven successful in terms of ease of use and "naturalness". LIG has also found applications in areas such as circuit design and electronic music composition. In the Department of Electrical Engineering it is used for LSI mask generation and editing. Part of the current thesis is the implementation of LIG as an extension to Pascal, and the design of a large number of new features for the language in areas where improvements could be made. This new version of the language is called LIG/P and supports three-dimensional graphics in contrast to the 2D-framework of LIG.

LIG/P is a tool to construct hierarchical models of graphical objects; it supports multi-level naming and modification capabilities. Graphical objects are defined in terms of graphical primitives, graphical functions, and other graphical objects; the language treats the data type GRAPHICAL as a type in its own right and supplies high-level operators to manipulate graphical data. The major emphasis in the current work is on the design of an appropriate data structure for the mapping of LIG/P's multi-level environment to the Core System's single-level environment. (Conceptual levels are
discussed in detail in Section 2.6.1.) The LIG/P data base stores information about the user-defined graphical model; its display functions traverse the data structures and generate calls to Core System functions. The Core System is based on the concept of one-level segments that are used for display purposes only; no modelling functions are provided at that level. The mapping puts emphasis on the efficient use of Core segments in terms of creation and deletion to minimize the "segment traffic" inside the Core System. The current project is an investigation of how useful the Core System is for the automatic generation of invocations of its functions. Improvements are suggested where ease of use and efficiency can be increased.

For the implementation of the LIG/P preprocessor a compiler writing system obtained from the University of Montreal was used [Lecar77]. This system accepts a formal definition of the language and associated semantic actions for each language construct; a compiler is generated in Pascal. The runtime library for LIG/P was also written in Pascal, so was the subset of the Core System that was required for this project. The output devices supported in the current implementation are a Tektronix 4027 colour graphics terminal, a Tektronix 4014 storage tube and CalComp plotters. The LIG/P system is portable to any installation supporting Pascal and the proposed Graphics Standard which is by now commercially available.

The three major parts of this thesis are a discussion of
concepts that led to the design of LIG/P, a language manual for LIG/P, and a description of the data structures required to interface LIG/P and the Core System.
Chapter 2

COMPUTER GRAPHICS SOFTWARE AND HARDWARE

In a sense computer graphics came into existence with John von Neumann's idea to connect a cathode ray tube to a computer as an output device [Taub63]. Another milestone was the design of SKETCHPAD by Ivan Sutherland in 1963 [Suth63] which broke the ice for interactive graphics. Since then the interest in computer graphics has continued and made it a special field of research in computer science. Information processing using pictorial data has found many applications; these range from passive plotting, grey scale picture processing and documenting systems to three-dimensional interactive design and synthetic picture generation, including shading and hidden surface techniques. Within computer graphics major developments have taken place with respect to hardware design, input and output devices, programming languages and algorithms. Techniques have advanced to a stage where colour graphics is accessible by anyone even on inexpensive but rather powerful and fast personal computers.
2.1 High Level Graphics Languages

As in other areas of programming language design, so far no general-purpose graphics language has been accepted as a "universal graphics language". The applications for computer graphics are too varied with many conceptual differences between them; the data manipulation for geographic data, for example, is very different from that required for three-dimensional computer-aided design, which again is very different from artistic design. Another factor is that in past years computer graphics has split into two separate fields: generative computer graphics, and cognitive graphics or image analysis. So far no unifying concept for the two has been found.

The discussion in this thesis is about interactive generative graphics, and in particular concerned with concepts for computer-aided design with three-dimensional data representation. This section gives an overview of what properties high-level graphics languages should possess and what implementations exist. The survey is by no means exhaustive but most of the better-known systems are mentioned.

One of the important early papers on high-level graphics languages was published by Kulsrud in 1968 [Kuls68]; it presents general concepts for high-level graphics programming in a rigorous and systematic way. Kulsrud divides graphical functions into four major areas and proposes sets of commands for data description, manipulation, regional analysis, and
topological analysis. It is interesting to note that the language does not have commands such as MOVE and DRAW which actually are low level display processor instructions.

Kulsrud's language defines, in addition to common data types, the five graphical data types: point, line segment, picture, subelement, and region.

Increasing interest in computer graphics and subsequent work in the area has brought forward a clear idea of desirable features for graphics languages. It has long been recognized that a graphics system should provide more convenience than a simple subroutine package can. A programmer should be able to express algorithms in as natural a way as possible and as close to his problem as possible. The language must provide facilities for modular design and convenient and readable control structures. Of particular importance are appropriate data types, and these are only useful if corresponding operators are defined. Existing graphics systems provide either implicitly defined data types such as arrays in APL, or explicitly declared ones, such as the type GRAPHICAL in LIG; the latter is preferable with respect to writing an algorithm in terms of the problem. A graphics language must provide primitive data objects from which more complex objects can be built, and possibly also some often-used "advanced primitives" such as arcs and polygons. Of particular importance is also text-handling since almost all graphic representations require the inclusion of text.
Graphic input and output is of a different nature than that of programs for other applications and the design of i/o statements must receive particular attention. Graphics i/o involves special hardware and only very recent efforts succeeded in finding abstract concepts for these special purpose devices. Wallace [Wall76] published one of the first papers that defines virtual input devices with the aim of achieving "program portability, human factors adaptability, economy and maximal use of terminal capability". The same topic for output primitives is addressed in a paper by Bergeron [Berge76]. Most of these ideas were incorporated in the definition of the proposed Graphics Standard of the GSPC. These ideas search for computer-independence for graphics systems; not only should graphics application programs be portable but also the run-time environment should be designed in such a way that it can easily be taken from one machine to another, with small changes only in the device dependent code and the routines interacting with the operating system. If a high-level language is chosen for the implementation, many of these problems are already partially solved.

A graphics language must be able to allow a certain amount of file handling to facilitate continuity between sessions. User-defined graphical objects can then be saved for later access or to create a library of often-used graphical building blocks for certain applications.
A language must have the ability of imposing a structure on graphical objects; these objects often have to be manipulated as a whole. They are highly dynamic entities (e.g. in computer-aided design) which are assigned a number of attributes (e.g. colour, line style, text size). Often there also exist some interrelationships between graphical objects, in particular if simpler ones are used to build more complex ones. The definition of such relationships must be supported by clear concepts in the high-level language and must adapt in a natural manner to the way in which graphic representations are perceived by the user of the system. Graphical data bases have complex interconnections between data items; often they are ring structures, linked lists, trees, or even combinations of these.

In most applications graphical objects are frequently transformed with respect to size, position and rotation. Transformations can be applied in the modelling stage and later again when an object is displayed. These are called modelling transformations and image transformations respectively and must be incorporated in a high-level language such as to make their use consistent and obey the "principle of least surprise". Functions of the underlying graphics system must automatically classify and distinguish between three kinds of data structures as described by Posdamer [Posd77]: the "topology structure" which defines the problem or the environment, the "computation structure" to store data and
The most commonly used graphics systems are either of two kinds: pure subroutine packages or high-level language extensions. Subroutine packages have the advantage of portability but do not allow the use of data types and operators. Language extensions require an extended compiler or a preprocessor; some existing programming languages are more suitable than others for an extension and are therefore called "extensible". An example is Algol 68 [Den75] which allows operations with programmer-defined types and operators. In general one distinguishes between two groups of graphics languages: there are those for large scientific programs and those for conversational use, e.g. picture editing.

The earliest language in the first category was an extension to Algol 60. Algol 60 is very clean in its program structuring facilities, but rather inconvenient for data manipulation (e.g. no string handling facilities, only arrays to build data structures). The graphics extensions to Algol 60 were LEAP, written in 1968 [Rovn68, Rovn69], and SAIL, written in 1971 [Swi71]. Attempts to improve on Algol 60's deficiencies were made in PL/1 which uses many of the concepts defined for Algol 60. A derivative of PL/1 is XPL with its extension XPLG [Tur75]. The most commonly used language as a basis for
graphics systems is FORTRAN; the language is available on almost every computer and a large variety of special purpose routines exist. For scientific computing it is also considered run-time efficient. Graphics extensions for FORTRAN are too many to be enumerated here; examples are EXGRAF [Wil72], LIG [Schr76, Schr78a, Schr78b], and GRIP [Gil78]. A very elegant language is EULEB, designed by Wirth, for which a powerful and convenient graphics extension called EULER-G [Newm71] was designed. EULER-G and XPLG were the first languages that implemented display procedures. Some experiments have also been carried out to extend BCPL for graphics [Rich76].

Languages for conversational use that were extended for graphics include CPL [Bar63], EASIC [Sharp70], APL [Fosd73, Stil77], and LISP [Jon78]. APL provides convenient means for handling matrices, which is essential for geometric transformations; it also allows definition of functions which makes statements such as "PLACE SPHERE AT (3,4,5)" a legal APL statement [Fosd77].

Future developments for graphics languages will be strongly influenced by developments of graphics hardware. Like elsewhere in the field of computer science, the advent of microcomputers will have a major impact; in addition to this the constraints of limited and slow memory are removed. New hardware systems will allow implementation of concurrent processing and through language constructs the programmer will have control over timing of evaluation. Data models will in-
include faceted polygons and patch elements that are used to compose free-form surfaces. Various ideas exist on how three-dimensional input and output can be achieved [Ries78]; the programs for such systems will eventually be written in high-level languages.

2.2 LIG, Concepts and Improvements

The particular high-level language studied more closely in this thesis and used to implement some of the demands made in Section 2.1 is LIG ("Language for Interactive Graphics"). It was originally designed in 1972 at the University of British Columbia; since then many improvements have been made to the language itself and its run-time environment, and it has been used for many years by graduate and undergraduate students. For two years now the language has been taught in an introductory graphics course [Schr78a].

LIG is an extension to FORTRAN, and although it is defined as a language by its own it takes advantage of FORTRAN's numerical, boolean and algorithmic capabilities. A preprocessor translates a LIG program into an intermediate FORTRAN program in which the original graphics statements appear as subroutine calls. The supporting LIG subprogram system, written in FORTRAN for consistency, handles graphical input, output and data base manipulation.

LIG's data type GRAPHICAL can represent the kind of in-
formation which two-dimensional line drawings are composed of. LIG uses the areal external representation model for the data type GRAPHICAL (the linear external representation is used for example by the languages PDL [Mill68], E-LINE [Fra68] and TUNA [Berk77]). This data type is treated consistently as a new type in its own right; thus, language elements such as graphical variables, graphical operators and graphical assignment statements correspond to equivalent elements of other high-level languages. The LIG graphics system supplies six graphical primitives and allows inclusion of user-defined graphical functions. There are six graphical operators in LIG; the syntax for using these is similar to that of EULER-G.

A special kind of graphical assignment is used which accounts for storage space efficiency and also implies a high interdependence between graphical data. The data base consists of a tree structure. The language characteristics are formally defined by a syntax in BNF. The preprocessor is generated by the XPL Translator Writing System by McKeeman et al. At the University of British Columbia LIG is implemented on two computer systems. The preprocessor runs on the computing centre's AMDAHL 470; the resulting FORTRAN programs can be run either on the same machine or may be transferred to a Data General NOVA 840 minicomputer with a Tektronix 4010 graphics terminal in the Department of Electrical Engineering.

LIG's original design goals of simplicity and "naturalness" of use have been met; however, new developments
in computer graphics demand an ongoing revision of the language. Chapter 3 describes in detail the proposed new syntax for the language. An overview of where new language constructs were felt to be necessary is given here:

1. FORTRAN as a host language was considered outdated and it is investigated how Pascal can be used instead. Existing and new language constructs are specified with this in mind. The new dialect of the language is called LIG/P.

2. LIG/P is a 3D-system in contrast to LIG's 2D-facilities.

3. The entire LIG/P system is written in Pascal; this includes the preprocessor and the run-time environment. The original data base has been redesigned; extensive changes were required to accommodate new features.

4. New data types were introduced; this fits well into the framework of Pascal. The types SUBPICTURE, INSTANCE, INPUT device, OUTPUT device, VIEWVOLUME and VIEWPORT have been added to the original data type GRAPHICAL. The concept of graphical primitives and system defined functions was modified.
5. A much improved concept of modelling and display functions is introduced. This includes the availability of a program defined world coordinate system, of clipping, of viewing and image transformations.

6. Multiple i/o streams combined with viewing operations allow mapping from view volumes into viewports on different devices, or multiple viewports on the same device. The selective erase functions have been redesigned.

7. Device enabling, setting and enquiry functions make the language adaptable to sophisticated hardware environments.

8. Graphical input functions have been redesigned. Text input, value input and stroke input have been included.

9. New identification constructs allow logical operators and expressions in the IF HIT ON... statement, and provide a CASE HITON OF... statement.

10. Attributes for graphical variable definition and display were introduced. Attributes such as colour, line style and polygon fill did not exist before in the language.
11. A number of new control statements were added; e.g. a PAUSE and BELL statement. Control over the physical size of hardcopy output is possible.

12. An archivation statement allows the permanent storage of graphical data. SAVE and RESTORE statements facilitate continuity between sessions.

2.3 Portability of Software using the Virtual Machine Concept

A programming language is defined with a certain class of problems in mind; these are to be solved conveniently using the control and data structures provided by the language. The question arises of how the language is to be implemented. For an implementation the high-level language constructs have to be broken into simpler configurations; this is a stepwise process until the result is a series of machine code instructions. Within this process one can distinguish between machine-independent and machine-dependent phases and codes. The problem of software portability has led to the development of language translators whose target code is an assembly language for a simple virtual machine; this abstract machine can be mapped onto a real machine and from the virtual machine code one can generate code for the actual machine. The two main advantages of this method are:
1. The language translation process is split into two simpler tasks — generation of virtual machine code, and generation of actual machine code; if the interface is clearly defined then the two processes are independent of each other.

2. As already mentioned, portability of a compiler can be achieved easily because an implementation of a compiler on a new machine requires that only the second phase has to be rewritten; this phase is the simpler of the two and the corresponding translator is relatively easy to write.

Various projects involving different high-level languages using this principle have provided valuable insight into the design of abstract machine codes. This section briefly mentions and explains objectives of such projects; Section 2.4 is an investigation of how graphics machines are designed, and Section 2.6 describes the design principles of an abstract graphics i/c device.

Code generation for high-level language programs can be considered effective if the output produced by a language translator is an efficient and compact representation of the original language constructs. The characteristics of the generated code reflect the architecture of the target machine. How machines are being used is of concern to system archi-
tects, compiler writers and language designers; among them many hold the opinion that existing machines do not facilitate effective representation and execution of high-level language constructs. Machine architects should view the computer as a tool to efficiently run the programs that people write for them. An important contribution to support this idea were the experiments by Tanenbaum [Tan78] who proposes a stack machine architecture to execute programs written in a block-structured high level language. Tanenbaum bases his work on earlier studies by Knuth [Knu71], Alexander [Alex75], Salvadori, Gordon and Capstick [Salv75], and Wortman [Wort72]. Wortman states: "We took as our design goal the development of new design goals to aid the designer in building computers that satisfied the actual rather than the imagined needs of the programmer."

The design of computers with high-level languages in mind is closely related to microprogramming. Microprogramming is in fact one way of implementing intermediate languages. The others are direct translation to machine code, as already mentioned, or interpretation. There are two classes of intermediate languages; they can be "machine specific" or "language specific" [Els79]. Machine specific intermediate languages are taylored to suit a particular target machine, and all compilers which translate programs for that machine are required to translate high-level language programs into that particular intermediate language. This approach has not proven too suc-
cessful because of the difficulty of finding suitable intermediate languages for different high-level languages. The intermediate language must in this case be at a relatively high level to allow common intermediate constructs for high-level languages, and much of the processing has to be done in the second phase of the translation. Language specific intermediate languages have shown more success. In this case a high-level language is translated into its own intermediate language that is particularly suitable for the representation of its language specific constructs. Thus the greater part of the translation process is done in this phase. All that remains to be done is to translate the simple intermediate language into target machine code. The first phase of translation is made entirely machine independent. By writing the compiler in the particular high-level language itself it is possible to bootstrap it when porting it to a new machine.

Examples of successful applications of intermediate language techniques are BCPL (translated to OCODE and INTCODE [Rich74], MINICODE [Peck75], SLIM [Fox78]), ALGOL 68 (translated to ZCODE [Bour77], ACODE [Els79]), Pascal (translated to E-code [Ncri74]), COBOL [Brown77], and SNOBOL [Dew77].

The abstract machines defined for a language can have instructions of a relatively high level, e.g. block structuring instructions, procedure calls, case statements; or they can be closer to a particular machine language and thus oriented towards a particular class of real machines.
A lower level intermediate language with very simple instructions can more easily be implemented on most machines, but it also has disadvantages. In most cases the target code is inefficient, and if a machine provides higher level constructs one cannot take advantage of these. It is also a longer translation process to generate low level intermediate code which may in practice make the compilation too inefficient. For some languages, such as Algol 68 and Pascal, some machine specific information (e.g., data sizes and register availability) is required to obtain reasonably efficient code; such data can be fed to the compiler in advance; the machine dependence is then "parameterized" and is not part of the intermediate language itself. If commitment to any real machine is to be avoided then no high-level primitives may be decomposed, e.g., the structure of a program may not be lost.

It depends on the particular project and the characteristics (e.g., size) of a high-level language which level of intermediate language should be chosen. A low level code with parameterized information is appropriate if one quickly requires a tolerable implementation on most machines and a good implementation on few machines; on the other hand, if the objective is a good implementation on most machines then a higher level machine independent and language-oriented intermediate language should be chosen, at the expense of a more difficult translation to machine code.
In the context of this thesis these results are interesting for the following reasons:

1. To our knowledge no intermediate language has ever been designed specifically for a high-level graphics language. The investigation of a machine architecture for LIG programs would be an interesting task. Since LIG/P is an extension to Pascal an intermediate language could be an extended P-code. In Chapter 4 of this thesis it is described in more detail what actions have to be taken for high-level language statements; this is a further step towards the definition of a virtual LIG-machine.

2. In this thesis only the i/o process of graphical data makes use of an already defined virtual graphics machine. It is investigated how suitable this relatively low level graphics machine is as part of the complete LIG/P graphics system.

3. The definition of graphical objects involves to some extent the encoding of such information for compactness of storage; the conclusions drawn above provide valuable guidelines for the design of certain parts of the LIG/P data base.
2.4 Graphics Display Processors

To achieve maximum speed for the execution of graphics functions so-called "intelligent" terminals have been built with a large amount of local processing power. Several approaches for building such graphics devices will be discussed.

2.4.1 Host/Satellite Configurations

In recent years much research has been done to build graphics systems in which a host computer is connected to several graphics satellites. In general the host is a large time-shared machine and the satellites are small mini- or micro-computers. The advantage of such a system is that the host does not spend any time with trivial but time-consuming graphics support, yet the user enjoys real-time response at the graphics terminal. The host can allocate its resources more effectively on application program processing and database functions. Another goal is to reduce communication traffic between host and satellites as much as possible, especially if the connecting link is, for example, a low-speed telephone line.

Two kinds of graphics satellites exist. "Fixed-function satellites" are accessed from the host computer via subroutine calls; in this case the application program is completely host-resident. When "programmable satellites" are used the connection between host and satellites is more relaxed. Some
of these satellites have their own secondary memory and store a complete graphical package; additional functions can be implemented and even parts of the application program can be processed there.

It is obvious that such a system provides superior facilities but is also much more complex in terms of overall operating systems functions. Considerable effort has been expended to investigate solutions to the "division of labour" problem. These are general problems of distributed processing and inter-process communication. Two interesting references addressing this topic are [Mou176] and [Cul179].

The following two subsections 2.4.2 and 2.4.3 describe in more detail what capabilities have been considered important for high-performance graphics satellites and how they have been implemented.

2.4.2 Hardware of Display Processors

High-performance display processors are special graphics computers which need minimal support from a host computer to display pictures. They are stand-alone systems with special hardware designed for applying transformations to picture definitions and thus being able to display flicker-free dynamic pictures in real time.

The components of a typical system are a transformation processor and a picture generator; the transformation proc-
essor has an interface to a CPU which sends it transformation parameters and drawing commands. Other instructions set certain modes in the transformation processor, e.g., how coordinate points are to be connected, whether coordinate data is two-dimensional, three-dimensional or homogeneous. The transformation processor may have a DMA link to the CPU in which case it can fetch coordinate data while the CPU can execute other code, e.g., optimize the organization of coordinate data to speed up the generation of a picture frame. An example of such a system is THE PICTURE SYSTEM by Evans and Sutherland Computer Corporation [Call75], a refresh vector display.

The transformation processor operates on the coordinate data a point at a time and places the result in the refresh buffer; the data in the refresh buffer are read by the picture generator, converted from their digital form to analog signals and output on the display surface. The steps taken by the transformation processor are conversion of point coordinates to absolute homogeneous coordinates, multiplication by the transformation matrix, scaling, rotating and translating, clipping to a window or view volume to remove parts that should not be visible, applying depth-cued perspective, and finally mapping onto viewport coordinates. THE PICTURE SYSTEM requires between 10 to 36 microseconds to apply these operations on one point; thus on the average it can process about 3300 points in 1/15 s, the maximum frame time for smooth dynamic output. Picture update in the refresh buffer and pic-
tured display do not occur at the same rate; they are independent of each other. The minimum refresh rate for flicker-free pictures is 30 Hz; some picture generators can refresh at 40 and 60 Hz. The screen resolution of such systems is normally 4096 x 4096 addressable points; display rates vary between 1600 and 26000 ten-inch lines for a 30 Hz refresh rate, depending on the manufacturer. To ensure smooth movements in dynamic displays the refresh buffer may be split into two sections and the technique of double-buffering is used. This can be done if frames fit into half the size of the refresh buffer and a displayed frame in one section remains intact until a new frame has been generated in the other section; then the refresh sections are switched.

Slow software-performed picture transformations such as shading and hidden-line removal are normally done by the CPU. In some systems the CPU can obtain information from the transformation processor which normally outputs only to the refresh buffer, and perform the time-consuming picture transformations before sending the data back to the transformation processor.

Some systems, such as THE PICTURE SYSTEM, have no options within the central processing path. Others are general-purpose expandable processors where critical display functions are hardwired, and autonomous asynchronously running processors for special functions can be added externally. Several such processors have been built; one at the North Carolina State University [Staud75] for example, has a very fast proc-
ess to implement operations such as specific arithmetic, linked list handling, fast output, Watkins' Clippers, and digital array multiplication. It is part of a colour scan-display system and has a worst-case instruction execution time of 80 ns. It uses plug-in processors for special arithmetic; one of the peripherals is a fast multiplier with a multiply time between 24 and 180 ns (for numbers between 4 and 32 bits). Another peripheral which communicates through the i/o bus provides general trigonometric functions and colour modifications for rotated objects. Perspective projection and matrix multiplication for rotation, using homogeneous coordinates, is done by a peripheral which accepts 16-bit \( x, y, z \) and \( w \) (distance from the observer) coordinates and returns the result in 2 microseconds. Raster scan output is provided by a colour display system which receives line and run-length encoded data from the main processor.

2.4.3 Microprogrammed Display Processors

Advantages of microprogrammable graphics processors are lower cost, greater flexibility, better maintainability through firmware diagnostic routines, and of course superior performance compared to the substituted software graphics functions. Two implementations of such processors are described here.

A research project at Brown University [vanDam72] inves-
tigated design tradeoffs between hardware, firmware and software. The goal was to build a medium-cost interactive design system with quick response time and smooth picture transformations, e.g. zooming. The system can operate either in stand-alone mode for small problems, or as an intelligent satellite for a host that handles the picture structuring and data base management. Hardware components of the system are two Meta 4 microprogrammable processors, a mini disk, a special arithmetic logic unit, and a Vector General display. The first Meta 4 processor is used as a small satellite computer which handles the interface to the host machine, local peripherals (e.g. a typewriter console and the disk for large user display files), and it controls the second Meta 4. That machine is the "display controller" and is microprogrammed to execute instructions in the display file; it performs 3D-transformations and clipping and also processes graphics device interrupts.

To be able to process and display at least 1000 to 2000 vectors per frame, another microprocessor is added; this one is microprogrammed to execute vector-matrix multiplication, scalar-vector multiplication, and perspective calculations. It is a very fast parallel processing computer with an instruction time of about 30 ns.

The Vector General display processor generates vectors and characters, performs z-axis depth cueing and viewport mapping.
An example of a commercially available microprogrammed processor is the Adage GP/400 interactive graphics system [Kerr75]. The design objectives were speed improvement by being able to reduce the overhead of graphics commands, being able to handle a high degree of image structure, and flexibility for different applications through programmability.

To increase speed, coordinate-transformation hardware performs vector multiplication and transformation concatenation. Microprograms have replaced sine and cosine calculations which are performed in about 5.6 microseconds compared to about 100 microseconds when executed by software. Overall speed improvements average 10 to 30 times over the execution of functions by software.

2.5 Achieving Device Independence

Analogous to the "kernel" concept for operating systems, graphics systems can be organized in a hierarchy of levels; each level has a well-defined interface and higher levels can access lower ones directly or through intermediary levels. Figure 1 illustrates the arrangement.

This configuration is a step towards achieving compatibility with other graphics software and machines. Each layer is a higher abstraction of the lower ones and provides additional features. A programmer is provided with the conceptual model of a "pseudo-machine" and writes his application in
application software

application-oriented graphics support

application-independent modelling functions
(see graphical expression evaluation, high-level language dependent)

display functions
(viewing transformations, image transformations)

special operations
(e.g. hidden line removal)

function library or graphics package
(e.g. segmentation of display files)

output primitives

input primitives

device driver

graphical output device
firm/hardware

device driver

graphical input device
firm/hardware

Figure 1

Those terms. Many such conceptual models exist, and much effort has recently gone into defining a standard abstract model for graphics; one of the major decisions was to decide on an appropriate level of abstraction.

Several solutions to achieve device-independence at lower level interfaces have been proposed and implemented. Two are briefly described here: one deals with the interface to the device driver, and the other is one level higher.

An example for the first case is HPGL ("Hewlett Packard Graphic Language") [Dan77]. HPGL is a command language for plotters, each command consisting of two ASCII characters, followed by numeric parameters and ended by a terminator character. Calculators or desk top computers send instruc-
tions to the HP Graphics Translator which translates them by firmware into ASCII-coded HPGL commands and sends them to the plotter. The plotter's firmware interprets HPGL. The language is expandable and new commands are added as required by new products.

An example of a language used to interface at the next higher level is L4 ("Linear Low-Level Language") [Gil78]. This is an intermediate language to serve for the communication between a host computer and its satellites. To minimize the data flow, the language itself is as concise as possible. Also, only picture editing information is exchanged, but not entire display files -- a display file is generated only once, and since picture editing can occur at either the host or the satellite, L4 carries information both ways. The satellite must perform L4 interpretation and display file management, object identification, accept input from input devices (the alphanumeric keyboard and the function keyboard), and handle certain interactions locally.

Considering the examples above and those in Section 2.4 it may be clear now that the levels of a graphics system as shown in Figure 1 are not clearly defined for any given implementation. There is a great dependency on the particular hardware available and on the operating system. Trends in hardware developments suggest that possibly a more linear representation (or "pipelining") of interacting parts is more adequate in the higher levels. This would better incorporate
new plug-in processors or programmable satellites with their local features such as surface-generating functions, hidden-line removal or display of shaded 3D-objects.

With respect to intermediate codes a conclusion similar to that mentioned in the discussion of high level languages has been reached; the device driver (or code generator) of a particular device has to make the decision about facilities available in software, firmware or hardware. For functions that are available in hardware no software overhead should slow down the performance. A device driver must, however, have access to device independent simulation software for functions that cannot be provided in hard or firmware; this software can be used by all device drivers and may for example contain stroke precision character definitions. To summarize what was said before, an intermediate code must therefore consist of relatively high-level commands so that good implementations are possible for most devices. It depends on the level of abstraction what form exactly they should have.

2.6 Design Criteria for Graphics Packages

The purpose of this section is to relate the Core System to established concepts and to other graphics packages and their capabilities. To present the Core System as a standard graphics package it has to be put into the right perspective by viewing it in the context of a wider range of design cri-
2.6.1 Levels of Picture Naming and Modification

Graphics languages and graphics subroutine packages can be categorized by the kind of organization they allow on graphical data. Some systems provide facilities for multiple levels or a hierarchical structure of defining graphical objects, others have a much simpler concept and permit only a single level. In most systems the various parts forming an object can be grouped together in a "segment". A segment has a unique name and has attributes which can be changed; these changes apply collectively to all parts contained in the segment. Identification and deletion also apply to an entire segment.

In a single-level system a segment contains only output primitives, and all segments are independent of each other with respect to modifications and naming. In a multiple-level organization references between segments are allowed and a particular segment can use another one as a subpicture. By referencing a particular segment name all lower level segments referred to by the addressed one are automatically included. It is obvious that a multi-level naming capability is more useful to serve a variety of applications than a single-level one.

Levels of naming are important in the way they are tied
to levels of modification because the two need not be identical. Modifications are attribute changes and addition or deletion of parts. With a zero-level modification capability, a picture of which a part is to be deleted has to be erased entirely and then be redrawn without the unwanted part. A single-level system has one level of segments and no hierarchical structure at all. A picture can consist of a number of displayed segments which can selectively be erased. If part of a segment is to be erased, the entire segment has to be redefined without the deleted part and is then redisplayed. Multi-level modifications are applied at some point in the hierarchy, or at some internal tree node, and apply to the structure underneath in terms of transformations, addition and deletion of parts; no redefinition of any component is required.

As mentioned above, modification and naming levels need not be identical. Foley [Fol76] states that "it is desirable to organize the naming and structuring capabilities so their use is completely optional: the application programmer should need to understand and use the capabilities only if he truly needs them. Otherwise the language becomes a burden, rather than an aid, to its user".

The high-level graphics language LIG fulfills this requirement. Picture structuring can be done to any desired level, and picture naming can optionally occur at any point in the hierarchy. This is the case for the picture model or in-
ternal representation, as well as the display structure to facilitate selective erasing of instances.

2.6.2 Common Graphics Packages

The better-known graphics packages include DISSPLA (Integrated Software Systems Corp., San Diego, written in FORTRAN), TCS/PL0T-10 (Tektronix Terminal Control System, written in FORTRAN), GFGS (Katholieke Universiteit, Nijmegen, written in FORTRAN and assembler), Omnigraph (XEROX P.A.R.C.), GINO-F (Cambridge England, written in FORTRAN), GCS (U.S. Military Academy, written in FORTRAN), DIGS (University of North Carolina), GKS (German Graphics Kernel System), IG (University of Michigan, written in IBM 370 assembly language), and the GSPC Core System (many implementations, most of them in FORTRAN). A more detailed description of each can be found in [Fcl76] and [Ewal78].

All of the above systems provide facilities to group output primitives in segments; they vary widely, however, with respect to the operations possible on segments. They are different in the way naming and modification levels are allowed; access to segments when defining graphical objects is also treated differently. Here is a list of possible actions on segments, of which each package provides a larger or smaller subset: segments can be created, named, renamed, deleted, referenced, extended, transformed, redefined, copied, and have
attributes. The usefulness and convenience of a graphics package depends to a large extent on the number of these functions it provides. The more functions are missing the more work is left to the application programmer and the more must he define his own concept of graphical data and model it with his own data structures and operations.

2.6.3 Standardization of Graphics Capabilities

Advantages of the standardization of graphics functions are device independence, computer independence and programmer efficiency. A disadvantage is the inability to take advantage of specific architectural characteristics of a particular processor. A graphics standard with too few features cannot use effectively such high-performance display processors as described in Section 2.4. On the other hand, if there are too many features it becomes bulky and expensive to implement and use. A compromise is a balanced number of commonly required features which can be implemented by hardware or simulated by software. Standardized interfaces can be defined at various levels in a graphics system (referring back to the diagram in Section 2.5). At the highest level an interface is application oriented, and some commonly-used examples are ICES ("Integrated Civil Engineering System") [Boes68], and SYMAP (a statistical mapping system) [Symap]. These are more "de facto" standards, though, which rely on standardized lower
level support.

A few levels of interfaces further down are the graphics packages. Among these one distinguishes between "application-independent, high-level graphics support" and "basic graphics support" [PCL79]. Examples of the first group are DISSPLA, GPGS, GINO-F and GCS; all of them are device- and computer-independent but vary in the kind of facilities they offer and towards which application areas they are biased, e.g. data plotting, cartographic projections, or hierarchical modelling. The proposed GSPC standard belongs to the "basic graphics support" group, as does Omnigraph, which offers less than the above packages; this was felt to be appropriate because it is unlikely that a consensus can be reached for the inclusion of higher level functions, and if all of them are included the result is an enormous system of which relatively few functions will be used by most applications.

2.6.4 Functions of the Core System

The relatively low level of the Core System seems to have some advantages; it is based on the compromise of "what is good for most programmers on most existing interactive displays most of the time" [Newm78]. It is considered functionally complete such that higher-level modules can easily be built on top of it; all functions must be present which cannot be substituted by or built from other Core functions. It is
primarily oriented towards medium-performance interactive vector displays and excludes features that are not yet accepted practice — "a standard will not be accepted if it is too far ahead of the state-of-the-art" [Newm78].

A brief overview of the Core System is given here for completeness; detailed discussions can be found in [GSPC79] and [CS78]. Grouping, naming and modification of 2D- or 3D-output primitives is done with segments; there are temporary and retained segments. Retained segments can be created, closed, deleted, renamed, and have their attributes modified. Segment attributes are either dynamic or static. Dynamic segment attributes are visibility, highlighting, detectability, and image transformations. These can be modified dynamically after the segment is defined and the effect will be visible immediately. Static segment attributes are image-transformation-type and view-surface-selection. Segments cannot contain references to other segments — the Core System is a single level system. Segments cannot be extended and cannot be copied. Why these decisions were made and how one can program without these functions is discussed in [Mich78].

Output primitives are moves, lines, polylines, text, markers and polymarkers. There are many attributes for output primitives, for example colour, intensity, line style, character size and text quality. Attributes for output primitives are called "static" because they are assigned once and cannot be changed for the life-time of an output primitive.
Input primitives are classified as event-generating pseudo-devices and sampled pseudo-devices. The first group consists of the devices PICK, KEYBOARD, BUTTON and STROKE, the second group of LOCATOR and VALUATOR. Input can be synchronous or asynchronous; the latter generates event reports and places them into an event queue. Sampled devices can be associated with event devices such that an event report from some event device contains data from the associated sampled device.

The Core System provides functions for modelling transformations, viewing transformations and image transformations. The application program defines segments in a world coordinate space; modelling transformations are applied and all coordinates are mapped into NDC (Normalized Device Coordinate) space. Viewing transformations are clipping, projection, and geometric transformations for mapping from a programmer-defined window onto a viewport. Image transformations are geometric transformations applied to a visible segment on all output surfaces associated with it. A device driver transforms Normalized Device Coordinates to physical device coordinates when a segment is displayed on a particular output device.

The Core System provides extensive inquiry functions, setting of initial values for devices and attribute values, and selecting display devices for output.

The Core System can be implemented in several upward com-
compatible levels for input and output and two levels for dimensions. This allows for adjustment to the requirements of a particular installation which can range from simple plotting to interactive graphics with refresh displays.

An important decision for the design criteria of the Core System was the complete separation of modelling and viewing functions. The Core System considers modelling the responsibility of the application program; when a picture is generated from a model, its data base is traversed and geometric transformations are applied to the graphic primitives it contains. Some graphics systems provide general-purpose transformation functions, some provide in addition facilities for model construction and traversal. However, every type of application requires a different set of model traversal functions and it was decided to completely exclude these from the Core System. It is purely a viewing system which reproduces images of the application program's "world" on the output surface. The display file is a highly temporal entity and is not intended to permanently store any picture description generated within the package.
Chapter 3

LIG/P LANGUAGE REFERENCE MANUAL

3.1 Introduction

The high-level graphics language LIG/P is a graphics extension to Pascal. It is intended for the definition of three-dimensional objects which are defined in terms of lines and filled polygon areas. High level operators allow the manipulation of graphical objects at the modelling and at the display level. Modifications provided for graphical objects are geometric transformations and attribute changes, such as colour and line type.

A LIG/P program is translated into pure Pascal source code by a preprocessor. A second translation phase compiles the Pascal program; when executing it, the LIG/P run-time library creates and manipulates entries in the graphics database.

The LIG/P run-time library was designed such that its input and output functions map onto the conceptual framework of the Core System [GSPC79]. The entire LIG/P system thus
becomes portable and device-independent.

3.2 Terminology, Notation and Vocabulary

The structure of a LIG/P program is essentially the same as that of a Pascal program. LIG/P statements are added to a Pascal program which determines the flow of control. At installations where separate compilations of Pascal procedures are possible, the same can be done for source code that requires preprocessing.

LIG/P introduces additional predefined data types for defining and displaying graphical objects. They are referred to by their name which is an ordinary Pascal identifier. An important distinction must be made between the modelling and display phase manipulation of graphical variables; since LIG/P supports a multi-level modelling system and a single-level display system, certain operations are only allowed on variables of a corresponding type.

To be able to differentiate those parts of the graphical data structure that must be distinguished conceptually, the following terminology is used:

A **graphical variable** is a graphical object which is composed of **subpictures**. Subpictures are either primitives, graphical function yields or other graphical variables. Subpictures are **named** or they are **anonymous**; if they are named, access to them for modification is allowed at a later stage; they can, for
example, be deleted from a graphical variable. Anonymous subpictures become part of the "constant" portion of a graphical variable and cannot be altered unless the graphical variable is redefined altogether. When a graphical variable is defined, its representation in the graphics data base is constructed.

When a graphical variable is displayed, an instance of it is created and output on a view surface. Like subpictures, instances can be named or can be anonymous; if they are named, they can be erased selectively from the view surface; anonymous instances are only erased when the entire view surface is erased, or when all instances of a graphical variable are erased.

The operations that are allowed on subpictures are a small subset of those applicable to graphical variables. Named subpictures must be distinguished from graphical variables even syntactically, although they appear superficially to be the same. Experience has shown that programming errors are frequently made in the use of named subpictures, and by making them different entities clarifies the concept and leaves some error detection and possibly error correction to the LIG/P system. In the implementation of LIG/P, they are syntactically distinguished.

The LIG/P vocabulary consists of special symbols, such as operators and delimiters, and a number of reserved words. Reserved words are always written in upper case and may not be
used as identifiers. Comments in the source text are enclosed in braces ("{" and "}")..

The LIG/P special symbols are:

```
{ . -> , := :-
++ -- <= ( ) :-
() [ ( . ) ] , +
- * / ... }
```

The reserved words are:

```
ABD AS AT BEGIN
ELANK BUTTON CASE CSIZE
CSPACE COL CUESOR DEG
DELTA DEVICE DISPLAY DIV
ELSE ENDErase FILL
FROM GRAPHICAL GRINPUT GROUTPUT
HARDCOPY HIGHLIGHT HIT HTON
IF INITIALIZE INSTANCE INTEH
INTO IS LIG LIGHTN
LINE LTYPE LWIDTH MOD
NOT OF ON OR
FATTEEN PERSPECTIVE POLYGON PROCEDURES
READ RCT SCL SCREEN
STROKE SUMPCTURE TEXT THEN
TO TYPES VALUE VARS
VIEWPORT VIEWVOLUME WITHIN WORLDCOORD.
```

The LIG/P preprocessor recognizes graphics statements by an asterisk ("*") as the character in column 1; all other source code lines are copied immediately to the output file and are not translated by the preprocessor. Except for this restriction, LIG/P statements can be entered in any format that would also be acceptable to a Pascal compiler. LIG/P and
Pascal statements may not be mixed in the same input line. Most LIG/P statements are translated into invocations of procedures of the graphics run-time system.

The run-time system types, variables and procedures have identifiers with the prefix "lig"; an application programmer can avoid interference with these pre-defined entities if he chooses identifiers not having "lig" as their first characters.

3.3 Program Structure

The general format of a Pascal program, expressed in Backus-Naur Form, is

<program> ::= <program heading> <block>.

The program heading contains declarations for constants, types, variables and external procedures; the items appearing in the program heading define a global environment for all procedures contained in the procedure block. To compile a LIG/P program, the graphics types, global variables and external procedures must be made known to the Pascal compiler. This is achieved by including type, variable and external procedure definitions in the preprocessed source code by the preprocessor. The application programmer must indicate to the preprocessor where to insert those additional declarations.

Thus, to the preprocessor, a LIG/P program has the general format
\[ \text{<LIG/P program> ::= <LIG/P program heading> \} <graphics statements> \}.
\text{<LIG/P program heading> ::= <lig types> \} <lig vars> <lig procedures> \}
\text{<lig types> ::= LIG TYPES;}
\text{<lig vars> ::= LIG VARS;}
\text{<lig procedures> ::= LIG PROCEDURES;}
\]

A complete program heading will therefore look like follows:

```
PROGRAM ligdemo;
  {Example for a program heading}
CONST
  minval = 0.0;  maxval = 100.0;
TYPE
  * LIG TYPES;
    coordinates = RECORD
      x,y,z : REAL
    END;
VAR
  * LIG VARS;
    poly_face : ARRAY[1..100] OF coordinates;
  * LIG PROCEDURES;
    PROCEDURE gen_arc(loc : coordinates); FORWARD;
BEGIN {main routine}
  ...
```

The first LIG/P statement generates pre-defined graphics types; the Pascal statement following it defines the global type "coordinates". The second LIG/P statement generates the LIG/P system global variables, and the Pascal statement following it declares the global array "poly_face".

Before any graphics operations can be performed, the graphics system has to be initialized. This involves establishing a minimal data base and initializing the default input and output devices. Initialization is performed by the state-
INITIALIZE LIG

This statement must be the first to be executed of all graphics statements.

LIG/P statements are separated from each other by semicolons. The last LIG/P statement in a program must be terminated by a semicolon, and the end-of-file is indicated to the preprocessor by a period. This period may either follow the last LIG/P statement immediately, or be inserted on any subsequent line (which has a "*" in column 1).

3.4 Modelling Functions

The modelling functions of LIG/P are invoked by the modelling statements; these allow a programmer to define graphical objects in various ways. The system generates a corresponding data base which is traversed by the display functions to generate visible images on an output device.

3.4.1 Graphics Types

LIG/P introduces additional data types which are pre-defined types to the application program. These types are:

{GRAPHICAL, SUBPICTURE, INSTANCE, GRINPUT, GROUTPUT, VIEWVOLUME, VIEWPORT}.

The first two of this set are used to define graphical objects, the others are used for displaying such objects.
Variables are declared in Pascal statements and follow the same scope rules as all other variables.

Global variables could, for example, be declared in the program heading like this:

```pascal
VAR
  * LIG VARS;
  and_gate : GRAPHICAL;
  chip      : ARRAY[1..10] OF GRAPHICAL;
  element   : ARRAY[1..10] OF SUBPICTURE
```

Graphical variables can be passed as parameters to functions and procedures. This must be done with care, though. A programmer must keep in mind that a graphical variable contains a pointer to a complex data structure in the graphics data base. When passing a graphical variable by value, only this pointer is being copied, and not the entire structure defining the graphical variable. To avoid unintended results, graphical variables should therefore always be passed by reference, or, in Pascal terminology, as "variable parameters".

Graphical variables are defined by assigning them graphical primitives, graphical function yields, or other graphical variables. LIG/P provides four pre-defined output primitives and four pre-defined functions. The identifiers of the output primitives and pre-defined functions are reserved words and may not be redefined. The output primitives are BLANK, LINE, POLYGON, and literal strings. LINE and POLYGON have absolute and relative coordinate versions. The functions are Arc,
Polyline, Polygon and Tvalue. A distinction is made between polylines and polygons because certain attributes may be used in conjunction with polygons but not with polylines.

Examples for primitives are:

BLANK
LINE FROM x1,y1,z1 TO x2,y2,z2 [single line]
LINE FROM x1,y1,z1 TO x2,y2,z2 TO x3,y3,z3
TO x4,y4,z4 [polyline]
LINE FROM x1,y1,z1 DELTA dx2,dy2,dz2 DELTA dx3,dy3,dz3
DELTA dx4,dy4,dz4 [relative polyline]
POLYGON FROM x1,y1,z1 TO x2,y2,z2
TO x3,y3,z3 [triangle, absolute coordinates]
POLYGON FROM x1,y1,z1 DELTA dx2,dy2,dz2
DELTA dx3,dy3,dz3 [triangle, relative coordinates]
'This is a string primitive.'

The pre-defined functions are declared as follows:

TYPE
  real_array : ARRAY[1..200] OF REAL;
  char_array : ARRAY[1..40] OF CHAR;
FUNCTION Arc(xcentre, ycentre, rad,
             s_angle, f_angle: REAL): GRAPHICAL;
FUNCTION Polyline(x_array, y_array, z_array: real_array;
                  n: INTEGER): GRAPHICAL;
FUNCTION Polygon(x_array, y_array, z_array: real_array;
                 n: INTEGER): GRAPHICAL;
FUNCTION Tvalue(text_array: char_array;
                n: INTEGER): GRAPHICAL;

In addition to the four pre-defined functions a programmer can define his own graphical functions. This is discussed in more detail in Section 3.4.6.
3.4.2 Coordinate Systems

LIG/P supports two coordinate systems: a world coordinate system for modelling and a Normalized Device Coordinate (NDC) system for display purposes. The range of coordinates for the world coordinate system is specified by the application program in an executable statement, i.e. the world coordinates can be changed dynamically. The statement has the format

```
WORLDCOORD xlow..xhigh, ylow..yhigh, zlow..zhigh
```

where xlow,xhigh,ylow,yhigh,zlow,zhigh are real constants or variables. Parameters for output primitives, e.g. LINE FROM x1,y1,z1 TO x2,y2,z2, must lie within the range of world coordinates. If no world coordinate range is explicitly defined, it defaults to the range 0..1 for all three axes.

Display functions use the NDC system; normalized device coordinates have the range 0..1 in all three directions. An application programmer need never be concerned with actual device coordinates — the conversion from NDC to device coordinates is the task of the device drivers.

3.4.3 Modelling Transformations and Attributes

Transformations and attributes are used to modify primitives, subpictures, and the value returned by graphical functions when these entities define a graphical variable. Modifications consist of any subset of the available modification operators; those not specified default to a pre-deter-
mined value. Modification attributes have the highest precedence of all LIG/P operators; their parameters are in world coordinates and are simple constants, Pascal variables or arithmetic expressions. Formally, modifications are defined as

\[
\text{Modification list} := \text{Modification} \\
| \text{Modification list} \text{ Modification} \\
\text{Modification} := \text{transformation operator} \text{ Parameter 3} \\
| \text{attribute} \text{ parameter} \\
\text{Parameter 3} := \text{parameter} \text{ parameter} \text{ parameter} \\
\text{parameter} := \text{variable} \text{ constant} \text{ arithmetic expression}
\]

A transformation operator specifies a geometric transformation which can be scaling, rotation or translation. The set of operators to perform these modifications is \{SCL, ROT, AT\}. Each has three parameters which must be of type real; they define the geometric transformations with respect to x, y, and z. A graphical variable p may for example be assigned a sub-picture of the graphical variable unit_cube, but half the size of unit_cube, rotated and translated. If unit_cube has as value the graphical object as shown in Figure 2, then the sub-picture

\[
\text{unit_cube} \text{ SCL 0.5,0.5,0.5 ROT 30.0,30.0,30.0 DEG}
\]

assigned to p, would give p the value shown in Figure 3. The three parameters of ROT may be followed by the keyword DEG if the three values are in degrees; if DEG is omitted, the values are assumed to be radians.

Translation parameters are in world coordinates. The
system generates an error message if the values lie outside the currently defined world coordinate range. The order of transformations is always scaling followed by rotation followed by translation with the reference point at the origin.

The attributes provided for modelling are the set \{\text{COL, LIGHTN, INTEN, LWIDTH, LTYPE, CSIZE, CSPACE, PATTERN, FILL}\}. Their use has the following effects:

\textbf{COL:} This attribute defines the colour of a primitive or sub-picture. One of eight colours may be selected as a parameter: \text{BLACK, RED, GREEN, BLUE, CYAN, YELLOW, MAGENTA, WHITE}. \text{LIG/P} does not allow mixing of colours or using different hues of a certain colour. If this is desired for larger areas, i.e. for filled polygons, the \text{PATTERN attribute} may achieve the desired effect. The default colour is \text{WHITE}.

\textbf{LIGHTN:} The default lightness of a colour is 1. Any value in
the range 0..1 is valid. The parameter defines the lightness of a corresponding COL-attribute.

INTEN: The intensity of a graphical object defaults to 0.5; this may be changed by specifying a higher or lower value. Some application programs achieve a depth cue by varying the intensity of output primitives. As with colour, the effect of this modification is dependent on the output devices available.

LWIDTH: If a line width other than that produced by a single stroke of the beam or pen of an output device is desired, the parameter of LWIDTH in world coordinates may be used to produce a thicker line.

LTYPE: This modification determines what line style is to be used to output a graphical object. The parameter is an integer value and defaults to 1 for "solid line". In the current implementation line type 2 is a short-dashed line and line type 3 is a long-dashed line.

CSIZE, CSPACE: For text output primitives the character size and spacing between the characters in world coordinates may be reset from the default values. Default values use the hardware character generator that most output devices have.
PATTERN: This operator has an integer parameter which specifies which of all available patterns is to be used to fill a polygon. The patterns available are installation dependent and the result of using this modification is also dependent on the output device.

FILL: This modification has no parameter and applies only to POLYGON primitives. If a polygon is FILLED then either the current PATTERN modification determines the shading of the polygon, or, if no pattern is specified, the COL modification applies.

A graphical variable may be defined as a hierarchy of primitives or subpictures. Certain scope rules must be established to define how a graphical object appears on the view surface if transformations and attributes are applied at different levels in the hierarchy.

For transformations the result is obvious. Transformation parameters are concatenated such that the resulting transformation is the sum of all previous transformations higher up in the hierarchy.

As an example, let the graphical variable p be defined in terms of q, and q be defined in terms of r, where r is the unit cube. A subpicture of q is

\[ r \text{ SCL 0.8,0.8,0.8 BOT 20.0,10.0,0.0 DEG} \]

If q is displayed, the graphical object in Figure 4 is shown.
Now let \( p \) have the subpicture

\[
\begin{align*}
g & \text{ SCL } 0.25,0.5,0.75 \ EOT \ 20.0,0.0,0.0 \ DEG
\end{align*}
\]

Displaying \( p \) results in Figure 5.

Attributes cannot be concatenated in the same way as transformations can; if attribute parameters at different levels are specified, then one will override all others. The following rules apply:

1. If graphical primitives are assigned attributes other than by default then they are always displayed with these attributes.

2. The non-default attributes at the highest level in the hierarchy are applied, unless Rule 1 overrides this rule.

An example shows the effect of these two rules. To il-
lustrate Rule 2, let \( r \) be a filled polygon in the default colour white. Let the graphical variables \( p, q \) and \( r \) be defined in terms of each other as before. If \( q \) has the subpicture

\[ r \text{ COL BLUE} \]

then \( q \) is displayed as a blue polygon. If \( p \) has as subpicture

\[ q \text{ COL YELLOW} \]

then \( p \) is displayed as a yellow polygon. If, however, \( p \) has subpicture

\[ q \text{ [default colour, i.e. no specification]} \]

then \( p \) is displayed as a blue polygon, just like \( q \). To illustrate Rule 1, let \( r \) be defined as the primitive

\[ \text{POLYGON COL RED FILL FROM } \ldots \text{ TO } \ldots \text{ TO } \ldots \]

Then any variable having \( r \) as a subpicture will be displayed as a red polygon (and possibly other parts).

### 3.4.4 The Synonym Assignment Operator

There are two different assignment operators in LIG/P; both are used to define graphical variables, but the variables' internal representations differ if one operator is used instead of the other. A simple assignment statement has the general format

\[ p := q \text{ <modifications>} \]

where \( p \) is a variable of type GRAPHICAL and \( q \) is a primitive, a graphical function or a variable of type GRAPHICAL, even \( p \)
itself. The synonym assignment is used to build a hierarchical structure as definition of a graphical variable. Unlike a numerical assignment such as \( x := y \), where the value of \( y \) is copied into storage location \( x \), the synonym assignment does not copy values but merely sets pointers in the database, in this case from \( p \) to \( q \). Thus, \( p \) is now dependent on \( q \) and if \( q \) changes, \( p \) changes accordingly.

The advantages of this scheme are flexibility, saving of storage space and in general taking into account the actions that are often performed on graphics databases; interactive graphics uses highly dynamic entities and database access to them must be fast; removing a subpicture from the definition of a graphical variable must be done quickly and efficiently, e.g. in terms of garbage collection.

### 3.4.5 The Copy Assignment Operator

The second assignment operator in LIG/P is the copy-assignment operator; the general format of a copy-assignment is

\[
p := q \ <\text{modifications}\]

where \( p \) and \( q \) represent the same symbols as in the previous section.

A copy-assignment traverses the entire data structure of \( q \) and creates a graphical constant from the definition of \( q \). This constant is assigned to \( p \). Subsequently, \( p \) is independent of \( q \) and changes to \( q \) do not affect \( p \). It is evident
that this is an expensive process in terms of time and storage space and should be used cautiously. Once a variable is defined in this way, it is not possible to delete subpictures from it.

3.4.6 Graphical Functions

A graphical variable can be defined with graphical function yields as subpictures. The function name and actual parameter list appear in an assignment statement just like primitives or other subpictures would. If a function has no formal parameters then the special symbol "()" must follow the function name. The general format of an assignment using a graphical function is:

\[
\begin{align*}
a &:= \text{Fname}(p1,p2,\ldots) \ <\text{modifications}> \\
a &:= \text{Fname}() \ <\text{modifications}> \\
a &:= \text{Fname}(p1,p2,\ldots) \ <\text{modifications}> \\
a &:= \text{Fname}() \ <\text{modifications>}
\end{align*}
\]

Graphical functions may have local graphical variables. An application programmer should use this feature cautiously, though. Each time the function exits, the pointers into the data base to the definition of the local variables are lost and the storage space cannot be reclaimed by the LIG/P garbage collection routines. A partial remedy would be the assignment to a simple graphical primitive before exiting the routine, e.g. \text{local_var} := \text{BLANK}, to lose as little space as possible;
alternatively, as many global variables as is reasonable in a particular program should be used, because each time they are redefined the storage space occupied by their previous definition is reclaimed.

Graphical functions are defined like other Pascal functions; a function declaration consists of a function heading, followed by the formal parameter list, followed by the type of object returned by the function. A graphical function returns an object of type GRAPHICAL. Formal parameters may include those of type GRAPHICAL (see comments on this in Section 3.4.1). Before a function returns, a copy-assignment must be executed which assigns a graphical object to the function name. This object is returned as the yield of the function.

The following program section illustrates the use of graphical functions:
PROGRAM funct_demic;

{Example for graphical function}

TYPE
* LIG TYPES;

VAR
* LIG VARS;
  fig : GRAPHICAL;

* LIG PROCEDURES;

FUNCTION Circle_2d(xcent, ycent, rad: REAL): GRAPHICAL;
{Generates a circle with radius "rad" and centre (xcent, ycent) in the z=0 plane.}

CONST
  pi = 3.141593;  z = 0.0;
  ten_degrees = pi/18.0;

VAR
  angl,
  x1, y1, x2, y2 : REAL;
  gen_circle : GRAPHICAL;

BEGIN
  * gen_circle := BLANK;
  angl := 0.0;  x2 := rad;  y2 := 0.0;
  REPEAT
    x1 := x2;  y1 := y2;
    angl := angl + ten_degrees;
    x2 := rad * COS(angl);  y2 := rad * SIN(angl);
  * gen_circle := gen_circle ++ LINE FROM x1, y1, z
  * TO x2, y2, z;
  UNTIL angl >= 2 * pi;

  {assign value to function name}
  * Circle_2d := gen_circle AT xcent, ycent, z;
  * gen_circle := BLANK; {release temporary storage}
END;

BEGIN {main routine}
  * INITIALIZE LIG;
  * WORLDCOORD 0.0..100.0, 0.0..100.0, 0.0..100.0;
  * fig := Circle_2d(20.0, 30.0, 15.0) COL RED;
  ***
END.
  * {End LIG/P} .
3.4.7 The Superposition Operator

The previous sections described the definition of simple graphical objects. The superposition operator "++" is used to define complex graphical objects. A graphical variable can consist of many primitives or subpictures; a hierarchy of these is created by using the synonym assignment operator and the superposition operator. "++" is a dyadic operator with graphical terms as operands to form graphical expressions. A graphical expression is assigned to a graphical variable. Formally,

\[
\text{<assignment statement>} ::= \text{<gr var> <assign op> <gr expression>}
\]

\[
\text{<assign op>} ::= \text{:= | :=}
\]

\[
\text{<gr expression>} ::= \text{<gr term> | <gr expression> ++ <gr term>}
\]

\[
\text{<gr term>} ::= \text{<gr primary> | <gr primary> <modification list>}
\]

\[
\text{<gr primary>} ::= \text{<gr variable> | <gr primitive> | <gr function invocation>}
\]

For example, let:

\[
a := \text{LINE FROM 0.0,0.5,0.0 TO 1.0,0.5,0.0;}
b := \text{a COL RED ROT 0.0,0.0,20.0 DEG;}
c := \text{a COL BLUE ROT 0.0,0.0,40.0 DEG;}
d := \text{a ++ b ++ c}
\]

The variable d, when displayed, shows a white horizontal line, as well as a red and a blue line rotated incrementally by 20 degrees about the z-axis.

When used in conjunction with the copy-assignment operator, a graphical expression is converted to a "constant" and
is assigned to a graphical variable. The statement
\[ d := a + b + c \]
for \( a, b, \) and \( c \) as in the previous example, will visually produce the same result. The internal representation of \( d \) will be different, though. The Sections 3.4.8 and 3.4.9 discuss cases where this is important.

Superposition of a graphical variable onto itself is illegal; a statement of the form
\[ d := a \langle m \rangle + b \langle m \rangle + d \langle m \rangle \]
results in a recursive definition of \( d \) and is not allowed.

Graphical functions may be called recursively, as long as the superposition does not generate a recursive definition of a graphical variable as shown in the example above.

### 3.4.6 The Naming Operator

The naming operator "AS" is used to give a subpicture a unique name by which it can be referred to for deletion and identification. For example,

\[
\begin{align*}
c &:= b \text{ COL BLUE LTYPE 2 AS } b\_blue \\
&\quad + b \text{ COL GREEN LTYPE 3 AS } b\_green
\end{align*}
\]

assigns two named subpictures to \( c \). The variables \( b\_blue \) and \( b\_green \) must be declared of type SUBPICTURE.

The naming operator may not be used in a graphical expression for a copy-assignment. As explained before, the structure of a graphical variable's definition is lost in
copy-assignment and it is therefore meaningless to name parts of it.

Any <graphical primary> may be named to create a named subpicture; in the example above, b could be a primitive, a function invocation, or a variable of type GRAPHICAL. The operations allowed on a named subpicture are not the same as those for a variable of type GRAPHICAL. The only legal and meaningful operations are deletion, identification and display.

3.4.9 The Deletion Operator

This operator, represented by the special symbol "--", allows the removal of named subpictures from the definition of a graphical variable. The subpicture must have been superimposed in a graphical expression evaluated previously, otherwise an error message is generated.

The general form of a graphical expression involving a deletion is

\[
a :- b <m> AS s;
\]

\[
a :- a -- s
\]
or

\[
a :- a ++ c <m> -- s
\]
or

\[
a :- a -- s ++ c<m>
\]

A statement of the form

\[
a :- b -- s
\]
where $s$ is a named subpicture of $b$ is not allowed. Deletion is allowed only at the highest level in a hierarchy of subpictures; the named subpicture must have been superimposed explicitly onto the graphical variable it is deleted from.

3.4.10 The Identification Statements

In the LIG/P system each named subpicture has associated with it an "identification area". This is the $z=z_{\text{min}}$ plane within the range of world coordinates that were valid at the time when the subpicture was defined. The identification area is transformed with the subpicture and can be "picked" by placing a locator inside the subpicture's identification area.

The application program has to test for a pick of a specific subpicture; this is in contrast to many graphics systems where any object can be picked and some identifier is returned to the application program.

LIG/P avoids this on purpose; its high-level constructs avoid the "ambiguity fallacy" discussed by Newman and Sproull [Newm79]:

"Systems designers should avoid the ambiguity fallacy, which postulates that since an item may have several levels of ancestry, it is impossible to determine to which level the user is referring when he points at the item. This problem is often posed using a circuit as an example: if the user points at a resistor, is he trying to indicate the resistor, or the circuit of which the resistor forms a part, or just a certain line within the resistor? This situation should never occur in a well-constructed program, which should always either know the level in which the user is interested or should ask him to
state the level. It is an indication of serious deficiencies in the command language if the user is able to make an ambiguous pointing action."

An identification statement consists of the clause "IF HIT ON" or "IF NOT HIT ON" followed by a subpicture name or a parenthesized logical expression involving subpicture names; the action to be taken is indicated in a statement group following THEN, which is optionally followed by ELSE and a statement group.

The formal definition of an identification statement is:

```plaintext
<identification statement> ::= <simple idn stmt>
                          | <composite idn stmt>
<simple idn stmt> ::= <idn clause> THEN <statement group>
<composite idn stmt> ::= <simple idn stmt>
                       ELSE <statement group>
{idn clause} ::= IF HIT ON <idn expression>
                | IF NOT HIT ON <idn expression>
{idn expression} ::= <subpicture name>
                     | ( <logical expression> )
 logistical expression> ::= <subpicture name> <log op> <subpicture name>
                        | <logical expression> <log op> <subpicture name>
<log op> ::= AND | OR
<statement group> ::= <lig statement>
                     | BEGIN <lig statement list> END
<lig statement list> ::= <lig statement>
                       | <lig statement list> ;
<lig statement>
```

An identification statement must be preceded by the control statement "CURSOR ON" (see Section 3.7) to display a terminal's graphics cursor and wait for an interrupt.

The following program section and figures illustrate the identification process:
VAR

menu, pyramid : GRAPHICAL;

m_item : ARRAY[1..10] OF SUBPICTURE;

BEGIN

* SET COORD 0.0..100.0, 0.0..100.0, 0.0..100.0;
* cube := Unit_cube(0.0, 100.0);
* pyramid := Pyramid_4(0.0, 100.0);
* menu := cube SCL xs, ys, zs AT x, y, z AS m_item[1];
** pyramid SCL xs, ys, zs AT x, y - dy, z AS m_item[2];

* CURSOR ON;
* IF HIT ON (m_item[1] OR m_item[2]) THEN
   <statement group>
* ELSE
   <statement group>

Figures 6 and 7 show the graphical objects cube and pyra-
mid, assuming that graphical functions Unit_cube and Pyramid_4 generate these objects. The dashed frames indicate the untransformed identification areas of cube and pyramid. Figure 8 shows the graphical object menu which consists of the two named subpictures m_item[1] and m_item[2]. These two subpictures may be picked in an identification statement; the cursor must lie in the respective identification area for a hit to have occurred.

It is evident that a rotation about the x or y-axis may distort the identification area to a degree where it could become almost impossible to identify a subpicture. Careful programming can avoid this and ensure that a named subpicture is always defined in such a way that a "reasonable" identification area results.

LIG/P also provides a case-construct to choose between alternative actions, depending on the subpicture that was hit. The case-statement has the general form
The formal definition is:

\[
\begin{align*}
\text{case stmt} & ::= \text{CASE BITON OF case list END} \\
\text{case list} & ::= \text{case list} ; \text{case element} \\
\text{case element} & ::= \text{case label list} : \text{statement group} \\
\text{case label list} & ::= \text{subpicture name} \\
\text{subpicture name} & ::= \text{case label list} , \\
\text{statement group} & \text{ is defined as before.}
\end{align*}
\]

3.5 The Display Functions

The display functions of LIG/P use the definition of graphical variables to generate visible images on a view surface. The language provides facilities to select different output devices, perform clipping, a perspective transformation, mapping from a view volume into a 3D-viewport, and applying image transformations.
3.5.1 Device Definition

Graphical output devices are referred to by names defined in the application program. An output device must be declared as a variable and is of type GROOTPUT (this is analogous to a Pascal file declaration using the pre-defined type TEXT). In an executable statement the declared device is associated with a physical device through its logical unit number. This number is supplied by the operating system at a particular installation. A device association statement has the form

DEVICE <device-id> IS <device-lu>

A <device-lu> is either a named Pascal constant of type integer, a Pascal variable to which an integer value has been assigned, or a Pascal function returning an integer. An example is:

CONST
    sysplot = 9;
    t4014 = 2;
    t4027 = 1;

VAR
    plotter,
    st_tube, col_terminal : GROOTPUT;

BEGIN
    {associate internal devices with physical devices}
    * DEVICE plotter IS sysplot;
    * DEVICE st_tube IS t4014;
    * DEVICE col_terminal IS t4027;
    ...

The devices plotter, st_tube and col_terminal can now be used for graphical output. Not all of them are enabled at this
point, but each device will be if it is referred to in a display statement.

The system provides one master terminal and one hardcopy output device by default. It is therefore not mandatory, and at some installations not even possible, to define and select a number of output devices.

3.5.2 Anonymous and Named Instances

The value of graphical variables, primitives or functions, when displayed on a view surface, is called an instance; thus, every graphical object on the screen of a terminal is an instance. Many instances of the same graphical variable, function or primitive may be displayed at the same time.

Instances may be modified by display modifications. If many instances of the same graphical item are displayed they may be distinguished from each other by giving them names, much like naming subpictures of the same graphical variable. An instance is named by using the operator "AS" followed by an instance name.

Variables of type INSTANCE are declared in a Pascal variable declaration statement. Such variables are used to selectively erase instances from a view surface. Anonymous instances are erased only when all instances of a particular graphical variable are erased or when an entire view surface
is erased.

3.5.3 The Display Statements

A display statement consists of the keyword DISPLAY or HARDCOPY followed by a list of graphical variables, primitives or functions. The difference between the two statements is that HARDCOPYed instances cannot be erased afterwards, and that at the end of a HARDCOPY statement a new-frame action is performed such that only the instances in the same display statement are shown superimposed on each other.

Each graphical item in a display statement may be modified by display attributes, viewing transformations and image transformations (described in Section 3.5.4).

A special operator "->" followed by a device name is used to specify on which view surface the instance is to be shown. If this is omitted then the master terminal or hardcopy device is used as default. A few examples are:

DISPLAY a;  {on master terminal}
DISPLAY a, b, c;
DISPLAY a -> ccl_terminal,
 b -> st_tube;
HARDCOPY a, b -> hi_resol_plotter;  {outputs
   b on special plotter}
HARDCOPY a, b, c  {shows a,b,c super-
imposed, then advances paper}
3.5.4 Display Attributes, Viewing and Image Transformations

Displayed graphical objects may be modified by two display attributes, HILIGHT and PERSPECTIVE. Highlighting of graphical objects is dependent on the output device; in many cases it is achieved by blinking. The attribute PERSPECTIVE has one parameter of type real which defines the centre of projection in world coordinates. The viewing axis is automatically moved to the centre of the front plane of the view volume. If no view volume is specified, the z=0 plane delimited by the x and y world coordinates applies. The parameter of PERSPECTIVE is the distance along the z-axis from the front plane to the centre of projection. This is illustrated in Figure 9. For example, the statement

DISPLAY unit_cube PERSPECTIVE 1.5 -> st_tube

produces the output of Figure 10 (the coordinate axes are drawn only for reference).

Viewing and image transformations are geometric transformations which can be applied only for display purposes without changing the definition of a graphical variable.

Viewing transformations involve the definition of a view volume in world coordinates and the definition of a 3D-viewport in NDC. The view volume dimensions determine which parts of a graphical object are to be clipped. If the projection defaults to parallel, a parallelepiped provides the view volume boundaries; for perspective projection a pyramid deter-
View volumes and 3D-viewports are data types in LIG/P. They have names and are declared as variables, e.g.
vvcl1, vvcl2 : VIEWVOLUME;
 vport1, vport2 : VIEWPORT

Their boundaries are defined in executable statements such as

VIEWVOLUME vvcl1 IS xmin..xmax, ymin..ymax, zmin..zmax;
 VIEWPORT vport1 IS.xlow..xhigh, ylow..yhigh, zlow..zhigh

In a display statement vvcl1 and vport1 may be used in conjunction with the operators WITHIN and INTO to define the viewing transformation, e.g.

DISPLAY p WITHIN vvcl1 INTO vport1 -> col_terminal

If no view volume is specified, the default world coordinate range in which a graphical variable was defined is taken. The viewport defaults to NEC in the x, y and z directions.

Image transformations are the last transformations applied for displaying graphical objects. They determine how a 3D-viewport is transformed before a parallel projection onto the z=0 plane of the NDC space is performed. Image transformations include scaling, rotation and translation; the operators SCL, ROT and AT are used, each followed by three operands which have values in the NDC space. The default values are

SCL 1.0,1.0,1.0 ROT 0.0,0.0,0.0 AT 0.0,0.0,0.0

if no image transformations apply. The parameters of the rotation operator may be followed by the keyword DEG for degrees.

Image transformations are useful if the same object is to be displayed with successive transformations applied, for
example in dynamic graphics. These transformations can often be performed locally by sophisticated graphics satellites without access to the global graphics data base and with no support from the host computer.

3.5.5 The Erase Statement

There are two versions of the erase statement. To erase the entire screen of an output device, either

ERASE SCREEN

or

ERASE SCREEN -> device

may be used. The first statement erases the screen of the master output device; the second erases the specified device.

To selectively erase instances, the erase statement contains a list of graphical variable or instance names. The statement

ERASE p, q, r

where p, q and r are graphical variables, erases all instances of p, q and r on all devices.

If named instances are defined as in

DISPLAY p WITHIN vvcl1 AS i1 -> terminal_1, p PERSPECTIVE 10.0 INTO vport1 AS i2, p SCL sx+dsx,sy+dsy,sz AS i3

then the statement

ERASE i1, i2

removes instances i1 and i2 of p from the devices where they
are displayed.

3.6 Graphical Input

Graphical input in LIG/P is in the two-dimensional space only. The language provides the logical devices keyboard, button, stroke, locator and valuator. Only synchronous input is used, i.e. program execution is suspended when an input statement is found until an interrupt occurs.

Input devices are given internal names in the same way as output devices are. A graphical input device is declared as a variable of type GRINPUT, e.g.

VAR tablet, joystick : GRINPUT

Declared input devices are associated with physical devices in an executable DEVICE statement in the same way as output devices are.

A set of default input devices is initialized and only those are enabled if there are no further specifications in the program. Depending on the hardware available these typically consist of a terminal keyboard, (programmable) function keys and the terminal cursor controlled by keys or thumb-wheels. For this minimal set the valuator is simulated by numeric input via the keyboard, and the buttons by available function keys and the keys of the keyboard.

If input is to be expected from a specific device then the device name preceded by the symbol "<-" is appended to an
input statement, provided that this is meaningful for a specific device.

The LIG/P input statements are:

(i) \texttt{READ CURSOR x,y,button} \\
\texttt{READ CURSOR x,y,button <- screen}

This statement displays the graphics cursor on "screen" and waits for a button interrupt. The coordinates x and y are world coordinate values, and "button" is an integer value returned by the button that was hit. It may be the ASCII or EBCDIC character code if buttons are simulated by the keyboard.

(ii) \texttt{READ BUTTON button} \\

The button number or character code is returned in "button".

(iii) \texttt{READ TEXT array,n} \\
\texttt{READ TEXT array,n <- keyboard-device}

The characters typed at a keyboard device are packed into "array"; the number of characters read is returned in n. The maximum number of characters that can be buffered is installation dependent; if more than this number of characters are typed they are ignored.

(iv) \texttt{READ VALUE x} \\
\texttt{READ VALUE x <- valuator-device}
A real value \( x \) is returned by the valuator.

(v) \[
\text{READ STROKE } x\text{\_array}, y\text{\_array}, n
\]

\[
\text{READ STROKE } x\text{\_array}, y\text{\_array}, n \leftarrow \text{stroke\_device}
\]

A stroke device, e.g. a digitizer, reads \( n \) coordinate values into real arrays "\( x\text{\_array} \)" and "\( y\text{\_array} \). The maximum number of coordinate pairs that can be read at a time is installation dependent.

3.7 Control Statements

The following control statements are provided:

(i) \[
\text{CURSOR ON}
\]

\text{CURSOR ON} \rightarrow \text{device}

These statements display the graphics cursor of an output device and wait for an interrupt. An identification statement can subsequently test for a pick.

(ii) \[
\text{PAUSE } t \quad \{\text{seconds}\}
\]

\text{PAUSE } t\ \text{MIN} \quad \{\text{minutes}\}

Program execution is suspended for the specified length of time, then continues automatically.

(iii) \[
\text{BELL}
\]

\text{BELL} \rightarrow \text{device}
To take advantage of acoustic signals the bell-statement can be used. This corresponds to sending ASCII code 07 to an output device.

(iv) The escape statements

\[
\begin{align*}
\text{SEND } & \text{iarray, } n \rightarrow \text{output-device} \\
\text{SEND } & \text{iarray, } n \leftarrow \text{input-device}
\end{align*}
\]

allow a programmer to make use of device characteristics which are not provided for in the higher level systems. An array of device codes is sent directly to the device; the codes are stored in an integer array and the first \( n \) bytes are to be read by the device.

3.8 Device Setting and Device Inquiry

A number of input and output device characteristics can be set by the application program. This is done in device setting statements. The program can also ask for information about certain devices; this is done by device inquiry statements. In the following sample statements, the device specification ("\( \rightarrow \) device", or "\( \leftarrow \) device") is optional.
3.8.1 Input Device Setting

For input devices, the following device setting statements are provided:

(i) \texttt{LCCFCET xlow..xhigh, ylow..yhigh <- device}
This is analogous to setting a viewport for output devices. The rectangle with two corner points \((xlow, ylow)\) and \((xhigh, yhigh)\), where \(x\) and \(y\) are in cm, is taken as the range of NDC. Only in this region of the device will input data be read. The default low and high values are the lower and upper bounds respectively of the NDC space.

(ii) \texttt{VALRANGE vlow..vhigh <- valuator-device}
This statement sets the full range of the valuator device to values from \(vlow\) to \(vhigh\). All actual values read are mapped into this range by the device driver. The defaults for \(vlow\) and \(vhigh\) are 0 and 1 respectively.

(iii) \texttt{STROKE d,t <- stroke-device}
A stroke device is set to read values every \(t\) seconds or after having traversed a distance \(d\) in NDC. Default values are 0.01 seconds and 0.01 in NDC.
3.8.2 Output Device Setting

For output devices, there are the following device setting statements:

(i) BACKGROUND COl colour -> device
The same colour scheme as for modelling attributes is used. A device screen is filled with the background colour when an ERASE SCREEN statement is executed. By default the background colour of all CRT devices is black.

(ii) ASPECTRATIO xa,ya -> device
The values xa and ya must be in the range 0..1 and one of them must be exactly 1. This establishes an aspect ratio for the specified device; this function is desirable mainly for non-square screens and for cases where one range (x or y) of program-generated coordinates is known not to exceed a certain value and one wants to use the full screen to display the generated graphical objects. The device driver determines a "best fit" for the rectangle of ratio xa:ya for the specified device.
### 3.8.3 Input Device Inquiry

Device inquiry statements interrogate the system about certain characteristics of graphical i/o devices. For input devices the following statements are available:

(i) \[ \text{ASK PRECISION } p \gets \text{ device} \]

The returned value is the precision of the specified locator or valuator device. The number of bits of precision is returned in \( p \).

(ii) \[ \text{ASK STATUS } b \gets \text{ device} \]

The boolean variable \( b \) is set to true or false, depending on whether the specified input device is enabled or disabled.

(iii) \[ \text{ASK LOCFOOT } x_h, y_h \gets \text{ device} \]

The values of \( x_h \) and \( y_h \) are device constants and are the full horizontal and vertical range of the input device, measured in cm, that can be mapped onto the NDC range 0..1.

### 3.8.4 Output Device Inquiry

Inquiring about output device characteristics is done with the following statements:

(i) \[ \text{ASK RATIO } x, y \to \text{ device} \]

The values of \( x \) and \( y \) are device constants in the range 0..1;
one of them is exactly 1. They are used to determine an optimal aspect ratio setting for the specified device.

(ii) **ASK PSIZE x,y -> device**

The values of x and y are device constants and are the physical size of the specified output surface, measured in cm. This is useful for hardcopy output, for example, where graphical objects have to be drawn to a certain scale.

(iii) **ASK RESOLUTION x,y -> device**

The values of x and y are the number of addressable elements per cm of the specified output device. The numbers are a measure of resolution.

(iv) **ASK STATUS b -> device**

The boolean variable b is set to true or false depending on whether the specified output device is enabled or disabled.

3.9 **The Archivation Statements**

Graphical variables can be saved on file and read back at a later stage to facilitate continuity between interactive sessions. The statement

```
SAVE <gr.var.list> -> <system.file>
```

e.g.  
```
SAVE p,q,r -> grfile
```
creates a "constant" of each graphical variable in the list as if the copy-assignment operator had been used and writes the encoded definition onto file, i.e. appends the new information to the end of the file. Graphical objects are retrieved with the statement

```
RESTORE <gr var list> <- <system file>
```

e.g.  
```
RESTORE p,q,r,s,t <- grfile
```

The encoded variable definitions are read from the current position of the file (files should therefore be RESET when the program starts executing) and "constants" are assigned to the graphical variables in the list. If an end-of-file is encountered before all variables have been assigned a value, then the remaining ones are set to BLANK. A file for storing graphical objects must be declared like any Pascal file and have records of type GRFILE.

### 3.10 Translating and Running a LIG/P Program

The currently implemented version of LIG/P at UBC supports all modelling and display functions except for 3D-clipping and selective erasing. Work is under way already to complete the entire system.
A LIG/P source program is preprocessed by the MTS command

$RUN ECM2:LIGP SCARDS=source SPRINT=listing
   PAR= CODE=pascal-source DONNEES=ECM:DONNEES

The file pascal-source is compiled by the command

$RUN *PASCAL SCARDS=pascal-source SPUNCH=object-code

A debugged program should eventually be compiled with the compiler options PAR= $de-,op+$.

To run the program, the LIG/P run-time library and Core System with device drivers have to be loaded with the object code:

$RUN object-code+ECM:LIGP.LIB.0+ECM2:CORE.0+*PASCALLIB
   PAR=ex=40,new=40

The run-time space allocation in the PAR-field may have to be increased for elaborate programs (see the manual UBC PASCAL).

LIG/P programs can be run at Tektronix 4027 colour terminals, Tektronix 4010/4014 storage tube terminals and interface to the UBC Plot routines.
Chapter 4

IMPLEMENTATION OF THE LIG/P SYSTEM

4.1 The LIG/P Preprocessor

The preprocessor which translates LIG/P programs into pure Pascal source code is written in Pascal. It was generated by a compiler writing system (CWS) developed at the University of Montreal [Lecar77]. Input to the CWS is a definition of the language in BNF form. Each production in the grammar may have a semantic action associated with it in the form of a Pascal procedure. Symbols in the grammar may have attributes associated with them which may be of any legal Pascal type. The syntax analysis of the generated compiler uses the bottom-up approach for parsing the source code; it looks one symbol ahead and up to seven symbols down in the stack to decide whether a rule in the grammar applies to make a reduction. Ambiguities which cannot be resolved automatically by the system are resolved by explicit interference in the reduction process by the BNF programmer. A listing of the LIG/P grammar as input to the CWS can be found in Appendix A.
The semantic actions have been omitted for readability of the grammar.

4.2 The LIG/P Run-time Library

A LIG/P program needs the support of the LIG/P run-time library to execute. A set of subroutines builds and manipulates the graphics data base. The library was written in Pascal and is based entirely on dynamic storage allocation provided by the Pascal NEW function.

The environment for all subroutine systems for run-time support (the LIG/P library, the Core System and the three device drivers) is created by dynamically allocating a record for each subsystem and passing a pointer to the appropriate record into each subroutine. This method makes independent subroutine systems out of each major part of the run-time library; the subsystems do not have to be recompiled with the application program to create their proper environment for execution.

4.2.1 The Modelling Data Structure

The data structure for each graphical variable consists of a hierarchical part and a linear part. This reflects the nature of graphical variables. Some are highly structured and access is required at all times to single entities of a variable's definition; on the other hand, a variable may be
defined as a sequence of output primitives with no structure at all, and the only requirement is that the variable as a whole can be manipulated; an example is a digitized map in geographic applications of graphics. A hierarchical data structure is much more complex than a linear one and requires a larger amount of storage space and computation time. When reading the source code, the system can distinguish between the two forms of storage required, translate accordingly, and update the data base appropriately when the program is executed.

The notion of a segment is used to indicate that primitive entities are grouped together and some operations can be applied collectively to all primitives at once. In the LIG/P system reference can be made to two kinds of segments: modelling segments (or abbreviated "m-segments") and display segments (or "d-segments"). M-segments are used to store the unstructured parts of a defined object. The hierarchical part is stored in the modelling tree (or "m-tree").

When a graphical variable is defined, a "root node" is created which has the name of the graphical variable. In the following diagrams this is represented by a shaded block with a name as label, as shown in Figure 11. The right hand pointer points to the graphical variable's m-segment; there is only one per graphical variable and it contains the anonymous primitives and evaluated anonymous functions assigned to it. The bottom pointer points to the m-tree. Pointers leaving nodes
at the bottom are referred to as "superposition pointers" because it is here that new structured parts are added to a variable.

The graphical variable p may therefore be defined as

\[
p := \text{LINE FROM } x_1, y_1, z_1 \text{ TO } x_2, y_2, z_2
\]

and is represented as shown in Figure 12. The entries of an m-segment are encoded output primitives and output device settings. The available m-segment commands can be found in the source code heading of the LIG/P run-time system.

Two special operations can apply to m-segments: they can be extended and they can be copied. New m-segment commands are added for certain superpositions; when anonymous output primitives or anonymous functions are superimposed, the system encodes them and adds them to the m-segment of a graphical variable. Access to them as an independent unit is lost; they cannot be deleted from the variable nor can their attributes
be changed. This "constant" part of a graphical variable is only removed when the entire variable is redefined.

The m-tree, as mentioned before, is the structured part of a variable's definition. This structured part is a highly dynamic data base and full access is permitted to entries in terms of deletion of parts, superposition and attribute changes. A node in the tree contains a pointer to either a named subpicture of a primitive, of a function, or of another graphical variable. When a graphical variable is displayed the tree is traversed and d-segment output primitives are generated; transformations and attributes applying to a particular subpicture are stacked and unstacked as the tree is traversed, and are concatenated to generate the appropriate final values. As new structured parts are added to a variable, new m-tree nodes are allocated and linked into the tree. For efficiency they are always put into the tree immediately after the root node. The diagram of Figure 13 illustrates this. It will become clear in the next section that the system has to keep track of which graphical variables make subpicture references to others. This is required to avoid unnecessary repeated tree traversals; tree traversals are expensive, alone for the many matrix multiplications when concatenating transformations.

It is not known in advance how many links to a graphical variable will be pointing from other m-trees; therefore it is necessary to "thread" the references to a particular variable
in a linked list. The list starts at the root node of a graphical variable and links together all tree nodes pointing to it. To quickly retrieve the root node to which a particular tree node belongs, a pointer from each tree node to its root node is set. Depending on the implementation, storage may be saved by letting only the last node of a tree point back to the root node. For any given tree node one can then follow the superposition pointers until the backward link is encountered.

In the illustration of Figures 14 and 15, the "thread" for subpictures of the graphical variable g is drawn as a dashed line; notice that the links shown are only those relevant to q; the other subpicture references are not shown but follow the same pattern. When the copy assignment operator is used, an m-segment is created which contains all parts of traversed subtrees in a linearized form. It is evident that this is generally expensive in terms of time and space.
Examples are:

\[ p := q \langle \text{m} \rangle \leftrightarrow r \langle \text{m} \rangle \]

Trees of \( q \) and \( r \) are traversed and the output primitives stored in the \( m \)-segment \( cf \) \( p \). For the statement

\[ p := p \leftrightarrow q \langle \text{m} \rangle \]

the subtrees of \( p \) and \( q \) are traversed and output primitives stored in the \( m \)-segment \( cf \) \( p \). The subtree of \( p \) is then de-
No deletion operation is possible in a copy assignment statement, and no deletion of subpictures is possible once a graphical variable is defined in this manner.

4.2.2 The Display Data Structure

For the execution of its display functions, the LIG/P system relies on the Core System which is proposed as the Computer Graphics Standard [GSPC79]. The Core System provides a single-level structure of display statements which consist of grouped output primitives. The LIG/P run-time library maps the hierarchical structure of graphical variables onto single-level Core segments (or "c-segments").

Displaying and erasing of graphical objects causes a number of actions in the Core System. For each instance that is displayed a c-segment is created. The Core System performs the clipping and window-to-viewport transformation of coordinate data; for each instance the LIG/P system selects the requested device, sets the viewing transformations and initiates the creation of a new c-segment; it then traverses the graphical variable's data structure to generate Core System output primitives. The m-segment is scanned through quickly because it has a very simple structure; then the m-tree is traversed. The tree traversal routine is recursive and "brother nodes" and their subtrees are evaluated before "son nodes" and their
subtrees are.

To avoid repeated unnecessary creation and deletion of c-segments the following scheme is used:

Any existing c-segment can be visible or invisible. With respect to a LIG/P model, it can be outdated or not. Since the Core System provides image transformations and highlighting as dynamic attributes for complete segments, LIG/P does not have to destroy a c-segment as soon as it is erased; rather, the c-segment is kept because it is assumed that there is a good chance that it will be redisplayed with new image transformations or highlighting. Consider the following sequence of LIG/P statements:

{1} DISPLAY p <imtrans1> -> tek27;
{2} DISPLAY p <imtrans2> -> tek27;
...
{3} ERASE p;
{read new parameters}
{4} DISPLAY p <imtrans3> -> tek27

Statements 1 and 2 cause the creation of two c-segments which are immediately visible. Statement 3 makes these c-segments invisible. Statement 4 is executed and p's instance chain is searched to see if there is any c-segment which can be used, i.e. which refers to the specified device and is currently invisible, and is not outdated. In the above example, the segment is found for which all these conditions are true, and one of the two segments is made visible again after the new image transformations have been applied.
In two cases this scheme will not work properly: The first is the case where viewing transformations have changed and different parts of an object are being clipped. Therefore another condition that has to be checked is that the current viewing transformations must be the same as the previous ones. Also a perspective projection must not have changed. If this is not the case, a new c-segment must be created. The second condition is that the definition of a variable has not changed between the last DISPLAY statement and the current DISPLAY statement. The definition of a graphical variable changes as soon as parts are added, or subpicture attributes or subpicture definitions are changed. For example:

```prolog
p :- g <m1>;
DISPLAY p;
q :- g -- gpart;
ERASE p;
DISPLAY p
```

As soon as q is changed, all variables using g as a subpicture are flagged. This is done very quickly by scanning through g's subpicture "thread". Therefore p gets flagged. Then p's instance chain is traversed and all nodes visited are flagged as outdated. The c-segments corresponding to the instance nodes that are both outdated and invisible are destroyed. Other c-segments which at this point are still visible are destroyed as soon as they are made invisible. An example:
\[ p ::= p ++ q \langle m1 \rangle ; \]
DISPLAY \( p \langle \text{imtrans} \rangle \) AS \( p_{\text{ins}} \rightarrow \text{tek27} \);
DISPLAY \( p \rightarrow \text{tek25} \)

Figure 16 shows the corresponding data base entries for these statements. The root nodes of device-trees in the diagram are shaded similarly to those of m-trees. The allocation of node (2) and (3) causes the creation of, say, c-segments number 102 and 103. Both nodes (2) and (3) are flagged "visible" and "up-to-date". The statement

\[ q ::= q -- \text{qpart} \]
causes the following changes: node (1) points to graphical variable \( p \) as its root node which has now been modified because \( q \) has been modified. Following the link from \( p \) to node (2) to node (3), these two nodes are now flagged "visible" and "outdated". The statement

\text{ERASE} \ p_{\text{ins}}
flags (2) "invisible" and "outdated" and therefore destroys c-segment 102. (3) is still "visible" and "outdated". The statement
DISPLAY p \rightarrow tek27

causes a new c-segment to be created, say c-segment number 104. A new node in the d-tree of device tek27, node (4), is allocated and flagged "updated" and "visible". The current data base is shown in Figure 17.

If the next statements are

\begin{verbatim}
BASE p;
DISPLAY p <imtrans4> \rightarrow tek27
\end{verbatim}

then c-segment 103 is destroyed and node (3) removed, c-segment 104 is made invisible and then visible again with new image transformations applied.

By now it should be clear why such a complex structure of trees and linked lists is used in LIG/P. The method is not space-efficient but it is expected that a good deal of execution time is saved by not using a brute-force method with a straightforward scheme of creating and destroying segments, and by avoiding traversal of the model-structure as often as possible.
Chapter 5

COMMENTS ON THE CORE SYSTEM

One of the goals of the investigations for this thesis was the usefulness of the Core System for the automatic generation of calls to its functions.

When the LIG/P data structure was designed the original intent was to make Core System segments part of the data base to avoid duplication of encoding and decoding of output primitives, and to save storage space. It was discovered that this was not possible; LIG/P had to be given its own m-segments which at some point would be translated into Core System segments for display purposes.

The reason is that Core segments cannot be copied and cannot be extended. Although it is claimed that a hierarchical structure can be built on top of the single-level Core segments [Mich78], this is not easily done and is in fact very time-consuming and requires a large amount of overhead.

LIG/P supports a multi-level structure of graphical variables and subpictures: a particular variable can serve as a subpicture to many other variables. A problem arises because each subpicture may have different modelling transformations.
The Core System Report [GSPC79] specifies that image transformations may be applied to a segment only once it has been defined; the LIG/P system could imitate modelling transformations by applying these image transformations while displaying a subpicture. However, image transformations in the Core System are immediately applied to all displayed instances of a segment which would retroactively change all images of the particular segment.

An example illustrates this point: Let \( p \) and \( g \) share a Core segment \( \text{seg}(r) \) which defines \( r \):

\[
p := r <m1>; \\
g := r <m2>; \\
\text{DISPLAY } r; \quad \text{[generates the first image of } \text{seg}(r)] \\
\text{DISPLAY } p; \quad \text{[changes image transformations of } \text{seg}(r) \text{ to } <m1>] \\
\text{DISPLAY } q; \quad \text{[changes image transformations of } \text{seg}(r) \text{ to } <m2>]
\]

When the last statement is executed, \( r \) and \( p \) are no longer displayed the way they should be. This leads to the conclusion that for the representation of a hierarchy, segments cannot be shared; one needs a copy for each subpicture. The Core System does not allow the generation of identical copies of segments with a simple command, however.

A second major drawback of Core segments is the restriction that they may not be extended. If a graphical object is composed of many output primitives which do not have to be accessed later (e.g., for deletion) but all of which are not
known in advance because they are added to a variable during an interactive session, then surely an extend-segment instruction would be a reasonable answer to avoid a large amount of overhead.

A third problem arises due to the fact that there is no access to primitive attributes within a segment. With respect to LIG/P this rules out Core segments as part of a hierarchical model structure.

The only advantages that Core segments offer, compared to a simple set of unsegmented output primitives, is (a) the deletion of entire segments and (b) the application of dynamic segment attributes to entire segments. The Core segment was not designed to be part of a graphical database; graphical data structures must be maintained by the application program in the form of its own data and procedure structure. A good example to prove this is given by Bergeron et al. [Berge78] who anticipate frequent creation and deletion of segments. Michener et al. [Mich78] suggest the use of temporary work segments to achieve the effect of copying and extending segments.

Graphics packages such as GINO-P or the German Kernel System (GKS) provide segments with more permanent characteristics and may possibly be more suitable as a subsystem for LIG/P.
Chapter 6

CONCLUSION

Due to time constraints the full LIG/P system could not be implemented up to this point; the largest part of it, however, has been implemented and tested and work is already continuing to complete the project.

LIG/P is a major improvement over previous versions of LIG in terms of programming convenience, portability and graphics features it provides. It is anticipated that further experiments will discover where more effective use of the Core System can be made by using its full capabilities once they are available. At some stage when the flexibility of the Compiler Writing System is not as essential any more, it is recommended that a preprocessor be written from scratch; the current version is rather expensive to use.

It is hoped that LIG/P will in the near future be used as a teaching tool for the undergraduate graphics course in this university. As with LIG, the feedback from users will provide a very valuable basis for the evaluation of and improvements to the system.
BIBLIOGRAPHY


[Lecar77] Lecarme O., Bochmann G.V., A Compiler Writing


Appendix A

THE LIG/P GRAMMAR

The grammar in BNF form shown on the following pages is the input to the Compiler Writing System used to implement the preprocessor. The semantic actions have been removed in this listing for most productions to increase the readability of the grammar. The attributes associated with many symbols in the grammar appear as in the original version; their types are defined in the auxiliary file GLOBAUX which is not listed in this report.
Appendix B

A SAMPLE PROGRAM

The LIG/P sample program listed on the following pages was used to produce the output shown in Figures 18 and 19. The output device was a Tektronix 4027 terminal.
Program to demonstrate LIG/P; it defines a cube in wire frame representation and its sides partially filled by coloured polygons. The cube is transformed and automatically recreated after each transformation in such a way that a hidden-surface effect is always achieved.

PROGRAM ligdemo;

CONST
  minv = 0.0; maxv = 1.0;
  centv = 0.5; ofst = 0.10;

TYPE
  * LIG TYPES;
arrayU = ARRAY[0..3] OF BEAL;
matrix4 = ARRAY[0..15] OF BEAL;
surface = RECORD
  x,y,z : BEAL;
  faceno : INTEGER
END;

zface = RECORD
  x : BEAL;
  faceno : INTEGER
END;

VAR
  * LIG VA$;
  c_frame,
  c_sides : GRAPHICAL;
  vpl, vp2 : VIEWPORT;
  mtx : matrix*;
  l, xang, xangd, zangd : INTEGER;
  d, xang, yang, zang : INTEGER;
  xsc, ysc, zsc,
  xtr, ytr, ztr,
  xmove, ymove, zmove : REAL;
  answer : CHAR;

connect = array (vertex connections for faces)
(1,2,3,4),
(1,2,5,6),
(2,3,7,8),
(1,4,7,6),
(2,3,8,5)
end;

centrep = array (centre points of faces)
(centerx,centry,minv,1),
(centerx,centry,maxv,2),
(centerx,centry,minv,3),
(centerx,centry,maxv,4),
(minv,centry,centrv,5),
(maxv,centry,centrv,6)
end;

offace = array (offset of filled area for each face)
(true,true,false),
(true,false,true),
(true,false,false),
(false,true,false),
(false,true,true)
end;

answer = 'y*;

** UBC LIG/P Preprocessor **

[Program to demonstrate LIG/P; it defines a cube in wire frame representation and its sides partially filled by coloured polygons. The cube is transformed and automatically recreated after each transformation in such a way that a hidden-surface effect is always achieved.]

PROGRAM ligdemo;

CONST
  minv = 0.0; maxv = 1.0;
  centv = 0.5; ofst = 0.10;

TYPE
  * LIG TYPES;
arrayU = ARRAY[0..3] OF BEAL;
matrix4 = ARRAY[0..15] OF BEAL;
surface = RECORD
  x,y,z : BEAL;
  faceno : INTEGER
END;

zface = RECORD
  x : BEAL;
  faceno : INTEGER
END;

VAR
  * LIG VA$;
  c_frame,
  c_sides : GRAPHICAL;
  vpl, vp2 : VIEWPORT;
  mtx : matrix*;
  l, xang, xangd, zangd : INTEGER;
  d, xang, yang, zang : INTEGER;
  xsc, ysc, zsc,
  xtr, ytr, ztr,
  xmove, ymove, zmove : REAL;
  answer : CHAR;

connect = array (vertex connections for faces)
(1,2,3,4),
(1,2,5,6),
(2,3,7,8),
(1,4,7,6),
(2,3,8,5)
end;

centrep = array (centre points of faces)
(centerx,centry,minv,1),
(centerx,centry,maxv,2),
(centerx,centry,minv,3),
(centerx,centry,maxv,4),
(minv,centry,centrv,5),
(maxv,centry,centrv,6)
end;

offace = array (offset of filled area for each face)
(true,true,false),
(true,false,true),
(true,false,false),
(false,true,false),
(false,true,true)
end;

answer = 'y*;
### DBC LISP/Preprocessor ###

```lisp
PROCEDURE make_matrix(nx, ny, sz, ax, ay, az, tx, ty, tz : REAL; VAR mtx : matrix4);
  (generates a 4x4-transformation matrix)
  VAR
  sinax, sinay, sinaz,
  cosax, cosy, cosaz,
  sinax_sinay, cosax_sinay : REAL;
BEGIN
  sinax := SIN(ax); cosax := COS(ax);
  sinay := SIN(ay); cosy := COS(ay);
  sinaz := SIN(az); cosaz := COS(az);
  sinax_sinay := sinax * sinay;
  cosax_sinay := cosax * sinay;
  mtx[0] := sx * (cosax * cosaz);
  mtx[1] := sx * (cosax * sinaz);
  mtx[2] := -sinax * sx;
  mtx[3] := 0.0;
  mtx[4] := sy * ((sinax_sinay * sinaz) • (cosax * cosaz));
  mtx[7] := 0.0;
  mtx[8] := sz * ((cosax_sinay * cosax) • (sinax * sinaz));
  mtx[9] := sz * (cosax_sinay * sinaz) • (cosax * cosaz));
  mtx[10] := sz * (cosax * cosay);
  mtx[11] := 0.0;
  mtx[12] := tx;
  mtx[13] := ty;
  mtx[14] := tz;
  mtx[15] := 1.0;
END;

PROCEDURE transform(VAR mtx : matrix4; x, y, z : REAL; VAR xtr, ytr, ztr : REAL);
  (Transforms a 3D-point according to the 4x4-matrix received)
BEGIN
  xtr := (x * mtx[0]) + (y * mtx[4]) + (z * mtx[8]) • mtx[12];
  ytr := (x * mtx[1]) + (y * mtx[5]) + (z * mtx[9]) + mtx[13];
  ztr := (x * mtx[2]) + (y * mtx[6]) + (z * mtx[10]) + mtx[14];
END;

PROCEDURE offset(VAR x, y, z: array4; xoff, yoff, zoff : BOOLEAN);
  { Kotifies points to achieve offset from cube vertex }
  VAR i : INTEGER;
BEGIN
  IF xoff THEN
    BEGIN
      FOR i:=0 TO 3 DO
        IF x[i] = minv THEN x[i] := minv+offst
        ELSE IF x[i] = maxv THEN x[i] := maxv-offst
    END;
  IF yoff THEN
    BEGIN
      FOR i:=0 TO 3 DO
        IF y[i] = minv THEN y[i] := minv+offst
        ELSE IF y[i] = maxv THEN y[i] := maxv-offst
    END;
  IF zoff THEN
    BEGIN
      FOR i:=0 TO 3 DO
        IF z[i] = minv THEN z[i] := minv+offst
        ELSE IF z[i] = maxv THEN z[i] := maxv-offst
    END;
  END;

FUNCTION faces(level : INTEGER): GRAPHICAL;
  (Recursive; generates six cube faces in the right order)
  VAR
  cur_face, cur_face, cur_fill : GRAPHICAL;
  maxval : zface;
  indx, conindx, polynum, i, mark : INTEGER;
  dist, xpoints, yPoints, zpoints : array4;
```

---

The code above demonstrates a procedure to generate a 4x4 transformation matrix and transform 3D points, and offsetting points to achieve offsets from the cube vertex. The `faces` function recursively generates six face representations of a cube, and the `offset` function modifies points by adding offsets to achieve a desired result from the cube vertices.
BEGIN

{ find smallest z-coordinate of centre points }
mark := level; maxval := tr_zcentre[level];
FOR i:=level+1 TO 6 DO
IF tr_zcentre[i].z < maxval.z THEN
BEGIN mark := i; maxval := tr_zcentre[i] END;
IF mark > level THEN (* exchange *);
BEGIN
tr_zcentre[mark] := tr_zcentre[level];
tr_zcentre[level] := maxval
END;
{ generate polygon face }
polygon := tr_zcentre[level].faceno;
{ define frame of cube face }
FOR i=0 TO 3 DO
BEGIN
indx := connectpolynum,i];
xpoints[i] := vp(indx,1];
ypoints[i] := vp[indx,2];
zpoints[i] := vp[indx,3]
END;
* cur_frame := POLYGON COL RED
FROM xpoints[0], ypoints[0], zpoints[0]
TO xpoints[1], ypoints[1], zpoints[1]
TO xpoints[2], ypoints[2], zpoints[2]
TO xpoints[3], ypoints[3], zpoints[3]
{ define shaded part of cube face }
offset(xpoints,ypoints,zpoints, of face[polynum,1],
of face[polynum,2], of face[polynum,3]);
* cur_fill := POLYGON FILL
FROM xpoints[0], ypoints[0], zpoints[0]
TO xpoints[1], ypoints[1], zpoints[1]
TO xpoints[2], ypoints[2], zpoints[2]
TO xpoints[3], ypoints[3], zpoints[3];
CASE polynum OF
1 : BEGIN
* cur_fill := cur_fill COL RED;
END;
2 : BEGIN
* cur_fill := cur_fill COL BLUE;
END;
3 : BEGIN
* cur_fill := cur_fill COL GREEN;
END;
4 : BEGIN
* cur_fill := cur_fill COL YELLOW;
END;
5 : BEGIN
* cur_fill := cur_fill COL CYN;
END;
6 : BEGIN
* cur_fill := cur_fill COL MAGENTA;
END;
END;
* cur_face := cur_fill ** cur_frame;
IF level < 6 THEN BEGIN
* cur_face := cur_face ** faces(level+1);
END;
* faces := cur_face;
{ garbage collect }
* cur_frame := BLANK; cur_fill := BLANK;
END;
BEGIN (ligdeo )
INITIALIZE LIG;
VIEWPORT vp1 IS 0.0..0.5, 0.25..0.75, 0.0..0.5;
VIEWPORT vp2 IS 0.5..1.0, 0.25..0.75, 0.0..0.5;
WHILE (answer='y'( 03 (answer='Y") DO
BEGIN
ERASE SCREEN;
write(" Enter angle of rotation about x-axis in degrees (integer value). ");
read(xang); xang := (xang/180.0) * 3.1415;
write(" Enter angle of rotation about y-axis.");
read(yang); yang := (yang/180.0) * 3.1415;
write(" Enter angle of rotation about z-axis.");
read(zang); zang := (zang/180.0) * 3.1415;
write(" Enter scale factor w.r.t. x-axis (0.0 <= scl <= 1.0)");
read(xsc); xsc := (xsc/100.0) * 3.1415;
write(" Enter scale factor w.r.t. y-axis.");
read(ysc); ysc := (ysc/100.0) * 3.1415;
write(" Enter scale factor w.r.t. z-axis.");
read(zsc); zsc := (zsc/100.0) * 3.1415;
makematrix(xsc, ysc, zsc, xang, yang, zang, trns);
262 FOR i = 1 TO 6 DO
263 BEGIN
264 transform (x, xcentre[i].x, xcentre[i].y, centre[i].z, 
265 x, ytr, tr_zcentre[i].t);
266 END;
267 END;
268 c_frame :- POLYGON FROM minv, minv, minv TO maxv, minv, minv 
269 TO maxv, maxv, minv TO minv, maxv, minv COL RED
270 ++ POLYGON FROM maxv, maxv, minv TO minv, maxv, maxv COL RED
271 ++ LINE FROM minv, minv, minv TO minv, maxv, maxv COL RED
272 ++ LINE FROM maxv, maxv, minv TO minv, maxv, maxv COL RED
273 ++ LINE FROM maxv, maxv, siv TO maxv, siv, maxv COL RED
274 ++ LINE FROM maxv, maxv, siv TO minv, maxv, maxv COL RED
275 ++ LINE FROM siv, maxv, siv TO minv, maxv, maxv COL RED
276 ++ LINE FROM maxv, maxv, siv TO minv, maxv, maxv COL RED
277 c_sides :- faces[i]; [Graphic function call]
278 [apply image transformations to display cube ]
279 c_frame :- c_frame SCL xsc, ysc, zsc ROT xangd, yangd, zangd DEG
280 AT centrv-xmove, centrv-ymove, centrv-zmove;
281 c_sides :- c_sides SCL xsc, ysc, zsc ROT xangd, yangd, zangd DEG
282 AT centrv-xmove, centrv-ymove, centrv-zmove;
283 DISPLAY c_frame INTO vpl, c_sides INTO vpl,
284 c_frame PERSPECTIVE 1.5 INTO vpl,
285 c_sides PERSPECTIVE 1.5 INTO vpl;
286 write(' Do you want to continue? (y,n) ?'); writeln;
287 REPEAT read(answer) UNTIL answer=''
288 END
289 END.
290 END. [End of LIG/P].

**** No errors detected. ****
Appendix C

THE RUN-TIME SYSTEM CALLING GRAPHS

The following pages show the interdependence of routines of the LIG/P run-time library. This listing is primarily intended to assist future work on the system. The routine names of the Core System have no special prefix and are all in upper case.
ligsprend:
ligsprend ligmisc ligerror

ligan_string:
ligan_string liginsert ligwaseq

ligapstring:
ligapstring ligksp ligsubp_init

ligan.fa:
ingan.fa ligasseq_garbage

ligsp.fa:
ingsp.fa ligsubp_init

ligstr_dump:
ingstr_dump ligstr_dump ligerror

ligtraverse:
ingtraverse ligtraverse liginsert liginsert
liginsert liginsert ligksp
ligerror

ligvar_mseg:
ingvar_mseg ligtraverse ligintsseg ligwaseq
ligerror

ligsuperpos:
ingsuperpos liglink_node
ligwaseq
ligwaseq
ligerror

ligsuperpos:
ingsuperpos ligwroot
ligsuperpos

ligsanfa:
ingsanfa ligwaseq

ligdeletion:
ingdeletion ligerror

lig.append:
ing.append ligerror

ligdevice:
ingdevice INITIALIZE_VIEW_SURFACE
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ligerdevice lame</td>
<td>Select view surface</td>
</tr>
<tr>
<td>ligerdevice ligerdevice</td>
<td>De-select view surface</td>
</tr>
<tr>
<td>ligerdevice ligerdevice</td>
<td>New frame</td>
</tr>
<tr>
<td>ligerdevice ligerdevice</td>
<td>Delete retained segment</td>
</tr>
<tr>
<td>ligerdevice ligerdevice</td>
<td>Set segment visibility</td>
</tr>
</tbody>
</table>

**Segement Visibility**

**Select View Surface**

**De-select View Surface**

**New Frame**

**Close Retained Segment**

**Create Retained Segment**

**Set Viewport**

**Set Front Plane Clipping**

**Set Window Clipping**

**Set View Reference Point**

**Set Projection**

**Set View Depth**

**Set Window**

**Set Image Transformation_3**

**Set Highlighting**

**Select**

**De-select**

**New Frame**

**Select View Surface**

**De-select View Surface**

**Select View Surface**

**De-select View Surface**

**Select View Surface**

**De-select View Surface**

**Select View Surface**

**De-select View Surface**