SLAC-187
UC-32

A SYSTEM FOR LARGE STRUCTURE GRAPHICS*

BARY WILLIAM POLLACK

STANFORD LINEAR ACCELERATOR CENTER

STANFORD UNIVERSITY

Stanford, California 94305

PREPARED FOR THE U.S. ENERGY RESEARCH AND DEVELOPMENT ADMINISTRATION
UNDER CONTRACT NO. E(04-3)-515

Manuscript Completed August 1975

Printed in the United States of America. Available from National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161. Price: Printed Copy $8.50; Microfiche $2.25.

* Ph.D. dissertation.

ABSTRACT

In this thesis we present the design for a system with the potential for solving real-world large structure graphics problems. Such problems are continually encountered in industry. Although present-day techniques for drafting, architectural drawing, airfoil design, automobile and ship design, and so forth are to some extent automated, these are mainly areas which are approached with traditional and mechanical methods.

This research: 1) demonstrates the practicality and power of using a parallel systems approach to graphical system design, 2) develops a dual data structure which is especially efficient in dealing with large structures, and 3) unifies a variety of techniques currently available in several disciplines.

The culmination of this research has been the implementation of the GRAPL system. Additionally, we have developed two languages: the GRAPL implementation language in which the GRAPL system is actually implemented, and the GRAPL command language, which forms the user-system interface.

GRAPL provides a system with which one may design a structure of major complexity. It is readily tailored to various user requirements while remaining efficient in its real-time response. And, GRAPL is capable of accepti "advice" on how it may improve its performance.

PREFACE

I wish to express my most sincere thanks to Professor William F. Miller who continually provided encouragement, guidance, and insight, and who shepherded me through this arduous task. Without his support I never could have completed this research. Dr. Alan Kay provided much of the early motivation for this project, guided me into the literature, and has always provided the most stimulating questions, suggestions, and criticisms. His hand is evidenced throughout.

I also would like to express my appreciation for the advice, support, and friendship given me by the other members of my reading committee -- Proffessors Forest Baskett and Cordell Green -- as well as to my fellow graduate students and co-workers at the SLAC Computation Group -- Jim George, Steve Levine, Jerry Friedman, and Harriet Canfield.

DEDICATION

This work is dedicated to my parents,
Seymour and Evelyn Pollack,
and to my wife,
Kimberly,
without whose support, confidence,
and understanding I surely would
have given up long ago.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1 02:38:08 08/04/75

1 Introduction

Which virtues of a computer-based graphics system are attractive to an architect or an urban planner? Why should a designer use such a system? Obviously, the savings made in the automation of the drafting process alone are inadequate reasons for choosing a computer-based system. Computer graphics has the potential for offering designers the following: a drawing medium better than paper -- more flexible, of higher resolution, more easily edited; a display medium superior to paper -- dynamic perspective, three-dimensional representation, translation, rotation, scale, etc; an assistant who "understands" and "remembers" the class of problems under consideration; instant accessibility, permanent storage of notes and designs; the ability to perform design-related calculations simultaneously with the design process; the possibility of trying out several alternative designs at small additional cost; seeing how these look, their expense, length of time for construction; and much more.

1.1 Objectives of this thesis

This thesis demonstrates the utility and power of two concepts: 1) a system design which is essentially parallel admitting recursion, backtracking, coroutines and parallel routines; and 2) a dual data structure especially suited for the design of extremely large structures -- these are structures so large that perhaps only 5% to 10% of the data can reside in core at any one time.

In addition, this thesis provides the basis for a class of systems with the following characteristics: the systems are capable of handling large data structures, are interactive, are capable of dealing with incompletely specified problems, have easily modified syntax and semantics, are easily and naturally extendible, and are portable.

Research generally progresses in two directions: in the development of new techniques, and in the application, simplification, and unification of existing techniques. This thesis is primarily a study in the second direction.

We view graphics primarily as a problem in system design. Although many advances recently have been made in this field, our research offers a clean synthesis of many of the techniques which have been developed.

1.2 Automation of design

A distinction may be made between the terms "design automation" and "the automation of design." The former usually means the attempt to replace the designer by a computer system capable of performing some subset of the usual design tasks. "The automation of design," on the other hand, is an attempt to bring to the designer the advantages of computer techniques. Our goals lie in the latter domain. We are not trying to solve the problems of the professional designer or partially replace him; rather, we are offering him tools which may enhance his problem-solving ability.

1.3 The user interface

The application of new techniques to a discipline is governed largely by how readily these techniques lend themselves to the specific problems which arise and by how readily these problems may be formulated within the scope of the new methods. It is for this reason that the user interface is of greatest importance. If it is too difficult, requires too much effort, demands too much time to use a new approach, traditional methods will continue to be used, even though they may be less powerful, less concise

or less esthetic.

Any computer design system which attempts to be useful to the design profession at large should meet most of the following criteria:

1) The system should offer a medium significantly better than paper. It should be more flexible, natural and easier to use, and more efficient for the designer to use the computer than to use traditional drawing and drafting techniques. The GRAPL system meets these criteria.

2) Problems which are easy to solve should be stated simply. Commands to generate the usual kinds of drawings should be readily accessible. The GRAPL system meets this criterion.

3) More complicated problems should grow corresponding to their complexity. A complicated design might well take a significant amount of time to create, but it should in no event take longer or cost more to produce a complicated design with the computer system than by traditional methods. The GRAPL system meets this criterion.

4) The basic concepts and primitives in the design system should be natural and appropriate to the design

process. An architectural designer should not have to learn concepts which are not part of his profession. Rather, the computer system designer should tailor his system to the needs and requirements of the architect. To the extent of the commands we have implemented, the GRAPL system meets this criterion.

5) The system should be relatively easy to modify, not only by the system programmer but also by the user. He should be able to define his own constructs and give their semantics; he shcuid be able to create macros in order to abbreviate, he should be able to change the meanings of already defined constructs. The GRAPL system meets this criterion.

6) The system should be sufficiently powerful so that problems of similar types may be solved essentially by analogy to already solved problems. The GRAPL system does not meet this criterion. Very serious questions arise when one attempts to be precise about what one intuitively means by "solving by analogy." The GRAPL system has the potential power and flexibility to implement analogy-solving systems at least as ccmplex as those already in existence (the high-school wcrd problem system of Bobrow, for example). We have not chosen tc pursue this avenue of research although we realize that it is especially important if we intend to

interface to the non-computer professsional.

7) The system should facilitate control of the degree of detail to be used at all levels. The user should be able to show all details, eliminate extraneous details from a picture, compute gross costs, compute precise or approximate results, and so forth. The decision as to the amount of detail and precision to be used should rest with the user. The GRAPL system meets this criterion.

8) The system should perform in real time. The GRAPL system only partially meets this criterion, primarily due to limitations inherent in the particular hardware-software environment in which we have implemented it. Given a dedicated machine, GRAPL would meet this criterion.

1.4 Graphics from a different point of view

There are several ways in which our approach to graphics differs from the traditional. Perhaps of greatest importance is the fact that we view graphics as a system design problem. This means that we are not bound by any specific language, data structure, control structure, or even hardware. Rather, we allow the requirements of the graphical design process to impose themselves and we present

a system which reflects these requirements. In particular, these requirements are reflected in: 1) the overall system design, 2) the types of data structures, 3) the format in which knowledge is represented -- as program, and 4) the maintenance of programs in interpreted rather than in compiled form.

Many attempts in graphics primarily have been concerned with which primitives to include in the command language, which primitive data structures to implement, and which primitives to include in the base language. Such decisions are important, but these decisions need not have the major emphasis. Graphics is essentially a modeling problem -- we should be most concerned with which properties form a good model of the world for the purposes of computer graphics. It is important to note that we were led to the development of our particular model by the requirements of the real world (not, say, by the a priori decision that ring structures are better suited for graphics than list structures).

The traditional design of graphics systems assumes that a large amount of numerical calculation is necessary both to produce the pictures required and to complete the calculations requested. As a result, most graphics systems have been implemented in either a computationally oriented

algorithmic language (such as Fortran or PL/1) or in a machine language. Because we view graphics as a modeling rather than as a computational problem, our choice of language has been one which includes general simulation capabilities. Rather than attempting to modify a language such as SIMULA or GPSS to include the general data structure facilities and control structures, which we find useful in graphics, we have elected to design and implement our own language. A description of the GRAPL language may be found in Chapter 5.

After rather exhaustive search we have found that few graphics systems have used many of the techniques currently available in the field of artificial intelligence. The closest application has been the work of Terry Winograd -- but his work is primarily in linguistics; he used graphics solely as a way to observe his simulated computer arm. The use of modifiable strategies for display and computation, the use of heuristics for control of detail, the representation of knowledge in the form of programs are all new to graphics. These approaches represent a significant advance both in providing the kind of generality necessary for the design of usable graphics systems and in providing the capability for the implementation of truly responsive systems.

A command language which may be fitted to a user's requirements is an attribute of any system which attempts to communicate with the non-computer specialist. Towards this end graphics systems have usually allowed the user to define macros in an effort to have a more flexible and concise user interface. A few graphics systems have gone further -- they have allowed the system programmer to specify the command language at system generation time and then run the system description through a pre-processing stage. We have gone still further -- the design of the command language has been brought to the user so that it may best be designed to satisfy his particular needs. The system we present has all the necessary tools available for such an endeavor.

Open-ended interpretive systems such as ours usually suffer great run-time inefficiency. This is in part due to the inherent overhead in interpretation and in part due to our great system generality. We have been able to keep this inefficiency to a minimum through the use of selective and incremental compilation. That is, those highly utilized portions of the system which are inefficient may be compiled by the user (or the system designer) to gain greater execution speed and more compact storage utilization. If at some time the user wishes to alter a compiled part of the system, he need only supply an updated interpretive version of it. Then at his convenience he may

compile his new version if he finds it satisfactory. Alternatively, he may try another version of his own or return to the original version. By using selective and incremental compilation we retain the system's generality and open-endedness without yielding to the inefficiencies of interpreted code.

CHAPTER 2 02:38:08 08/04/75

2 Problems in Graphics

In this chapter we discuss the problems which are encountered in graphics and point out that subset which we have attacked. We describe some of the kinds of capabilities required by a sophisticated user of a computer design system. All current systems incorporate some subset of the capabilities we list here, but to our knowledge no system (including curs) currently incorporates all of them.

2.1 Display and computation

Two general requirements are placed on graphics systems. These are: they must provide for the calculations necessary to represent the structures being modeled -- structural analysis, population density, heating requirements, traffic flow, what have you; and they must allow for the efficient display of the objects being designed. Although these two requirements are independent, they are linked at the data structure level. A set of data structures must be available which lends itself to both the requirements of interactive display and to the calculation of the many parameters involved in the simulation proper.

Two basic approaches are available: represent the graphics information with one data structure and the compuational information with another, providing a mapping between the two; or, represent the information for both in a single data structure, including the appropriate selection mechanisms. The former approach leads to a proliferation of data structures and large amounts of duplicated information; the latter approach does not. To obtain information from data structures of the former approach does not usually involve traversing the topology of the model, whereas obtaining information from data structures of the latter approach usually does. This situation reflects one of the basic trade-offs between storage and speed. The greater flexibility of the single data structure approach has led to our adoption of it in GRAPL.

2.2 Hidden surfaces, textures, colors, shadows

The suppression of hidden lines and surfaces is one of the obvious requirements of usable graphics systems. A designer must not be encumbered by the display of information which cannot be seen in the real world. On the other hand, it should be possible for him to request such information if he so desires. The current implementation of GRAPL does not give the user the ability to remove hidden

lines and surfaces interactively. However, the addition of this capability as well as the addition of textures, colors, shadows, etc., would be straightforward and is discussed in Chapter 8.

In many applications it is important to see the actual textures of surfaces, how a wall siding appears in full sunlight, or in diffuse light, etc. The design process should proceed in full color if the designer so requests. Color displays are currently on the market and will be relatively inexpensive in the near future. Primarily because of cost, design is done currently either in black on white or in white on blue. Computer graphics will make color design less expensive.

In architectural design it is occasionally important to determine the effect of shadows on the environment. This is especially a consideration in designing buildings adjacent to other structures. The display of shadows and the calculation of their effects on the heating and ventilation requirements of a building is currently available.

2.3 Visual effects

During the design process one frequently needs to

view the object being designed from various angles and from several points of view. If the object is a building, this may mean viewing it from fifty feet, or one hundred feet, or a quarter of a mile away. It may be advantageous to get a "bird's eye view" of the building. One should be able to describe these vantage points in a simple manner, save their descriptions, and then see one's building from each vantage point at will. Better yet, one should be able to view the building from several positions simultaneously.

Also, one should have the ability to project the object onto any given plane. This would allow the creation of the normal orthographic projections which some designers find useful.

Stereoscopic views also are feasible. Various techniques exist for the creation and viewing of stereo information; these should be available to the designer if he wishes to use them.

The generation of perspective views of objects should be the normal display mode. But one should allow for the normal mode to be changed to elevation, one-, two-, and three-point perspective, in addition to any other views the designer may request.

In some applications the generation of exploded view pictures will be important. Techniques are currently available for their generation and this capability should be available to the designer.

It may be very important at some time in the design process to be able to zoom in dynamically on the objects being designed. Alternatively, one may find it important to be able to visualize walking down the corridors of a modeled office building, looking through a doorway into an office, looking out a window, traveling down a street, and so forth. These capabilities should be possible.

And, it may be helpful to an architect to be able to pass a section plane through his structure. He could then view an arbitrary cross-section of his building, see the floors, halls, beams, conduits, and so forth.

The production of hardcopy output is a requirement of any usable graphics sytem. This hardcopy may be in the form of microfilm, plotter output, blueprint, etc. GRAPL gives the user control over point of view, zoom, perspective and section plane. We have not implemented stereoscopic vision or exploded views. Our current hardcopy output is obtained via 35mm photographs or via a post-processor to the Xerox Graphics Printer. All the illustrations in this

document were produced on the XGP.

2.4 Control of detail

The control of detail is one of the most important capabilities a design system can offer. The designer should be able to suppress unneccessary detail at all levels. This means he should be able not only to suppress background, foreground or midground information and suppress the generation of subpictures, but also to suppress any objects satisfying criteria which he gives the system. For example, when designing a room, it may be irrelevant to the designer that the wall panelling has a rough texture or that the ceiling soundproofing has randomly sized and shaped holes. If the display of these features interferes with the design process, the designer should be able to eliminate that type of display quickly and easily.

Moreover, most often it will be the case that the designer will be monitoring the cost of a structure, its area, volume, cube foot cost, and other attributes. In most cases it will not be necessary to know the exact values for these calculations, a rough estimate will suffice. The designer should be able to specify that he needs a precise answer when he wishes one, but otherwise not burden the system with detailed calculation (and incidently, most

likely degrade the system response time).

If the user needs to make use of this kind of generality, he must supply the system with not only the selectional criteria involved, but also the gross description or approximation algorithms to be used in place of the exact calculation. The ability to control detail is one of the most important features which GRAPL includes.

2.5 Context and neighborhoods

Perhaps one of the most distressing attributes of most design systems is the fact that at the beginning of each session the designer must ask the question, "Where am I now?" Moreover, when designing several objects in parallel he may be forced to ask this question again and again with each change of object. One should be able to define a context and then return to it at will. A context must contain the totality of the information which represents the current "state of the world." This information includes not only the current objects being designed and the points of view, but also the states of all system variables at that time.

A design system should provide for the easy and natural description of neighborhoods of discourse. A

neighborhood is one kind of context. A neighborhood might be defined as the objects currently being displayed; in that case we speak of a display neighborhood. Or, a neighborhood may be defined as those objects in the model relating to heating, power, ventilation, etc.; in that case we speak of a computational neighborhood. A neighborhood may be considered a logical grouping of information which may be referenced by name, may be displayed, computed with, stored, and retrieved. The efficient retrieval of various neighborhoods is one of the most difficult tasks facing the designer of a graphics system.

Neighborhoods occur within both the GRAPL system and the GRAPL language. In the GRAPL system we have implemented display and computational neighborhoods. In the GRAPL language we allow access variables which correspond to the kind of neighborhood called "state" or "context" in process-oriented environments.

2.6 Curves and surfaces

Often designers are concerned with the creation of new shapes and the ability to describe arbitrary curves and surfaces in a natural way. A good design system should facilitate the drawing of the standard engineering curves and surfaces as well as the freehand generation of forms and

their later least squares, polynomial, spline, or other fit. We have not implemented the ability to create arbitrary curves and surfaces; a description of how it might be added to the GRAPL system using current techniques is presented in Chapter 8.

2.7 Time-varying displays

One important advantage a computer based design system may offer is the ability to generate time-varying displays. We include in this category computer animation and computer art, modeling of dynamic structures, monitoring the time-varying inputs and outputs of a model, and the display of histograms, graphs, and wave-forms representing accounting or other information about the model. This capability has been one of the least exploited in the design field, yet it offers, perhaps, the greatest potential. We have not implemented time-varying displays primarily as a result of the already slow response of our time-shared PDP-10 environment.

2.8 Attributes

To be useful, a design system must be more than merely a drawing or drafting tool. It must be able to create objects and then give these objects various

attributes such as cost, weight, delivery date, tensile strength, and so forth. It is in this manner that the designer may put into the data base the information necessary to run the various application programs which compute total cost, beam loadings, power requirements, etc. GRAPL has complete generality for the specification of attributes.

2.9 Sketching

One important activity of a designer is making "thumbnail sketches." Sketches of various types, views, and complexities are generated throughout the design process. These sketches may lack most of the detail of engineering drawings, may be extremely rough or moderately neat, but they always allow the designer to plan, to try different ideas, to experiment with very small cost. A sketching capability is crucial to any good design system. We must not force the designer or architect to be explicit or rigorous in his expression. He must have the freedom of creativity. Therefore, our system gives him the means by which he may refine his ideas incrementally and slowly evolve his final plan.

2.10 Defaults and a sympathetic environment

Interacting in a "sympathetic" environment is one of the more important capabilities of an interactive system. It is significant that little research has been done in this area of human factors and that computer systems have remained correspondingly hard to use by the novice or non-computer professional. One should be able to design a system according the user's specific requirements, establish default conditions which remain in effect, and define an environment suited to the requirements of the project at hand. The user should be able to give advice to the system as to what things are important and unimportant, which conditions must be enforced rigidly, which (at least temporarily) may be ignored, and so forth.

The user should not be burdened with having to learn the whole repertoire of system commands nor learn all the features the system provides in order to make good use of it. He should be able to interact with as little or as much of the system as he wishes.

2.11 How to represent knowledge

The representation problem is one of the most difficult facing the designer of any system. How are the various features of the problem to be represented? What representation will yield the most economical solutions for

most problems?

There are at least four aspects to the representation problem. First, there is the question of representing the problem itself. This includes modeling the various parameters of the situation, the interactions between them, the specifications of size, position, cost, and so forth.

Second, there is the representation of the system's knowledge. This include what the system knows and what it knows how to do. The system requires such a self-model in order to be able to respond "The following information is required before calculations may be completed: ..."; or, "This calculation will cost approximately $..., and take N hours. Are you prepared to wait?"

Third, there is the relationship of local and global knowledge in the system. Local knowledge consists of various details specific to particular aspects of the problem. For example, it includes the arrangement of furniture in a given office or in an apartment building. Global knowledge, by contrast, is information which is significant to the whole building. It might include overall cost, dimensions of the structure, type of foundation, etc. A major obstacle in all large modular systems is ensuring

that the local information does not become global and thereby slow down the overall computation.

Fourth, there is the question of the form the knowledge may take. It may be represented in some set of data structures, or it may be represented in the interactions of a set of programs, or by some combination of the two.

2.12 Data structures

The question of which data structure to use to model the various aspects of a computer system is an extremely important one. Various special-purpose structures have been developed or extended especially for graphics. These include special forms of trees, rings, lists, graphs, and heirarchichal versions of all of these. It is significant that no one structure has been found which satisfies all problems. Rather, it is always a set of data structures which are implemented, each data structure modeling a particular set of features. One of the main contributions of this research has been the development of some new data structures. These are described in Chapter 4.

2.13 A summary of our research

This section presents a summary of our research in tabular form. Table 2.1 shows the most significant parts of the GRAPL system.

Table 2.1 - GRAPL's most significant features

Problem	Previous Solns	Our Solution	Our payoff
Accessing very large data structures	Trees, spheres, linked lists...	Cubes	Very fast; especially for architecture
Modular system; no interference	Very careful system design	Parallel system	Great simplicity and ease of implementation
Controlling the detail of computation	NOT DONE	Specifying approximate calculations	Speed
Give advice to system	NOT DONE	Accept advice as strategies	Ability to modify behavior dynamically
Monitor constraints, functions, variables, (e.g. cost, cube-footage, etc.)	NOT DONE, or with great difficulty	A parallel process	Can easily give user the information he requests

Our approach has several additional advantages, as shown in Table 2.2

Table 2.2 - Additional advantages

Problem	Previous Solns	Our Solution	Our payoff
Control the amount of detail displayed	Clipping after examining objects	Use the cuboid data structure	Speed, do not need to traverse the whole data structure
Move around quickly within some neighborhood in the data structure	No faster than most other motions	Compile the neighborhood	Speed
Representation of objects	Data	Program	Flexibility, power
Define a context, a neighborhood	NOT DONE	Save state var's of the system	Accessibility
Fast hidden line and surface removal	Warnock, Watkins, etc.	Preclip with cubes	Speed Speed
Several points of view simultaneously	Difficult	Start a couple more display processes running	
Flexible system, command language, etc.	Syntax-driven translator	Interpreter with compiler	Greater flexibility without loss of efficiency

The GRAPL system demonstrates several new approaches and techniques. These are summarized in Table 2.3.

Table 2.3 - What's new in GRAPL

Using a DUAL data structure	Cubes and master-instances
Using a PARALLEL system design	Yielding modularity, flexibility, and ease of modification
Giving ADVICE to the system	As strategies for display, computation, etc.
COMPILING a picture	Providing speed and extremely concise representation for a neighborhood
EXECUTING an object	To produce a picture, its electrical or cost characteristics, etc.
SELF-MODIFYING data structure	The cubes automatically partition themselves into subcubes when they become too complex

We have encountered difficulties in developing several aspects of the GRAPL system. These are summarized in Table 2.4.

Table 2.4 - Difficulties encountered

What strategies to provide initially for splitting cubes. How many levels of cubes to have.
How to access uniquely and efficiently the appropriate cubes given a position in space pyramid of vision (visual neighborhood).
How to access secondary storage efficiently.
Determining how much information of what kind to include in the masters and instances for most efficient use of storage and time.
How to maintain good response time in a heavily loaded time-shared environment.

CHAPTER 3 02:38:08 08/04/75

3 Survey of Related Work

This chapter has two objectives. First, we wish to give the reader a certain perspective with which to view our research by presenting some background. Second, we wish to acknowledge the sources of many of the techniques and concepts which we have used.

The basic philosophy behind the design of the GRAPL system has been one of striving for consistency, uniformity, and power. Whenever possible we chose the more general path rather than the more restrictive. Thus, the system has been implemented in a specially constructed language based on a high-level interpreter. The implementor may deal with the system on any of several levels: the user level, the GRAPL language level, the MLISP2 level, or the LISP 1.6 level. This capability is not hidden from the user. The sophisticated designer might well avail himself of some of the facilities present at one or all of the levels.

Much research in graphics has been devoted to the selection of data structures both for the representation of graphical entities and the representation of the elements of

the model. It is in this area that graphics systems usually either succeed or fail. If the choice of data structures is not large enough, if they do not have enough representational power, or if they can be accessed only very slowly, the system must ultimately fail. Or, rather, it will succeed only for the smallest of structures (log cabins and the like). In GRAPL we have provided not only an extremely efficient and powerful set of data structures, but we have provided the mechanisms for easily and quickly altering these data structures to meet a user's particular requirements.

The remainder of this chapter acknowledges those sources which have been most helpful in the development of GRAPL. A comprehensive bibliography on interactive computer graphics may be found in <Po 72a>.

3.1 Architecture, design, and general philosophy

The basic architectural concepts and ideas have come from a variety of sources. The most important of these were Alexander's fine bock, "Notes on the Synthesis of Form" <Al 64>, and Koestler's "The Act of Creation" <Ko 67>. Our system reflects much of the same philosophy as Negroponte <Ne 70> and Frankel <Fr 70>.

Much of the basic system design philosophy is due to Dennis <De 68>, Farley <Fa 71>, and Huffman <Hu 71>.

3.2 Graphics systems

A variety of graphics languages and systems were examined prior to and during the development of GRAPL. This survey included the works of Carr <Ca 69>, Garwick <Ga 69>, George <Ge 71>, Johnson <Jo 63>, Kulsrud <Ku 68a>, Negroponte <Ne 68>, Newman <Ne 71>, Prince <Pr 71>, Sutherland <Su 63>, and Wehrli, et al. <WS 70>. The result of a literature survey in the field of graphics is reported in Pollack <Po 72>.

3.3 Display algorithms

A large number of people have made contributions in the field of display algorithms. These include algorithms for the manipulation of data structures, hidden line and surface removal, the mathematical representation of curves and surfaces, and the generation of pictures of objects illuminated from one or more light sources.

Appel, at IBM, has been active in the first two areas for many years, <Ap 66, 67, 68, 72>. More recently, at the University of Utah, Bouknight <Bo 69, 70>, Kelley <BK

70>, Carr <Ca 69>, Gouraud <Go 71>, Warnock <Wa 68, 69>, and Watkins <Wa 70> have made major contributions. In the area of mathematical representation of curves and surfaces, Coons <Co 67> and Forrest <Fo 68> have developed the most sophisticated representations.

Hidden line and surface algorithms for specific classes of objects have been developed by Galimberty and Montanari <GM 69>, Loutrel <Lo 67b, 67c, 70>, Mahl <Ma 72>, Matsshita <Ma 69>, as well as Warnock <Wa 68, 69> and Watkins <Wa 70>, and others at the University of Utah.

An excellent summary of state-of-the-art techniques for hidden line and surface removal may be found in a recent Computer Surveys article by Sutherland, Sproull, and Schumacker <SS 74>.

3.4 Simulation approach

The simulation aspects of GRAPL have been most influenced by the SIMULA language <DN 66>, <DM 70>, <IM 69>, as well as long and fruitfull conversations with Alan Kay.

3.5 Partial application/incremental compilation

The basic ideas behind partial application and

incremental compilation have been around for a long time. The most pertinent works include those of Lombardi <Lo 67a>, Lombardi and Raphael <LR 64>, Mitchell <Mi 70>, and Sandewall <Sa 68>. Much of the groundwork was laid by McCarthy, et al., in the development of the LISP programming language.

3.6 Artificial intelligence

The artificial intelligence features of GRAPL were most influenced by the LISP, PLANNER, QA-4, and LISP70 languages. LISP is best described by Berkeley and Bobrow, <BB 64>. PLANNER was and is being developed by Carl Hewit at MIT, and is described in <He 71>. The QA-4 language was developed at Stanford Research Institute by Rulifson, et al. <Ru 70, 71>, <RW 70>, <RD 72>. The LISP70 language is under development at the Stanford Artificial Intelligence Project and is not yet well documented. The related languages MLISP and MLISP2 are described in Smith <Sm 70> and Smith and Enea <SE 73>. The work of Terry Winograd <Wi 70> also was influential.

3.7 Syntax

The syntax of GRAPL was most influenced by the work of Smith, Tessler, and Enea in their development of the

MLISP, MLISP2, and LISP70 languages. MLISP is described by Smith in <Sm 70>. Smith and Enea describe MLISP2 in <SE 73>. LISP70 is currently under development and has yet to be described in the literature.

3.8 Semantics

The semantic ideas incorporated in GRAPL come from a variety of sources including Balzer <Ba 67>, Dennis and van Horn <DV 66>, Hewit <He 71>, Reynolds <Re 70>, Rovner and Feldman <RF 67>, Strachey <St 66>, Teitelman <Te 66>, and Winograd <Wi 71>.

3.9 Data structures

A variety of data structuring ideas were valuable including those described by Abrams <Ab 71>, Balzer <Ba 67>, Earley <Ea 69, 71>, Rulifson, et al. <Ru 70, 71>, <RW 70>, <RD 72>, Standish <St 67>, Tou and Wegner <TW 71>, van Dam <VD 71>, Wegner <We 71>, and Winograd <Wi 71>. Our algorithm for detecting the proximity of objects is similar in some respects to the interpenetration algorithm of Carr <Ca 69>.

3.10 Control structures

Control structures have become increasingly important in the design of languages and systems. The most pertinent references are Fisher <Fi 70>, Hewit <He 71>, Reynolds <Re 70>, and Rulifson, et al. <Ru 70, 71, 73>. The (pseudo-) parallel portions of the system were influenced primarily by the SIMULA language. (See Section 3.4 above.)

3.11 Extensible languages

The extensible language features included in GRAPL were most influenced by Berry <Be 71>, Cheatham <Ch 69>, Christensen and Shaw <CS 69>, Jorrand <Jo 69>, and Perlis <Pe 69>.

CHAPTER 4 02:38:08 08/04/75

4 The GRAPL Approach

In this chapter we present a description of the GRAPL system. This discussion includes a description of the implementation of the various data and control structures. A discussion of the philosophy explaining why certain design criteria were established may be found in Chapter 2. This chapter discusses the details of how these criteria have been met. In Chapter 5 we offer the details of the GRAPL implementation language and a description of the GRAPL command language.

This disussion proceeds from the particulars of the implementation to more philosophical considerations.

4.1 A parallel system design

GRAPL is implemented as a set of simulated parallel processes running under a scheduler within the MLISP2 environment. Chapter 5 contains more information on MLISP and MLISP2. The great ease with which GRAPL may be modified, and commands altered, or added to is largely due to this fact. Also, it is only within such an environment

that one may implement such commands as Monitor and Notify (see the description of the GRAPL command language below). This approach yields a sophisticated system without the usual corresponding system complexity. Each GRAPL command is implemented as a process, each body or object is a process, even the 1 Km cube representing the world and all its subcubes are implemented as processes.

4.1.1 A concise modular system

Two advantages of parallel system design are: it provides an extremely concise manner in which to implement the system, and it affords the opportunity to design an extremely modular system.

The conciseness of GRAPL yields a system which is easily added to and readily modified. Almost all commands are implemented in less than a single page of code. The kernal system is just over 10 pages long; the simulation routines occupy only 6 pages of code.

System modularity has three direct benefits: 1) independence of commands from one another; 2) flexibility in command format, alternate forms of a command may coexist, and commands are quickly and easily updated; and, 3) the system may be segmented with a demand overlay scheme -- only

those parts of the system which are currently active need be in core.

4.1.2 A small system

When the size of the GRAPL system is compared to most conventional graphics systems which commonly have thousands of lines of code, the advantages of our approach become even more apparent.

In addition, the GRAPL system with all commands resident in core (a highly unlikely circumstance!) occupies less then 60K PDP-10 words. This figure includes 34K MLISP2 system overhead -- GRAPL itself only occupies 26K.

4.1.3 A powerful and flexible system

The parallel system approach enables us to implement options such as multiple viewports as multiple instances of the viewport process. It also enables us to utilize semi-continuously evaluating expressions (see Fischer <Fi 70>) tc implement commands such as Monitor and Notify.

Additionally, the existence of a scheduler provides the capability for deferring actions until a more propitious

time. For example, if the cube data structure needs garbage collection or some other "housekeeping" chores to be performed, one may schedule this event now for activation at some future time, and thereafter cease to be concerned.

4.2 Duality of data and program

One of the most powerful concepts in computer science is the duality of program and data. One may view all computations as sets of programs interacting with one another with no data whatsoever <Ba 67>, or as a vast data structure with a single access mechanism and no other programs at all.

Many systems have striven to divorce programs from the data upon which they operate. But the most powerful (and intelligent) programs tend to operate not only upon data but upon themselves as well. The utility of a single form of representation for both program and data is apparent. One may view an expression either as a data structure having some value, or as a program which computes the same value.

In GRAPL, all entities are represented in the form of programs. This means that all bodies, objects, cubes, even the visual neighborhood and the world model, all are

programs. We interact with these programs in different ways to achieve different effects (e.g., display, computation of cost, retrieval of attributes, and so forth).

4.3 The GRAPL data structures

GRAPL utilizes two dual data structures simultaneously. These are the cube data structure, which is used to represent the physical modeling space; and the body-object data structure, which is used to represent the elements to be inserted into this space. The cube data structure provides an extremely efficient way in which to represent and access large physical structures. The body-object data structure is composed of two parts: the class of bodies and the class of objects. Bodies and objects are the primitive elements used in the construction of any structure.

4.3.1 The cube data structure

The data structure we present for modeling large structures is the following: We define the working space as a cube one kilometer on a side. We divide this cube into 64 subcubes. Each subcube may be referred to by name or by relative location. (This partitioning of space is not generally available to the user for he has no need to

reference it. To the user, space is essentially continuous.) We compartmentalize the data structure representing our structure into these subcubes. Whenever a given subcube becomes too complex (has too much data structure), we subdivide it into smaller cubes and recompartmentalize its data structure.

Now, to modify a structure, it is only necessary to change those subcubes containing information which have been modified. To lock down a corridor and display what is seen, it is only necessary to examine those cubes along the pyramid of vision for visible surfaces. As advice to speed up the display, we may ask the system to reject automatically all cubes of size smaller than some given volume. (One system default for display is to reject from consideration all cubes whose sizes are more than three orders of magnitude smaller than the current cube. This default also displays a dot if the cube contained visible information, otherwise it displays nothing.)

The data structure within each of the smallest subcubes is the true modeling information for the structure being modeled. This data structure includes structural information, information as to the electrical system, mechanical system, ventilation system, etc. Larger cubes may contain some amount of information which is considered

to be the default data structure if the subcubes are too small to be considered for a given calculation.

The choice of a one kilometer cube as the largest representable space is entirely arbitrary, but we feel it is reasonable in terms of using the system for architectural design. Should one wish to do urban planning, a cube 10 or 20 kilometers on a side would be more reasonable. The choice of partitioning the cube into 64 subcubes is motivated by the following considerations: 1) A small number of levels of heirarchy is essential for fast access of data -- if one must continually traverse an extremely deep structure one will spend too much time in the process, 2) We feel that the resolution for an architectural design system should exterd down to about 1 millimeter. These two considerations yield a scale factor of one million between the largest and the smallest representable objects. Partitioning each cube into 64 subcubes achieves this scaling in 10 levels. We have experimented a little with alternative partitionings but have no conclusive results as to optimality. In Chapter 8 we discuss other partitioning schemes with which it would be interesting to experiment.

It should be noted that the data structure presented here is essentially the three-dimensional analog of the Warnock algorithm. Moreover, both are instances of

the more general "Divide and Conquer Algorithm" which is in use throughout the field of computer science. The Warnock algorithm is performed "on the fly," resulting in a picture on a display device, whereas our analog is continually in operation and results in a speedily accessed data structure.

The term "data structure" as we have used it here should not be taken too literally. What we are really talking about is a dynamic process structure reflecting the current organization of the model. In more conventional systems this corresponds to a data structure.

The cubes are actually realized as instantiations of the class CUBE, the definition of which we now give. (Refer to Section 5.11.4 for a description of the CLASS declaration.)

```
CLASS ('CUBE,'(SIZE,BASELOC,DETAIL,GROSS)),
          '(PROG () ()),NL);
```

where,

SIZE	-	is the size of the cube (a power of 4)
BASELOC	-	is the coordinates of the bottom-most corner of the cube
DETAIL	-	is the detail flag; if on, subcubes of this cube exist; if off, the complete description is contained in GROSS
GROSS	-	the gross description of the contents of this cube (a list of pairs: (object, positioning matrix)
PROG	-	is a dummy program
NL	-	means no process is GLOBAL to CUBE

4.3.2 The body-object data structure

Bodies are the most primitive objects which may be represented in GRAPL. They need have no physical significance. Bodies are generally collections of points, lines, surfaces, and attributes which are to be dealt with as a single entity. Bodies cannot be decomposed in any way. However, they may be altered or redefined.

Objects are collections of instances of bodies. Objects therefore have substructure. This substructure may be examined and modified. An object may be thought of as a collection of bodies and other objects which, while not primitive, may be manipulated in a uniform manner affecting all constituents equally. For example, one might define a block-like structure as a body, and then use several of these blocks to construct a table. Alternatively, one might describe a table from the outset; then one would have a body-table instead of an object-table.

Bodies and objects are both implemented as CLASSes in GRAPL. This provides the flexibility to modify GRAPL's basic data structure at the definitional level.

We give the CLASS definitions for BODY and OBJECT below:

```
CLASS ('BODY,'(BOX,DATUM,NV,NE,NF,V,E,F),
          '(PROG () ()),NL);

CLASS ('OBJECT,'(OBOX,ODATUM,DETAIL,GROSS),
          '(PROG () ()),NL);
```

where,

BOX	-	an enclosing cube specifying the space spanned by the body
DATUM	-	is a 4 x 4 positioning matrix
NV, NE, NF	-	the number of vertices, edges, and faces in the body, respectively
V	-	a list of all vertices of the body
E	-	a list of all edges in the body (a list of pairs of vertices)
F	-	a list of all faces of the body
OBOX	-	as BOX
ODATUM	-	as DATUM
DETAIL	-	a list of pairs (entity name,datum) which comprise the detailed description of the object
GROSS	-	similar to DETAIL, but for the gross description instead

4.4 Splitting cubes

The cube splitting algorithm is used whenever the structure of a cube's gross description becomes so complex that it is worthwhile to partition the cube into subcubes. The algorithm which we have implemented is known not to be optimal. However, it does perform satisfactorily. Other possible algorithms are discussed in Section 8.3.

The algorithm proceeds as follows:

1) We are given a cube to be split (the base cube)

2) Measure the cube's complexity

3) If the complexity is less than SPLT (an integer variable set by the user), return

4) For each object in the base cube's gross description, intersect the object's envelope with all subcubes of the base cube

5) Into each subcube where there is a non-null intersection, insert a description of the object

6) Set the base cube's detail flag to True

Note that so long as the same measure of complexity is used at each subcube level, we need not examine the complexities of the generated subcubes. Moreover, once a cube is split, it never need be split again if the algorithm for object insertion guarantees insertion at the lowest possible level cube. An alternative method would be always to insert objects at the highest cube; then split it, and let the splitting algorithm recursively force the object into the correct subcubes.

The complexity measure we have implemented is simply a count of the number of objects in the gross description. If all objects are of roughly equal complexity, then this is a good measure of the total complexity. Since one commonly constructs relatively simple aggregates of objects at any one time, this rather crude approximation is usually reasonable. Alternative measures of complexity, such as the total number of points in the

object or the number of points plus the number of lines, etc., easily may be incorporated into the system if the user wishes. Other measures of complexity are discussed in Section 8.3.

4.5 Display algorithms

The display algorithm consists of two sub-algorithms: one for the display of bodies and objects, and another for the display of worlds. The simpler of the two algorithms is the one for bodies and objects, and it will be described first.

4.5.1 Algorithm for bodies and objects

Bodies are displayed by generating a set of CRT commands from their internal descriptions. Rather than creating and saving this display file of commands, they are sent to the CRT immediately. This necessitates the recomputation of the commands each time a body is shown, but saves considerable memory. In addition, if one is interested in "walking through" a body, the display file would have to be regenerated for each picture regardless.

Orientation, perspective, point of view transformations, etc. are all computed using the usual set

of matrix transformations. Homogeneous 4-by-4 coordinates are used to represent positioning information.

Objects are displayed by recursive expansion of the bodies and objects in their descriptions. Positioning information at each level of the expansion is used to properly orient each subpart of the object. Recursive expansion proceeds until either the most primitive level (bodies) is reached or level of detail cutoff occurs.

4.5.2 Algorithm for worlds

Display of worlds is based upon GRAPL's cube data structure. The current visual neighborhood is intersected with top-most world cube. If a non-null intersection is obtained, the intersection procedure recursively descends into the cube structure obtaining those cubes with a non-null intersection. When level of detail cutoff occurs, the body/object display algorithm is invoked with the description of all entities within the visual neighborhood as data.

Display of world information is slightly more complicated than for bodies and objects alone because entities may reside within several cubes. This occurs whenever an entity is physically larger than the smallest

cubes used to represent it. The world display algorithm keeps track of the status of each entity sent to the body/object algorithm so as to avoid displaying the same entity several times.

Algorithms for clipping and hidden line/surface removal were not implemented for two reasons: the techniques for accomplishing both procedures are now well-known, and doing either or both procedures would use up valuable core as well as slow down the display process. The addition of both facilities in the form of special commands would be a reasonable approach if one were interested in pursuing it.

4.6 Secondary storage algorithm

GRAPL secondary storage consists of a two-level heirarchy: the usual PDP-10 disk file system and a magnetic tape backup system. The implementation of a more efficient special purpose disk filing system was considered, but it quickly became apparent that the PDP-10 system was adequate for our needs.

4.6.1 Disk storage

GRAPL files are of five types: system commands,

catalogs, bodies, objects, and worlds. In each case the file name extension describes a file's contents. The user requests files by specifying its prefix name alone.

System commands normally reside on the disk. When a command is executed by the user, a check to see if the processing routines for the selected command currently reside in core. If they are not, they are loaded immediately from the appropriate system command files. Whenever a CLEAR command is executed, all extraneous system commands automatically are purged from memory. This yields the maximum amount of storage for display of pictures (at the expense of a small amount of added processing time the first time the user selects a command).

Catalog files contain the names of all currently defined and accessible bodies, objects, and worlds. Entities are described by their textual names. Each time a body, object, or world is created or deleted these files are modified appropriately.

Body files contain the explicit description in terms of lines drawn on vertices of the visual properties of the body plus all associated attribute information.

Object files contain the structural description of

the components of the object, positioning information, and attribute information.

World files contain the complete structural description of an entire world. The description is in terms of the names of objects paired with positioning information. The actual object and body descriptions are not part of this file. Rather, they are loaded automatically when a world (or part of one) is loaded for display.

4.6.2 Tape storage

Tape storage is used primarily for backup supplementing the normal system file backup system. Additionally, it may be used to store arbitrary files of any type. Loading files from tape rather than from disk is relatively automatic.

4.7 Selective incremental compilation

Selective incremental compilation is the ability to select certain entities, compile them, later retrieve their uncompiled form, modify them, and then recompile them.

A common trade-off in computer science is the one between time and space. Incremental compilation is a

mechanism for trading decreased execution time for increased storage requirements.

In GRAPI one may compile any object into a body. The new body's display and computational characteristics will be the same as the object's, but the internal structure of the object will be lost. The old (uncompiled) description of the object always is available for later modification. The actual compilation is invoked through use of the infinity key. One loads or instantiates one or more objects, compiles them, and then has the option of saving them under a name of one's choice.

The cube data structure may be compiled as well. This will greatly increase the processing speed but will sacrifice details of the given cube's substructure. One loads the cube, compiles it, and then uses its compiled form as the new gross description. The detail flag is then turned off.

If one wishes, one may compile the contents of the visual neighborhood as well. This is particularly useful if one wishes to examine a specific neighborhood in great detail.

The compilation process consists of recursive

expansion into primitive descriptions of each element of the object's sutructure.

4.8 Neighborhcods and constraints

A neighborhood is a collection of access paths. We can poke a neighborhood or some element within a neighborhood and either store or retrieve information. For instance, to access we might say (line, fetch, type) and we would get back "A to B, type T."

From this point of view it makes no difference whether "line" refers to a data structure for a line or to a routine to generate the line.

A constraint is a neighborhood with special attributes which are interpreted in a particular way. For instance, "parallel (line A, line B)" defines a neighborhood and additionally attaches the attribute "parallel" to it. A processor continually runs around checking to see if it can satisfy the constraints on the current neighborhood or all neighborhoods.

Neighborhoods (constraints) may be small or large, local or global. The most global neighborhood is WORLD: the largest cube in the data structure. Local neighborhoods

may be the current subcube, the current visual neighborhood, or any computational neighborhood.

4.9 Giving GRAPL advice

The user may give GRAPL various forms of advice. He may give an object both a gross and detailed description. This will greatly speed display, especially if the object is used many times in the current picture.

The user may advise the system not to display objects below some threshold size. And, the user may restrict display to only those objects satisfying some criteria which the user supplies.

Another form of advice the user may supply is in the form of a constraint. He may tell the system to perform (or not to perform) some set of actions only when a constraint is satisfied (not satisfied).

4.10 Giving results in real time

One important thing we can do for a user is to give him results in real time. For example, if an architect is designing a building and he asks, "What is the cost of the structure as it stands now?" the resulting computation could

well take several seconds. Yet, he might easily be satisfied with an approximate answer, if he were allowed the option.

4.10.1 Approximating calculations

We have as the default manner of operation an estimator which will approximate the cost of performing each user request. If the cost is high, the system will attempt to approximate the answer quickly, inform the user of its actions, and queue the computation for background evaluation. It is the responsibility of the user to specify how to approximate those things which the system does not already know how to calculate.

A similar approach is used for display. If the user requests a particularly complex structure to be displayed, the system takes the following actions. It estimates the cost of generating the display. Since in this example the cost is assumed to be high, it approximates the display as best it can by presenting the superficial details and outlines of the structure involved. It informs the user that complete detail will take some amount of time and it puts the display generation task into the background queue.

For any task the system may request advice on how

to upgrade its performance. This advice might be of the form, "To compute the gross cost, sum the costs for each major module. A major module is one occupying over 2000 square feet." Or, it might be something like "For this window, only the outermost structural details are necessary. Delete the interior entirely."

The system continually evaluates the cost of displaying structures when the user is in the process of examining them by zoom, moving down the halls, sectioning, etc. If it finds it more economical to do so, the system compiles the appropriate portions of the structure.

4.10.2 Speeding up display

One option the user has which will enable him to speed up the display of objects is to advise the system of the default appearance of things when viewed from far away. For instance, objects of small projected cross-sectional area will automatically be clipped, but a long I-beam will not. The structure of this I-beam will be unnecessarily complex. It will save processing time if the user advises the system that I-beams, when viewed from greater distances or when some other conditions hold, look like straight lines. The system includes default appearances for all cataloged objects which may be modified by the user if he

wishes.

4.10.3 Compiling pictures

Obviously the cost of compiling a picture depends on the complexity of the picture. But if we were to do very much zooming or wandering around within a building, it would clearly be cheaper to compile the entire building than delving deeply into the substructure of each wing, floor, room, etc.

Any object in the system may be compiled into a body at the user's request. He then may replace the old definition of the object with its compiled form or retain the old definition.

4.11 What is a wall with windows?

The question of how to represent a wall with windows or doors is one which has plagued every designer of a graphic system. Is a wall a solid? If so, then how do we represent windows within the wall? Do we intersect this solid wall with some "negative" space in order to allow room for the window?

Is a wall a space within which we may specify an

interior? This interior may be solid, hollow, partially solid, and may include the specification of a window.

Is a wall a tree structure, with the properties of "wallness" hanging off the top node, and substructure specified as a subtree?

The above approaches all have merit, but they have too many disadvantages to be properly useful in a graphics system. After all, what is our objective? To model the real world? Or, to create a system which reflects enough of the characteristics of the real world so as to be useful? We claim that the first statement is emphaticly NOT our objective. The concepts of solid versus non-solid, space versus non-space, etc. do not have to be modeled in order to arrive at a useful system. We therefore compose our model of three parts: 1) What the real world looks like -- how it appears to our eyes, 2) What the real world is made of -- what are the components of these objects we see: a wall, a door, a plate glass window from PPG costing $ 13.95, etc., 3) How space is partitioned -- which areas are considered enclosures, which are rooms, which are stairwells, etc.

The question of what the real world looks like is purely a display question. It is independent of the model

of the (possibly dynamic) physical system we are constructing. The display process should, therefore, occur simultaneously or in parallel with all other portions of the graphics system. The display process corresponds to the visual semantics of the object we are modeling.

The question of what the world is made of corresponds to the rest of the semantics of our object. It includes how the object reacts to heating, cooling, wind, etc.; what are the object's requirements for power, cost, heating, etc.; what physical laws the object must obey, and so forth. This set of semantics is contained within the attributes of each of the subparts of our model.

The question of how space is partitioned is handled in two ways. First, in some cases it may be by attaching a name (or some other attribute) to some neighborhood which is important to us in a spacial sense. Secondly, it is handled by the cubing process which partitions our whole model space.

So what then is a wall with a window? A wall with a window is a structure within some cube(s) with boundary points, lines (edges) defined on these points, possibly with the addition of surface attributes to some of the resulting surfaces. There is no specific modeling of the property of

"solidness." The non-intersection of "solid" bodies is a constraint which may be locally or globally imposed, but the system will dynamicly determine which structures are solidly intersecting with one-another. This model handles the problem of holes quite easily. (Holes are the generalization of spaces for windows, doors, conduits, passages, etc.) For each hole, we just increase the number of boundary points and the number of edges. And, this model trivially allows us to extend a two-dimensional structure into three-dimensions. We just double the number of boundary points, double the number of edges, and connect all old-new boundary pairs with a new edge as our first attempt at interpretation of this 2-D to 3-D extension.

4.12 What is a "master"? What is an "instance"?

A significant problem in graphic systems design is the definition of masters and instances. The general approach we take is that masters should be viewed as templates which generate instances of a specific form. The structure of an instance is not frozen; it may be altered at will after it has been instantiated. The structure of a master, however, is partially but generally not completely frozen. Masters may be altered only in their unfrozen dimensions.

One creates an instance from a master in the obvious way -- by creating a copy with new variables in each appropriate slot. This is essentially an unfreezing operation; instances have more dimensions of freedom than their defining masters.

Creating a master from an instance is the converse operation -- that of freezing in specific relations into the defining form. For example, if we have created an object which we wish to define as a master wall, doing so freezes in the relations which correspond semantically to "wallness." Having instantiated a specific wall from this master, we may wish to add doors, windows, conduits, electrical wiring, and so forth.

In GRAPL we have realized masters and instances using classes. Both the cube data structure and the body-object data structures are classes. Each subcube which is generated is an instance of the class CUBE. Likewise, each body which the user creates is instantiated as a BODY, and each object which he creates is an instance of OBJECT. The class structures serve as templates (or masters); physical bodies are instances.

The approach we have taken corresponds to the incremental compiler and partial evaluation concepts

originated by Lombardi and Raphael <LR 64>. It additionally reflects the properties of parameter specification and reparameterization of subprograms as applied to languages such as ALGOL. Parentheticly, no "algorithmic language" to our knowledge allows reparameterization of subprograms. One must go to a simulation language such as SIMULA before one can find even static reparameterization (via class, class prefix, and virtual declarations). Or one must go to a truly general language such as LISP, which does allow the full generallity of dynamic reparameterization.

CHAPTER 5 02:38:08 08/04/75

5 Description of the GRAPL Languages

The design of a new computer language should always be approached with some caution. One should ask whether the new language will in fact give the user greater flexibility, more expressive power, more freedom, retain some amount of portability, and be better suited to his particular problems. We have designed the GRAPL languages with these requirements in mind.

We have implemented two languages: the GRAPL implementation language and the GRAPL system command language. The implementation language was developed for the design and implementation of interactive systems for computing with large data structures. It is relatively general-purpose, and a wide variety of systems may be designed and implemented with it. The command language, which forms the user-system interface, was developed to facilitate interactive use of the system. The bulk of this chapter is concerned with a description of the implementation language.

The semantics of the GRAPL system and of the system

command language are implemented in the GRAPL implementation language. The semantics of the GRAPL implementation language are currently implemented in MLISP2 -- a language which has all of the virtues of LISP (and a few of its drawbacks) in addition to some powerful features which lie beyond the scope of most current LISP systems.

MLISP2 is an extension of MLISP -- a language developed by David Canfield Smith <Sm 70> at the Stanford Artificial Intelligence Laboratory as a pre-processor to Stanford LISP 1.6 <QD 72>. MLISP is well documented, and the interested reader is refered to Smith's description. The MLISP2 extensions just recently have been described by Smith and Enea <SE 73>.

In this chapter we will present a complete description of the GRAPL language in addition to the relevent portions of the MLISP2 and MLISP languages.

5.1 The GRAPL implementation language

GRAPL embodies features from several different classes of languages. It includes: parallel process facilities somewhat more general than those available in languages such as SIMULA and SIMSCRIPT; complete generality of control structure as specified by Fisher <Fi 70>; the

flexibility of being interpretive while retaining efficiency through incremental (re-) compilation; and it represents all knowledge in a completely uniform way -- in the form of programs.

GRAPL is tied together via a multiprocess recursive and backtrack control structure. Backtracking is more or less a la PLANNER <He 71> and LISP70. States and control points are established with each decision. Backtracking is much more general than pure recursion but should not be used in place of recursion or iteration. Sussman <Su 72> has more to say on this point.

More than one portion of GRAPL may be executing at any one time. We admit coroutines and parallel routines. Sequential control is implicit within a given process. Parallelism is implicit among the several processes which may be activated, passivated, terminated, and so forth. Sub-processes are processes whose execution is monitored by a parent process.

5.2 Constants

GRAPL includes three forms of constants; numbers, quoted expressions, and strings.

5.2.1 Numbers

Numbers may be of two types -- integer or real. Integers are either signed or unsigned and must lie in the range: 0 <= K <= 2**16.

Real numbers are either an integer followed by a decimal which is followed by an integer, an integer followed by an exponent, or some combination of the two. Both the number and the exponent may be signed. Reals must lie in the range: 0 <= ABS(K) <= +/- 2**35.

5.2.2 Quoted expressions

A quoted expression is a single quote (') followed by an S-expression. This is exactly the same as in LISP.

5.2.3 Strings

A string is a double-quote (") followed by any sequence of characters except % ("); these are followed by a double-quand ote. Strings are primarily used in input/output operations. GRAPL is not designed to be a string processing language (as is, say, SNOBOL 4).

5.3 Identifiers

Identifiers are names for objects of all types. They may be of arbitrary length. Identifiers must begin with an alphabetic character (upper or lower case). The following characters may be alphabetic or numeric.

5.4 Lists, segments, and S-expressions

Lists are formed in the same manner as in LISP and MLISP2, either as '(THIS IS A LIST) or as <'THIS,'IS,'ALSO,'A,'LIST>. The former method creates a list constant; the latter constructs a list each time it is referenced.

Segments are formed by use of the segment operator, slash (/). For example, /'(THIS IS A SEGMENT) and /<'SO,'IS,'THIS> both yield segments. Segments are most useful in pattern matching.

The rules for forming S-expressions are similar to those in LISP. An S-expression is either an atom or a list of sub-lists, each of which is an S-expression. The lists may be formed either as constants (a left parenthesis, followed by the list elements, followed by a right parenthesis) or by use of the list operators (left and right angle brackets).

5.5 Indexing

Indexing is handled in a completely uniform way: the function GET ('(A B C),2) yields B, GET ('((A) (B) (C)),1) yields (A), and so forth. GET is defined for lists, tuples, bags, and sets. Its value on sets is the i'th component of the set expressed in cannonical order.

Moreover, GET and PUT allow extended access in the following way, if

X = '(A (B C (D) E) F G)

```
GET (X,2)          yields      (B C (D) E)
GET (X,2,1)                    F
GET (X,2,3,1)                  D

PUT (X,3,'H)                   (A (B C (D) E) H G)
PUT (X,2,3,'H)                 (A (B C H E) F G)
PUT (X,1,4,'H)        ((A NIL NIL H) (B C (D) E) F G)
```

5.6 Specifying data structures

In GRAPL we specify data structures in the following ways:

F	function call returning element value
/F	function call returning segment value
( )	lists
:( )	tuples
|( )	bags
:( )	sets

A prefixed "/" will force a function to return a segment value rather than an element value. A prefixed "|" will force parallel execution of a function call.

5.7 Binding, function definition, and access

In GRAPL we view binding, assignment, and function definition in a completely uniform and consistent manner. A function is viewed as a value which is a list of the following form: It has the symbol LAMBDA, followed by a list of arguments, followed by the expressions forming the function body. Thus the expression:

```
(SET 'FN '(LAMBDA (X) (CAR (CDR X))))
```

sets the value of the atom FN to the list '(LAMBDA (X) (CAR (CDR X))). Whenever EVAL encounters the atom FN, its value will be obtained; and as its value is a Lambda

expression, argument binding and function evaluation will commence. The LISP functions DEFINE and DEFLIST are both replaced by simple assignment in GRAPL .

Binding and assignment are viewed as two syntactically different mechanisms for achieving the same semantic result. For example,

```
(LAMBDA (X) ()))          'FOO
(LAMBDA (X) (SET 'X 'FOO))  ()
```

both give X the value 'FOO, the first by binding X to 'FOO, the second by assigning X the value 'FOO. (The second example is not quite fair, X is first bound to NIL, then assigned the value 'FOO.)

5.7.1 Rules for function definition

The rules for function definition are similar to those for defining Lambda expressions in LISP, but differ importantly in the area of argument binding.

Argument binding is done in the following way:

1) Argument atoms are paired with their corresponding expressions
2) Expressions and segments are elevated if their corresponding argument atom is prefixed with an exclamation point (!)
3) Argument atoms are bound to their evaluated or unevaluated corresponding expressions according to whether they appear unquoted or quoted in the argument list

Function definition itself is accomplished by assignment rather than by declaration.

5.7.2 Examples of function definition and evaluation

Assume U is bound to 3 and V is bound to '(U U).

The expression	Yields
(SET 'IDENTITY '(LAMBDA (X) X))	makes IDENTITY the identity function of one argument
(SET 'IDENT '(LAMBDA (!X) /X))	makes IDENT the identity function of indefinitely many arguments
(SET 'FCN '(LAMBDA (X) (TIMES X X)))	makes FCN the function: F(X) = X*X
(LAMBDA (X Y) (LIST X Y)) 'A 'B	(A B)
(LAMBDA ('X 'Y) (LIST X Y)) 'A 'B	('A 'B)
(LAMBDA (X) X) V	(U U)
(LAMBDA ('X) X) V	'(U U)
/(LAMBDA (X) X) V	-U U-
/(LAMBDA ('X) X) V	-QUOTE (U U)-
(LAMBDA (!X) X) V	((U U))
(LAMBDA (X !Y Z) (LIST X Y Z)) 'A 'B 'C 'D	(A (B C) D)
(LAMBDA ('!X) X) A B C D	'(A B C D)
(LAMBDA (!X) X) U V U V	(3 (U U) 3 (U U))

5.7.3 Access

In GRAPL the concepts of normal variable access, local and global variables, and free and bound variables have been extended slightly to include values obtainable by access.

Access variables correspond somewhat to OWN variables in ALGOL, but they are process-oriented rather

than procedure-oriented. That is, access variables are those local (OWN) variables which are declared at process instantiation time. They retain their values so long as their process exists within the system; they are incaccessible by any means after their process disappears from the system. Moreover, access variables follow a separate rule for global (or free) reference: Whenever a process is generated, an access variable whose name is GLOBAL is declared. Its value is generally set to the name of the generating process. It may be explicitly set if the user wishes. Then, any reference to an access variable of the form (<alpha> variable) will automatically reference:

1) the current process' OWN variables
2) the OWN variables of the process pointed to by GLOBAL
3) if the variable still has not been found, step (2) is repeated until either the variable is found or the topmost process is reached (in which case an error is reported)

Access variables have characteristics both of static local and global (bound and free) variables such as found in ALGOL, PL/1, and LISP, and of dynamic state variables in a process such as found in SIMULA or SIMSCRIPT. Moreover, in GRAPL the chains of access links may be modified during execution.

5.8 Operators

The GRAPL operators include all MLISP and MLISP2 operators in addition to <alpha>, the access operator. For completeness, we present the following table:

Abbreviation	Function	
*	TIMES	
/	QUOTIENT	
+	PLUS	(ok as a prefix)
-	DIFFERENCE	(MINUS if a prefix)
<up arrow>	PRELIST	(a generalized CAR)
<down arrow>	SUFLIST	(a generalized CDR)
@	APPEND	
=	EQUAL	
<not equal sign>	NEQUAL	
<less/equal sign>	LEQUAL	
<great/equal sign>	GEQUAL	
<epsilon>	MEMBER	
&	AND	
<inverted V>	AND	
|	OR	
V	OR	
¬	NOT	
<alpha>	ACCESS	(as a prefix only)

Parentheses may be used to force the order of evaluation. In addition, all binary LISP functions (such as CAR, CDR, etc.) may be used as infix operators.

A precedence system is used in parsing expressions; the reader is referred to the MLISP manual <Sm 70> for a fuller discussion.

The access operator, <alpha>, was discussed in the previous section.

5.9 Sequential control

Six sequential control expressions exist in GRAPL: GO, IF, FOR, WHILE, UNTIL, and CASE.

5.9.1 GO-expressions

A GO-expression forces an unconditional transfer of control. A GO-expression is the word GO followed by an expression which must evaluate to an atom. This atom must be a label on one of the expressions within the current procedure. Global lables (such as are possible in ALGOL and PL/1) are not allowed.

```
GO LABL;
GO IF A=B THEN L1 ELSE L2;
```

5.9.2 IF-expressions

The IF-expression is the conditional expression in GRAPL. It is formed by the word IF followed by an expression, followed by the word THEN, followed by another expression. Optionally, this sequence may be followed by the word ELSE and another expression. One or more ALSO-clauses consisting of sequences of the word ALSO and an expresion may fcllow the THEN-expression and/or the

ELSE-expression. The semantics of the IF-expression are the same as in MLISP.

```
IF A=B THEN C <- C+1;

IF FINISHED THEN FINALFUNCTION(RESULTS)
     ELSE GO LOOP;

IF PRED THEN I <- I+1 ALSO L <- CDR L
     ELSE I <- 0 ALSO L <- OLDL ALSO GO LOOP;
```

5.9.3 FOR-expressions

The FOR-expression is one of the most powerful expressions in GRAPL (and MLISP). Rather than duplicating the excellent description found in the MLISP manual, we present a list of the capabilities of this expression and give some examples. FOR-expressions allow one to:

1) Increment (decrement) through a numerical range with arbitrary step size

2) Sequence through a list using the first, second, third, ... element

3) Sequence through a list using the whole list, the list minus the first element, minus the first and second elements, etc.

4) Force the FOR-variables to be local to the FOR-expression or use variables global to the FOR-expression

5) Control the manner in which the results of the FOR-expression are accumulated

6) Terminate execution of the FOR-expression at any time

7) Run any number of FOR-variables in parallel and/or nest FOR-expressions.

The following examples illustrate some possible constructions and the results of their execution.

```
Let L = '(A (B) C)

FOR NEW I <- 1 TO 10 BY 2 DO PRINT I;

     prints    1
               3
               5
               7
               9
     returns   9
     uses a local I

FOR J <- 1 TO 999 DO PRINT <I> UNTIL J EQ 4;

     prints    (1)
               (2)
               (3)
               (4)
     returns   (4)
     leaves J set to 4

FOR NEW K IN L DO PRINT K;

     prints    A
               (B)
               C
     returns   C
     uses a local K

FOR K ON L DO PRINT K;

     prints    (A (B) C)
               ((B) C)
               (C)
     returns   (C)
     leaves K SET to NIL

FOR NEW I IN L DO COLLECT PRINT <I>;

     prints    (A)
               ((B))
               (C)
     returns   (A (B) C)
     uses a local I

FOR J ON L DO COLLECT PRINT J;
```

```
prints     (A (B) C)
           ((B) C)
           (C)
returns    (A (B) C (B) C C)
leaves J SET TO NIL

FOR I <- 1 TO 5 FOR J IN L DO PRINT <I,J>;

prints     (1 A)
           (2 (B))
           (3 C)
returns    (3 C)
leaves I set to 3 and J set to C
```

Further examples may be found in the MLISP manual.

5.9.4 WHILE-expressions and UNTIL-expressions

These two forms allow one to form iterative expressions with arbitrary or no specific sequencing control. The WHILE-expression is formed by the word WHILE, followed by an expression, followed by the word DO or COLLECT, followed by another expression. So long as the first expression evaluates to a non-NIL value, the second expression is repeatedly evaluated. The UNTIL-expression is formed by the word DO or COLLECT, followed by an expression, followed by the word UNTIL, followed by another expression. Its execution is similar to that of the WHILE-expression except that the body of the expression in guaranteed to be evaluated once before termination.

```
WHILE NEQUAL(A,B) DO A <- A+1;

WHILE CAR L = 'A DO
  BEGIN
```

```
        L <- CDR L;
        I <- I+1
    END;

DO A <- A+1 UNTIL A=B;

DO BEGIN
        L <- CDR L;
        I <- I+1
    END
UNTIL NEQUAL(CAR L,'A);
```

5.9.5 CASE-expressions

The CASE-expression is similar to the CASE statement of ALGOL. It is formed by the word CASE, followed by an expression, followed by the words OF BEGIN, followed by a sequence of expressions, followed by a closing END. The value of the first expression must be an integer greater than zero and nc larger than the number of expressions following the BEGIN. If the value of the expression is outside these limits, an error occurs. If the value is N, (and is within the limits) then the Nth expression is evaluated and is the value of the CASE-expression.

```
CASE N OF
  BEGIN
      PRINTSTR "N IS ONE";
      PRINTSTR "N IS TWO";
      BEGIN
        PRINTSTR "N IS THREE";
        TERPRI()
      END;
      PRINTSTR "N IS FOUR"
  END;
```

5.10 Backtrack control

Backtracking is accomplished through use of MLISP2's SELECT function. The syntax of the SELECT function is:

```
SELECT <value-expression>
     FROM <identifier> : <domain expression>
     NEXT <successor-expression>
     UNLESS <terminator-expression>
          IN WHICH CASE <final-expression>
```

where, if the phrases are omitted, the defaults are:

```
<value-expression>        =  CAR
<successor-expression>    =  CDR
<terminator-expression>   =  NULL
<final-expression>        =  FAILURE()
```

A simple example of the use of the SELECT function is Floyd's CHOICE function:

```
EXPR CHOICE(N);
     SELECT I FROM I: 1 NEXT I+1
          UNLESS I GREATERP N;
```

5.11 Processes and process control

GRAPI processes are named collections of state variables among which are EXPR (the functional body), PC (the program counter), and GLOBAL (a pointer to the next-most global process).

5.11.1 The GRAPL queues: QUEUE and PQUEUE

GRAPI has two built-in queues, QUEUE and PQUEUE. QUEUE contains the "active" process (the process at the head of the queue), and several "passive" processes (those not at the head of the queue). A "passive" process is one which is on PQUEUE rather than on QUEUE. A "terminated" process is one whose PC is 'TERMINATED; it will be on neither queue after the scheduler has examined it.

5.11.2 Scheduler functions

GRAPI includes three types of scheduler functions: scheduler execution functions, scheduler queue control functions, and user queue control functions.

There are four scheduler execution functions:

INITSCHED(RUNP)

> Initializes the scheduler. If RUNP is NIL, sets QUEUE and PQUEUE to NIL. If RUNP is non-null, the scheduler is called.

INITSCHEDS(RUNP)

> Initializes the scheduler, sets PQUEUE to NIL and sets QUEUE to 'SYSTEM. If RUNP is non-null, the scheduler is then called.

SCHEDULER()

> The actual scheduler: runs processes on QUEUE until QUEUE becomes empty.

SHALT()

> Halts the scheduler. The scheduler may be continued by invoking SCHEDULER directly.

There are six scheduler queue control functions:

SINITIATE(PROCESS,STATE)

Binds the state variables given in STATE, then inserts PROCESS at the tail of QUEUE.

SINITIATEP(PROCESS,STATE)

As SINITIATE, but inserts PROCESS at the tail of PQUEUE instead.

SACTIVATE(PROCESS)

Puts PROCESS at the head of QUEUE regardless of whether it was suspended or passive.

SSUSPEND(PROCESS)

Puts PROCESS at the tail of QUEUE if it was on QUEUE.

SPASSIVATE(PROCESS)

Removes PROCESS from QUEUE if it was there, and puts it at the tail of PQUEUE.

STERMINATE(PROCESS)

Removes PROCESS from QUEUE if it was there, and sets its PC to 'TERMINATED. If it was not found in QUEUE, then PQUEUE is searched.

There are five user queue control functions:

UINIT(PROCESS)

Binds the PC of process PROCESS to 'SYSO and puts PROCESS at the tail of QUEUE.

UINITP(PROCESS)

As UINIT, but inserts PROCESS at the tail of PQUEUE.

UACTIV(PROCESS)

If process PROCESS is in QUEUE or PQUEUE, it

makes PROCESS the next process to be run.

UPASSIV(PROCESS)

Removes PROCESS from QUEUE if it was there, and puts it at the tail of PQUEUE.

UTERM(PROCESS)

Sets the PC of process PROCESS to 'TERMINATED and removes it from QUEUE if it was there, or from PQUEUE if it was there.

5.11.3 Local, OWN, and global variables

There are three types of variables a process may reference. LOCAL variables are those variables which have no value upon process activation and whose values are discarded upon prccess suspension. OWN variables are those which reside within the "state" of a process. The PC (program counter) is one example of an OWN variable. GLOBAL variables are those which reside within the state of some other process. Access to GLOBAL variables is actually unrestricted: any process's variables may be read or written. However, the usual case is only to reference those variables in processes superior to one's own. This is done via a link contained in the state variable "GLOBAL" (and by use of the access operator, <alpha>).

Changes are made to LOCAL variables using the normal EVAL access functions (SET, SETQ, and GET). Changes are made to OWN variables and GLOBAL variables through use

of the functions GET and PUT. These functions are called automatically when variables are referenced in the following manner:

OWN Variables:

```
PROCESS.VARIABLE <- VALUE, or
VALUE <- PROCESS.VARIABLE
```

GLOBAL Variables:

```
(ACCESS VARIABLE).VARIABLE <- VALUE, or
VALUE <- (ACCESS VARIABLE).VARIABLE
```

The system function ACCESS may be abbreviated by the special symbol <alpha>, yielding:

```
(<alpha> VARIABLE).VARIABLE <- VALUE, or
VALUE <- (<alpha> VARIABLE).VARIABLE
```

Access to arrays may be made in the following manner: Assuming ARRAY(BETA, ...) is an array, and BETA is global to the current process,

```
EVAL <(ACCESS BETA).BETA,I>, or
EVAL <(<alpha> BETA).BETA,I>
```

both of which yield BETA(I).

5.11.4 Process definition and instantiation

GRAPL allows one to define processes through use of the CLASS expression, which corresponds to the CLASS statement of SIMULA but is not so restrictive. The syntax of the CLASS expression is:

CLASS(NAME,STATE,BODY,GLOBAL)

NAME	=	the name of the class
STATE	=	the names of the process's OWN variables, or NL
BODY	=	the functional body, or NL
GLOBAL	=	the name of the process global to the current one, or NL

Final arguments whose values are to be NL may be omitted.

CLASSQ(NAME,STATE,BODY,GLOBAL)

CLASSQ is similarly defined, but quotes all its arguments.

Processes may be instantiated by use of the NEW expression. Its syntax is:

NEW(INST-NAME,MAST-NAME,STATE)

INST-NAME	=	the name of the instance
MAST-NAME	=	the name of the master (class)
STATE	=	the initial values for the instance's OWN variables, or NL

Instances may be made of other instances or of classes.

Final arguments whose values are to be NL may be omitted.

NEWQ(INST-NAME,MAST-NAME,STATE)

NEWQ is similarly defined, but quotes all its

arguments.

We give some examples:

```
CLASS ('PATIENT,'(NAME AGE WEIGHT HEIGHT),
     '(BEGIN
          the semantics for a patient
     END));

CLASS ('COMPLEXNUMBER,'(REALPART IMAGINARYPART));

NEW (NEWNAME,'PATIENT,<'SMITH,46,165,68>);

NEW (NL,'COMPLEXNUMBER,<5,3>);
```

The first example establishes the class PATIENT and declares that the four characteristcs of name, age, weight, and height are to be state variables. It then schematically continues with the definition of how a patient is to behave.

The second example sets up a "data" class called COMPLEXNUMBER, having two parts: a real part and an imaginary part.

The third example instantiates one patient. It assumes that NEWNAME will yield the name for this particular patient. It then also associates the pairs: (NAME SMITH), (AGE 46), (WEIGHT 165), and (HEIGHT 68).

The last example instantiates one complex number of value 5+3i.

5.11.5 Process control functions

GRAPL provides two special process control functions, HOLD and WHEN. HOLD is a special case of WHEN, but it is especially useful for doing simulations as it is time-oriented. Their descriptions are:

HOLD(LABEL,AWAKE)

> Suspends (passivates) the current process until time AWAKE (or later), then activates it with PC set to LABEL.

WHEN(PREDICATE,ACTION,PREDICATE,ACTION, ...)

> WHEN takes a series of predicates and actions. When the associated predicate becomes true, the action will be performed. There are no particular restrictions on either the predicates or actions. Note that WHEN will only perform the action once; if it is desired to have an action always performed when a particular predicate is true, the action should issue the appropriate WHEN.

WHENQ(PREDICATE,ACTION,PREDICATE,ACTION, ...)

> WHENQ is similarly defined but quotes all its arguments.

The WHEN expression is a straightforward application of the concept of "semi-continuously evaluating expression" due to Fisher <Fi 70>.

We give some examples:

```
HOLD ('LBL1,CURRENT+EVENTTIME);
WHEN ('CONDITION,'(BEGIN ... END));
```

```
WHEN (PRED1( ... ),ACTION1( ... ),
      PRED2( ... ),ACTION2( ... ),
             ...          ...
      PREDK( ... ),ACTIONK( ... ));
```

The first example suspends the current process until the amount of time EVENTTIME has passed; then it is reactivated.

The second example schematically illustrates a simple use of WHEN. After the predicate CONDITION becomes true, the code in the BEGIN-END block will be executed.

The last example schematically illustrates the use of several predicates and actions. The PREDs are taken to be various predicates and the ACTIONs are arbitrary expressions or function calls.

5.12 Expressions

An expression may be either a simple expression or two or more simple expressions separated by infix operators.

A simple expression may be a block, Lambda expression, IF-expression, FOR-expression, WHILE-expression, UNTIL-expression, assignment expression, CASE-expression, etc. The GRAPL syntax for expression and simple expression is the same as is found in MLISP.

Blocks are formed by the word BEGIN, followed by any number of declarations, followed by any number of expressions, followed by the word END.

```
BEGIN
  NEW X,Y;
  NEW Z;
  X <- CAR (Y <- READ());
  Z <- SUBST ('A,X,Y);
  PRINT <X,Y,Z>
END;
```

5.13 Programs

A GRAPL program is an expression followed by a period. Usually the program is a sequence of expressions enclosed in a block, but single expression programs are allowed.

```
PRINT "THIS IS AN EXTREMELY SHORT PROGRAM."

BEGIN
     NEW I;
     I <- '(THIS IS ANOTHER SHORT PROGRAM);
     PRINT I
END.
```

5.14 The GRAPL evaluator

The GRAPL evaluator, EVAL, is similar in most respects to the LISP function of the same name, but it differs significantly in several important ways.

Table 5.1 - The GRAPL evaluator

GRAPL EVAL	LISP EVAL
Atoms only may have one value	Atoms may have an arbitrary number of values
Atoms may have a property list of indefinite length, with repeated indicators	Same
Function call occurs wherever a Lambda expression is encountered	Function call can occur only just after a left parenthesis
Has a consistent method for the Lambda expression of an atom bound to same	No direct method for obtaining such a value is available
Has a uniform method for function definition and argument binding	Has EXPR's, FEXPR's, etc.; a non-uniform argument binding mechanism
Has a means for elevation to lists	No such mechanism exists
Incorporates backtrack control	Has only recursive control
Incorporates parallel and coroutine control	No such mechanism exists

We present some examples of the evaluation of GRAPL expressions.

The GRAPL Expression	Yields the following
(SET 'B '(LAMBDA ...))	binds B to (LAMBDA ...) ie. gives to B the value (LAMBDA ...)
(SET 'A 'B)	gives A the value 'B
(SET 'A B)	gives A the value of the evaluation of EVAL (LAMBDA ...)
(SET 'A .B)	gives A the value (LAMBDA ...)
(SET 'F '(LAMBDA (X !Y) (LIST X Y)))	gives F the value (LAMBDA ...)
(SET 'Z F 'A 'B 'C 'D)	gives Z the value of F applied to 'A 'B 'C 'D which is (A (B C D))
(SET 'Z (F 'A 'B 'C 'D))	gives Z the same value
(SET 'Z F '(A B) 'C 'D)	gives Z the value ((A B) (C D))
(SET 'Z F 'A 'B 'C F 'D 'E 'F)	gives Z the value (A (B C (D (E F))))
(SET 'Z F 'A 'B 'C (F 'D 'E) 'F)	gives Z the value (A (B C (D (E)) F))

5.15 The GRAPL system command language

The GRAPL system has facilities both for the creation of new objects and for the collection of old objects into structures which we may then save as new objects.

The GRAPL system includes commands for the construction of primitive entities (bodies), the combination of these primitives into more complex forms (objects), and the incorporation of these entities into a world model.

Commands fall naturally into several categories:

1) Control Commands - those dealing with general control functions

2) Drawing Commands - those dealing with the actual drawing process; the creation and manipulation of bodies and objects

3) Attribute Commands - those dealing with attributes given to bodies and objects

4) Monitor Commands - those dealing with the monitoring facilities

5) World Commands - those dealing with the creation and manipulation of world models

Table 5.2 - Command language summary

CONTROL COMMANDS

```
Command              Action

C              -  (top level)  Continue GRAPL system
G              -  (top level)  dead start GRAPL system

C              -  Clear screen, etc.
H              -  reset window to normal viewpoint
M              -  set Mode to
  D            -        Dilate
  R            -        Rotate
  S            -        Scale
  T            -        Translate
N abc          -  sets Name to 'abc
                        WORLD PEN BODY abc (abc . #)
P              -  set system Parameters
  A            -        DA:   Angular constant
  N            -        FVN:  auto-Number vertices
  O            -        ORM:  Order of magnitude
  Q            -        FQN:  Query status
  S            -        DS:   Scale factor
  T            -        DT:   Translation constant
  W            -        FQW:  World query status
Q              -  Query system status
  L            -        List catalog of bodies and objects
  Q            -        Query system variables
  W            -        query World variables
X              -  eXecute a LISP expression
Z              -  Zap!  Terminate run
'              -  reset name to PEN
<infinity> abc -  compile object 'abc
ALTMODE        -  restart numeric input
*              -  escape -- terminate with no action
<alpha>        -  output picture to XGP printer
```

DRAWING COMMANDS

```
Command              Action

D              -  Delete commands
  E # #'       -        Edge joining vertices # and #'
  F # #' #"    -        Face bounded by vertices #, #' and #"
  O #          -        Object #
  V #          -        Vertex #
E              -  new vertex at pen; new Edge
I abc (x y z)  -  Instantiate object 'abc at (x y z)
J # #'         -  Join vertex # to #'; pen at vertex #
```

```
L abc          -  load object 'abc for edit
S abc          -  Saves current object under name 'abc
T (x y z)      -  move pen To (X Y Z)
V              -  new Vertex at pen

0              -  penup; pen to (0 0 0); pendown
#              -  penup; pen to vertex #; pendown
.              -  name for the last vertex created
:;  )(  *-     -  rotate about the X, Y, Z axis by DA
:;  )(  *-     -  translate on the X, Y, Z axis by DT
```

ATTRIBUTE COMMANDS (side effect: perform a load)

```
Command            Action

A abc          -  show Attributes of object 'abc
D abc attr     -  Delete Attribute attr from object 'abc
G abc attr val -  Give attr value val for object 'abc
R abc attr     -  Retrieve attr's value, object 'abc
```

MONITOR COMMANDS

```
Command            Action

M              -  Monitor commands
  M nam expr   -  Monitor expression using name 'nam
  N expr       -  Notify (once) when expression expr is true
  U nam        -  Unmonitor expression 'nam
```

WORLD COMMANDS

```
Command            Action

W              -  World commands:  all refer to current world
  D abc        -  Delete object 'abc in current visual nbhd
  I abc (x y z)  -  Instantiate object 'abc at (X Y Z)
  L abc        -  Load world 'abc and initialize
  S abc        -  Save current world under name 'abc
  U abc        -  Update object 'abc
```

CHAPTER 6 02:38:08 08/04/75

6 Several Examples

In order to demonstrate the power and flexibility of the GRAPL system we present four examples. The first example demonstrates power of GRAPL data structures. We have created a world composed of a set of buildings. As we slowly decend intc the first structure by decreasing the size of the display neighborhood, more and more detail becomes apparent until we reach the (current) limits of the resolution of the system. The ease of representation of highly complex structures, control of the level of detail, and the efficient access and display of large structures are due largely to the manner in which GRAPL stores information about the real world.

The second example demonstrates the power of our parallel approach to the design of interactive systems. We have simulated a small operating system and display graphically various parameters of the model. The user may examine the queues which arise, modify the model's parameters, completely change the structure of the model, and display his results. Some of the system's characteristics are displayed using histograms which

dynamically reflect the current status.

The third example demonstrates the addition of a small package for the display of algebraic functions of the form y = f(x). The graphing package generates a body which contains not only the representation of the function, but the coordinate axes as well. The form of the equation, scale, and various other parameters may be set by the user interactively.

Our fourth example is a projection of how GRAPL might be used by an architect and his client in the design of a house. The GRAPL system in its current form could be used in the design; however, it most probably would be more economical for a few modifications (additions mostly) to be made first so as to "tailor" GRAPL to the requirements of the architect and his client.

6.1 Walking through a building

As our first example, we present a demonstration of the power of control over the level of detail in the presentation of pictures which vary over a wide range of magnitudes.

In Figure 6.1.1, we see the gross description of a

skyscraper modeled after the Transamerica Building in San Francisco. We are about 1 km away from the building, at about 250 meters elevation. The level of detail is set at one.

Figure 6.1.2 shows us zooming in on the 30'th story. The visual neighborhood has been set so that only the front faces of the building are retrieved. Detail remains set at one.

Figures 6.1.3 and 6.1.4 show use zooming still closer. In Figure 6.1.5, we modify the level of detail to two. Thus, the interior room closest to use now becomes visible. Not all of the contents of the room are visible, however, because the visual neighborhood currently extends only just beyond the desk and chair.

Figure 6.1.6 shows us just inside the physical boundaries of the room with all contents visible. We begin to approach the desk in chair in Figure 6.1.7.

Getting closer still, Figures 6.1.8 - 6.1.10 show us concentrating our attention on the desk.

Figures 6.1.11 and 6.1.12 zoom in on the second drawer on the left side of the desk. In Figure 6.1.13, we

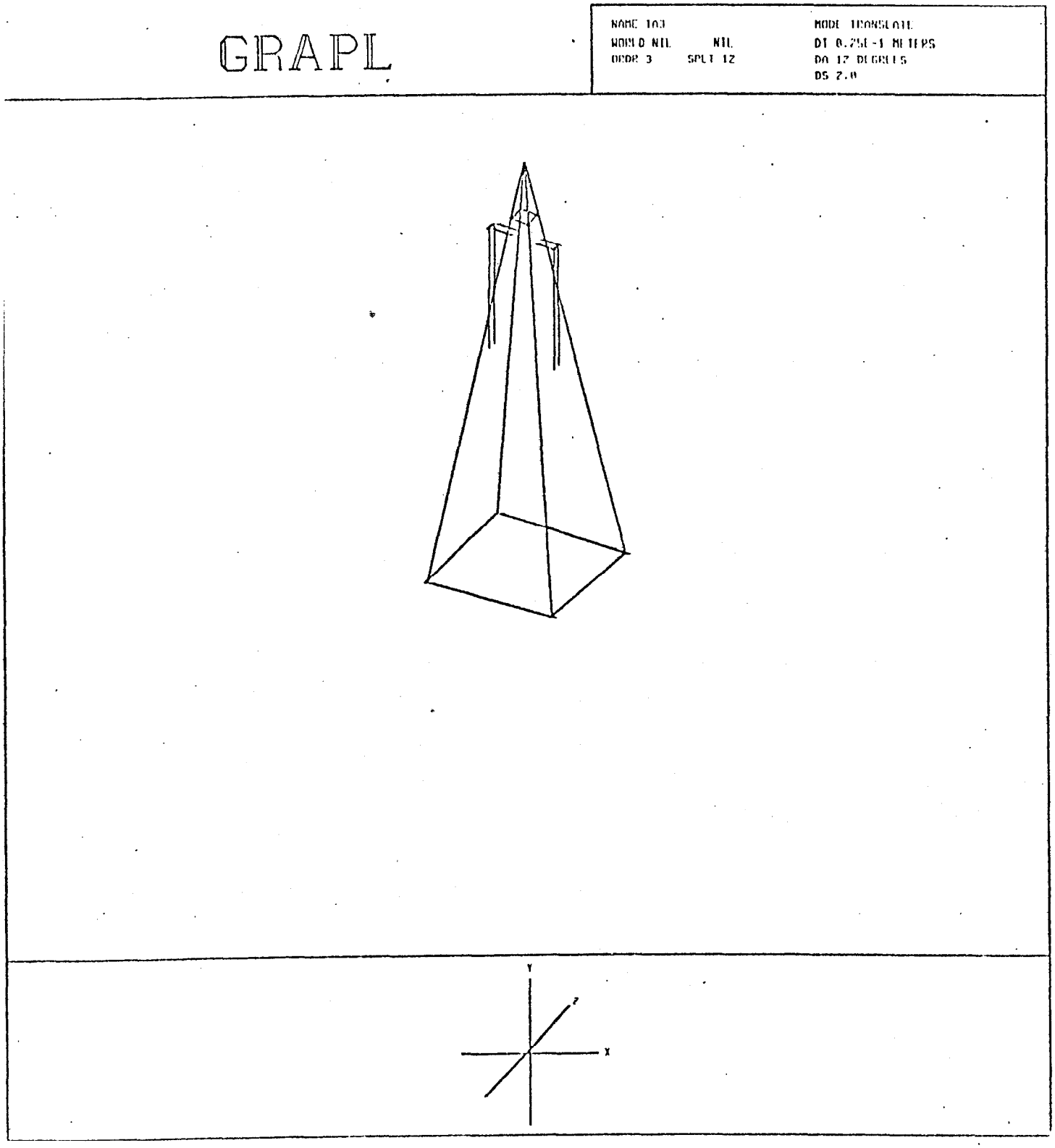
increase the level of detail again, and discover that the second drawer has a cube within it.

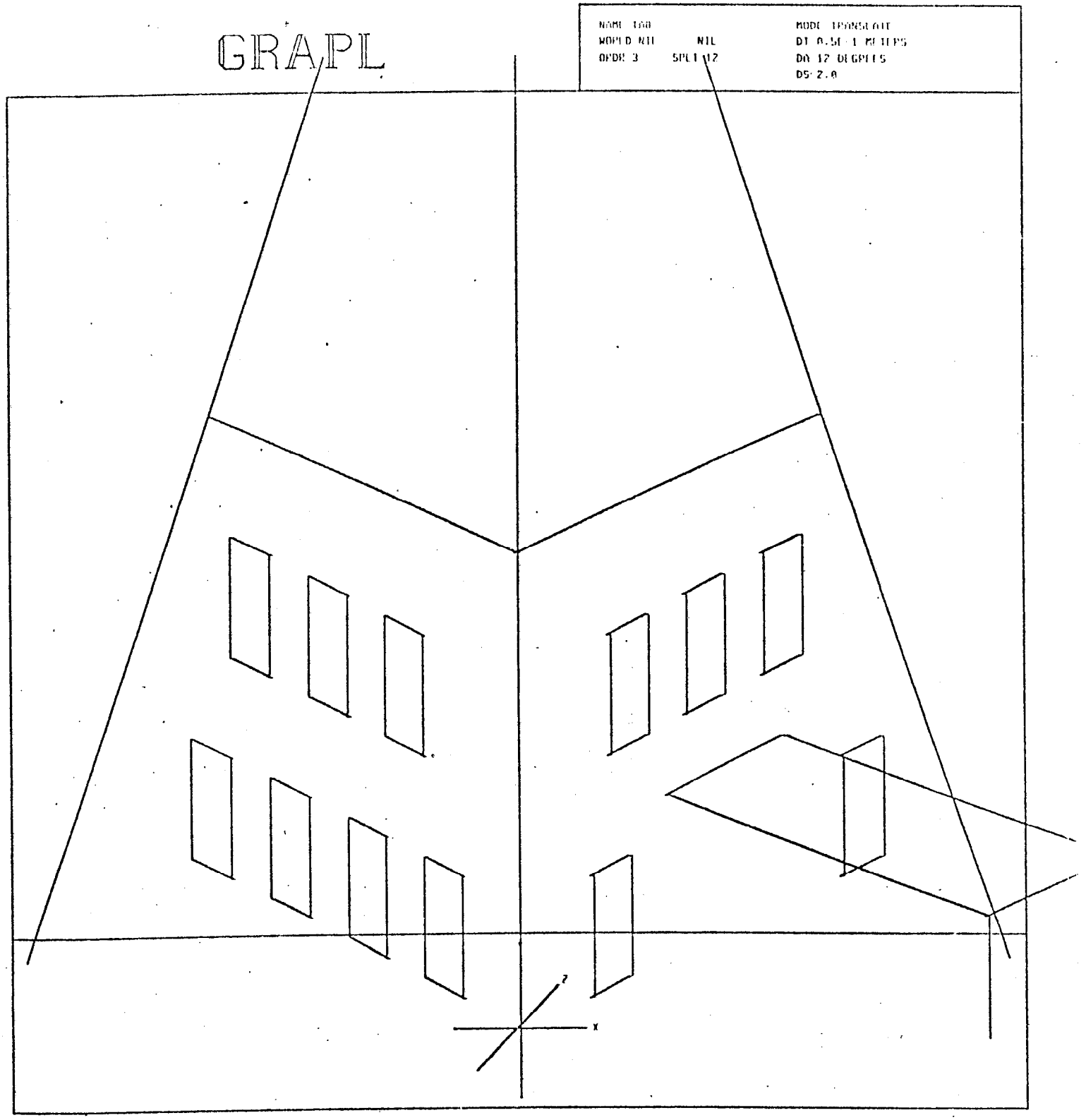
Figure 6.1.14 shows us closer still, and in Figure 6.1.15 we discover a pyramid within the cube. The cube is 6cm on a side. The pyramid is 2cm on a side.

Figures 6.1.16 and 6.1.17 show us getting closer still. At this point the visual neighborhood is a 10 cm cube. Increasing the level of detail again, we see the word "GRAPL" within the pyramid.

We have traveled over the range 1 Km - 10 cm, four orders of magnitude. It is important to realize that all of the data would have been accessed and displayed if the level of detail were tc have been increased at any time. Due to the physical size cf the objects involved, however, most of the time the cube, its pyramid, and the word "GPAPL" would have been displayed as a single point (although all of their internal structure would have been there).

Figure 6.1.1 1' kilometer away




GRAPL
NAME TOO
WORLD NIL NIL
OPDR 3 SPLT 12
MODE TRANSLATE
DT 0.5E-1 METERS
DA 12 DEGREES
DS 2.0
Z
X

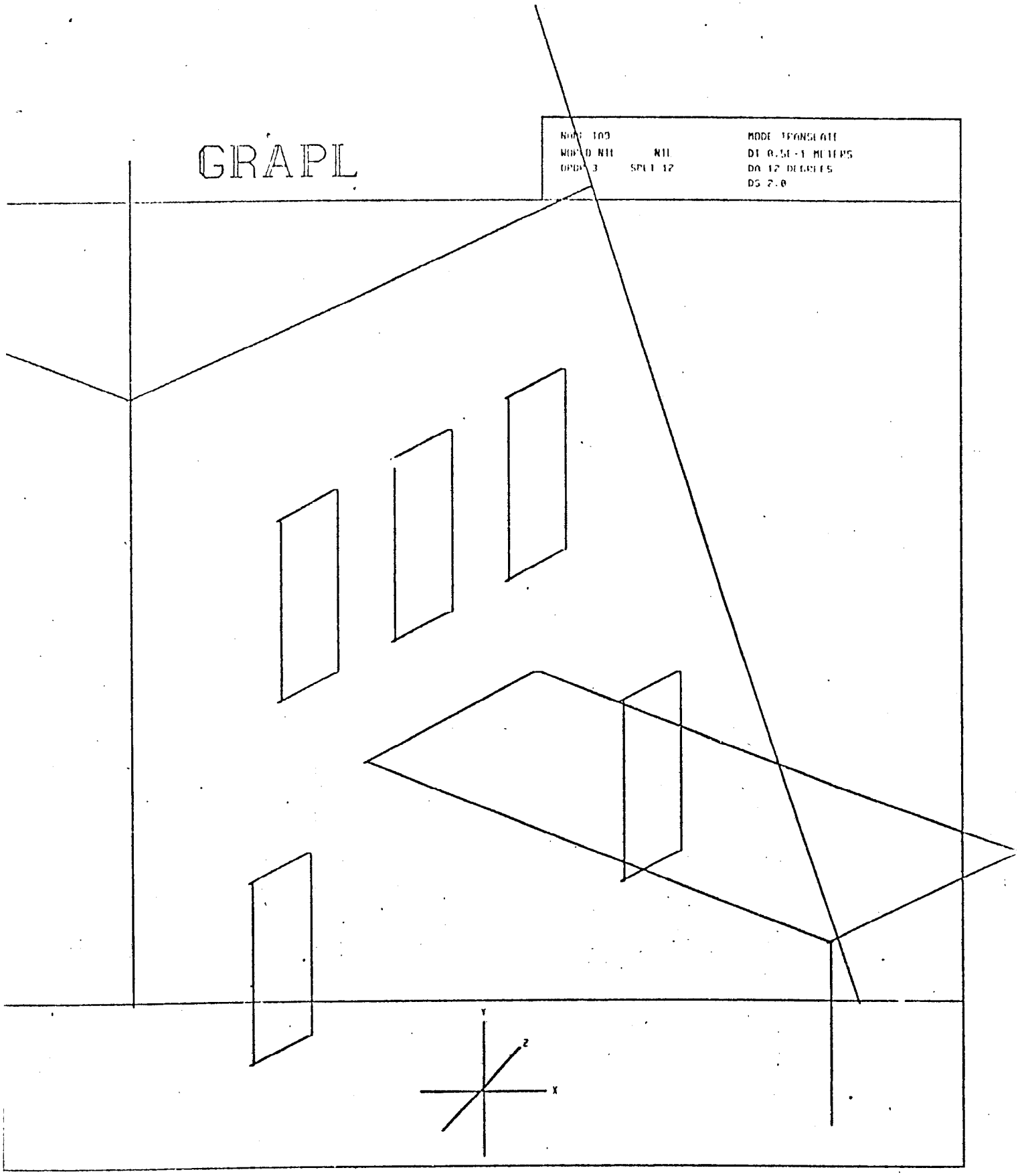

GRAPL
MODE TRANSLATE
DT 0.5E-1 METERS
DR 12 DEGREES
DS 2.0
Y
Z
X

Figure 6.1.4 The 30'th story, still closer Page 101

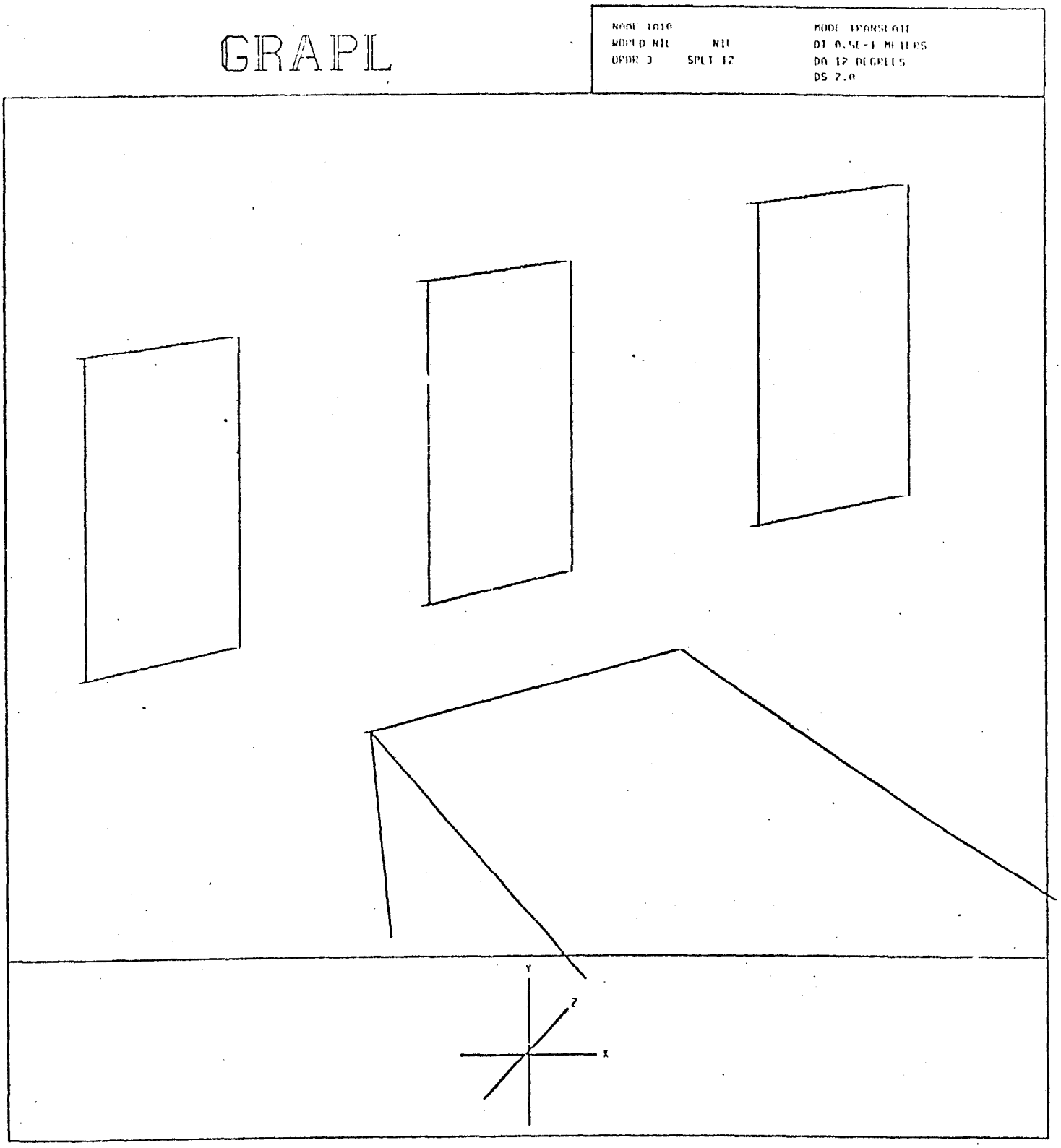


Figure 6.1.5 Increasing the level of detail

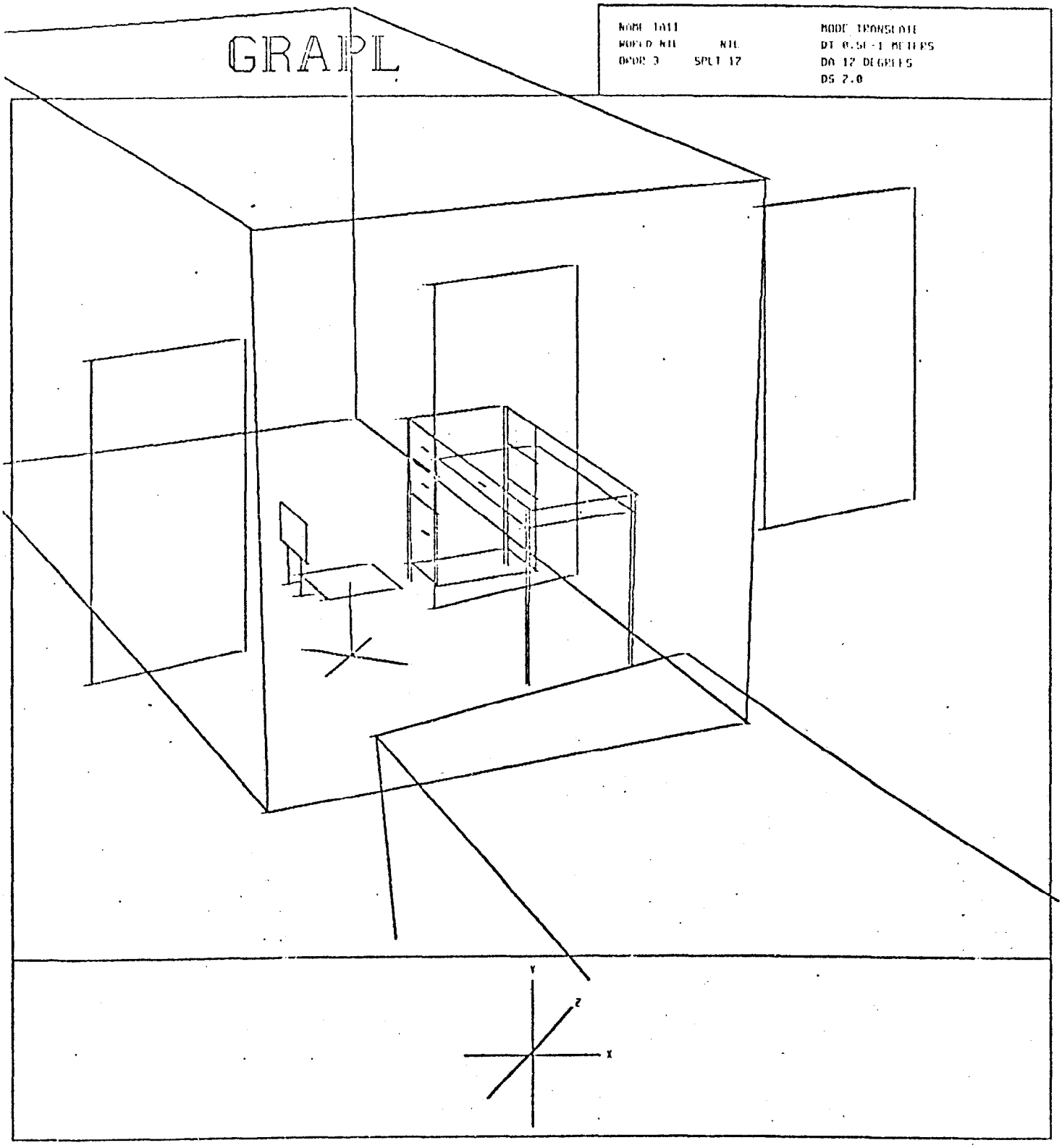
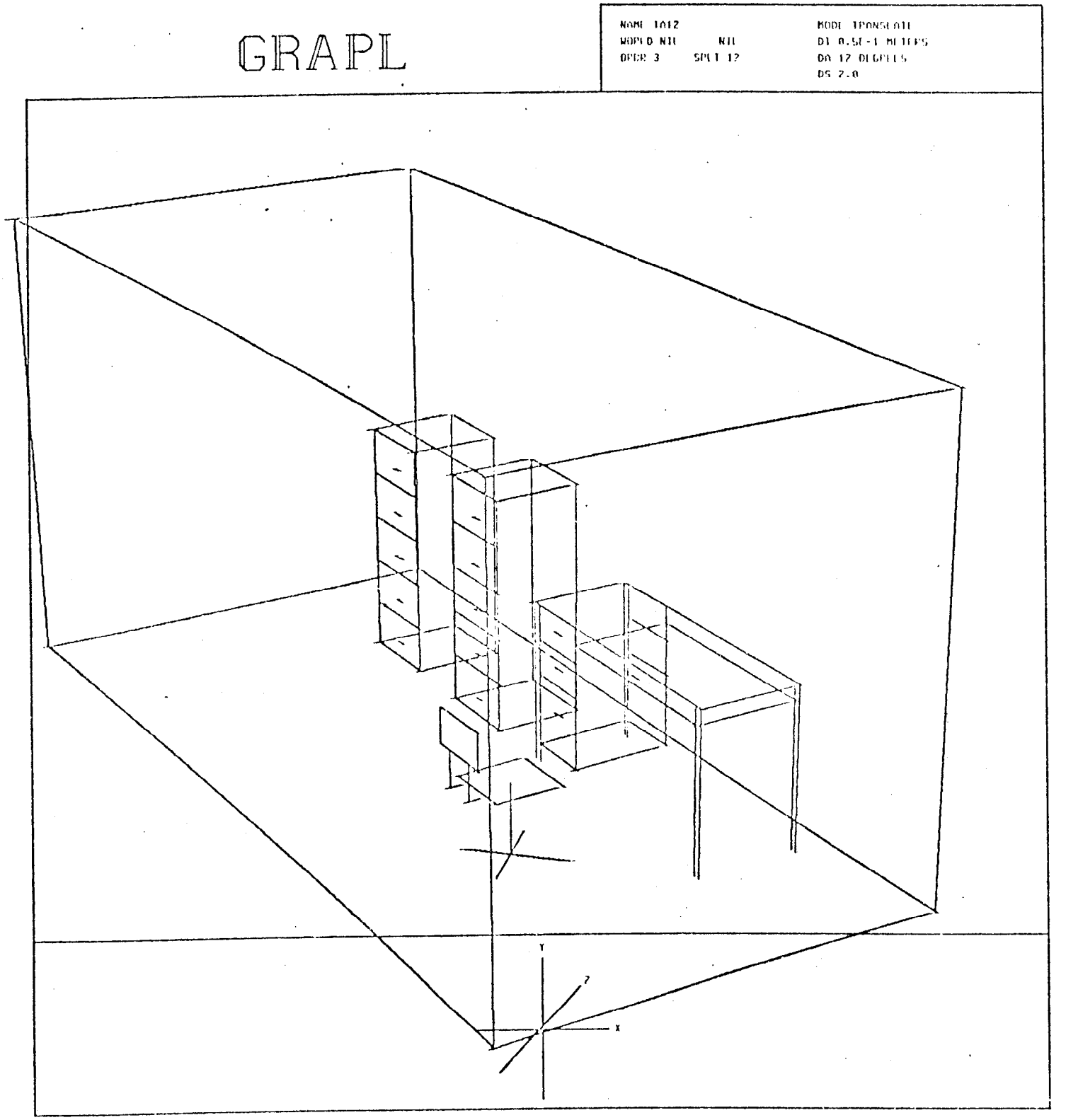


Figure 6.1.6 Just inside the building



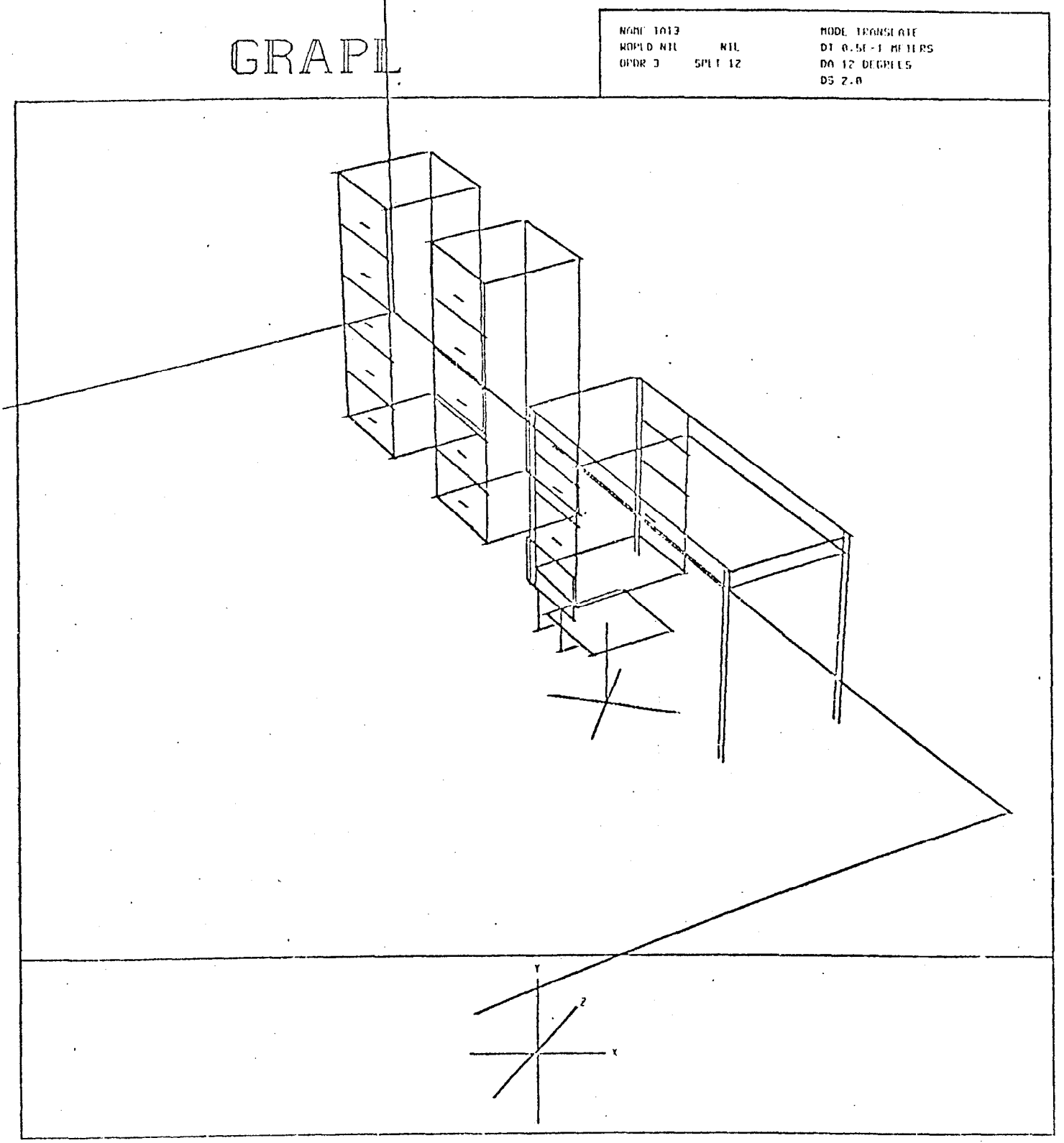

GRAPL
NAME T013
MODE TRANSLATE
WORLD NIL NIL
DT 0.5E-1 METERS
OPDR 3 SPLT 12
DA 12 DEGREES
DS 2.0
Y
Z
X

Figure 6.1.8 A lateral view

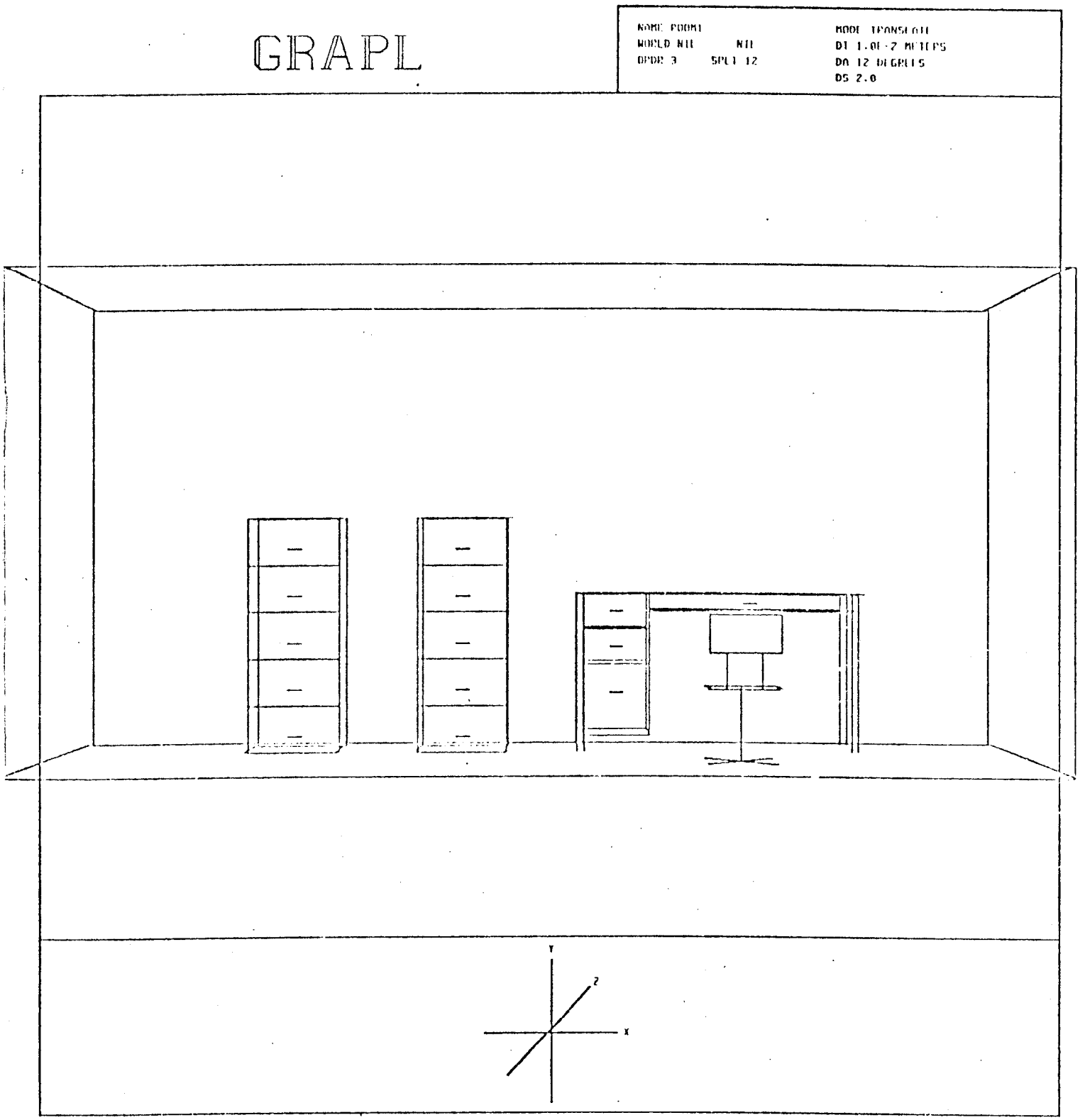


Figure 6.1.9 The file, desk, and chair

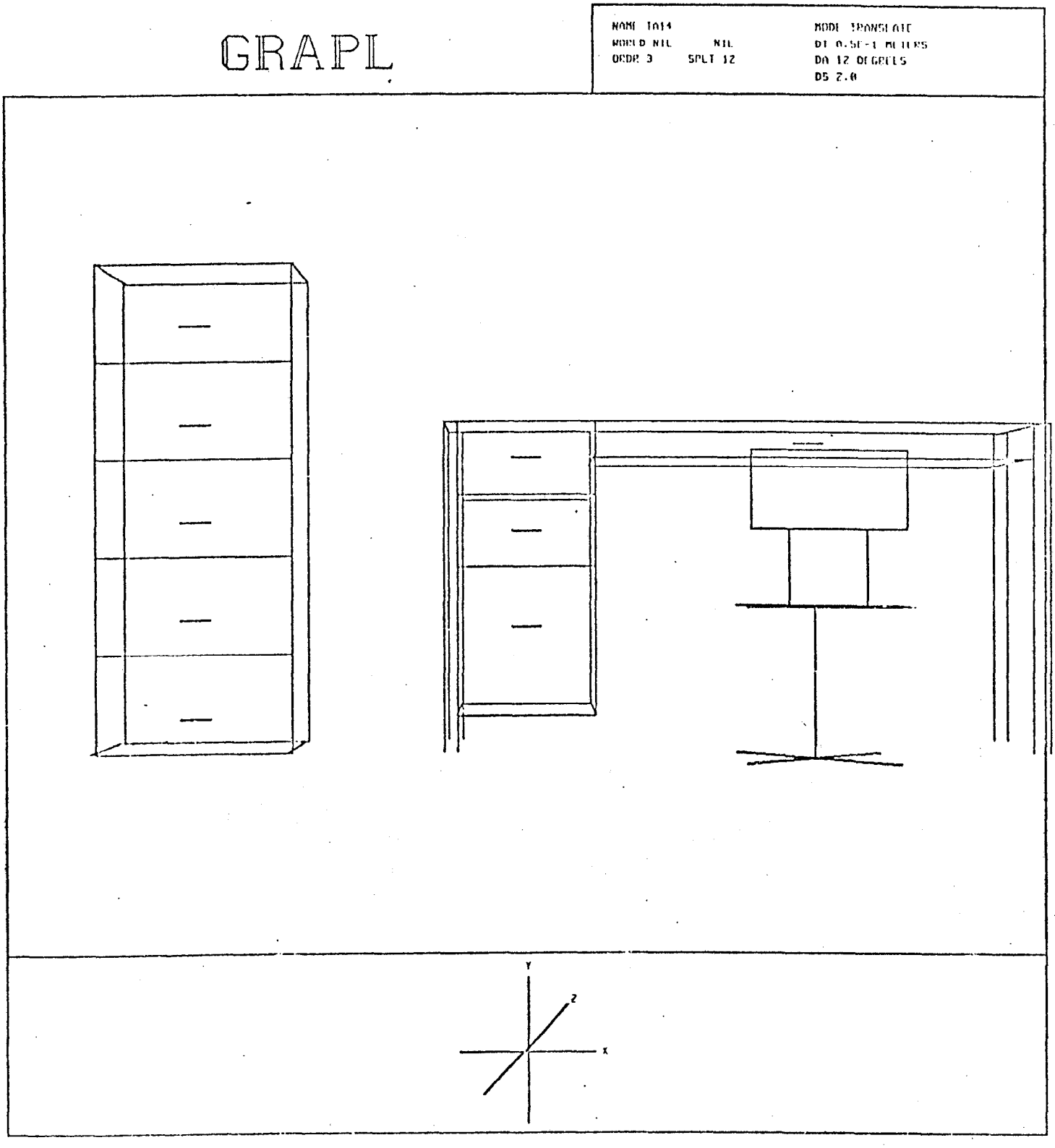


Figure 6.1.10 The desk and chair alone

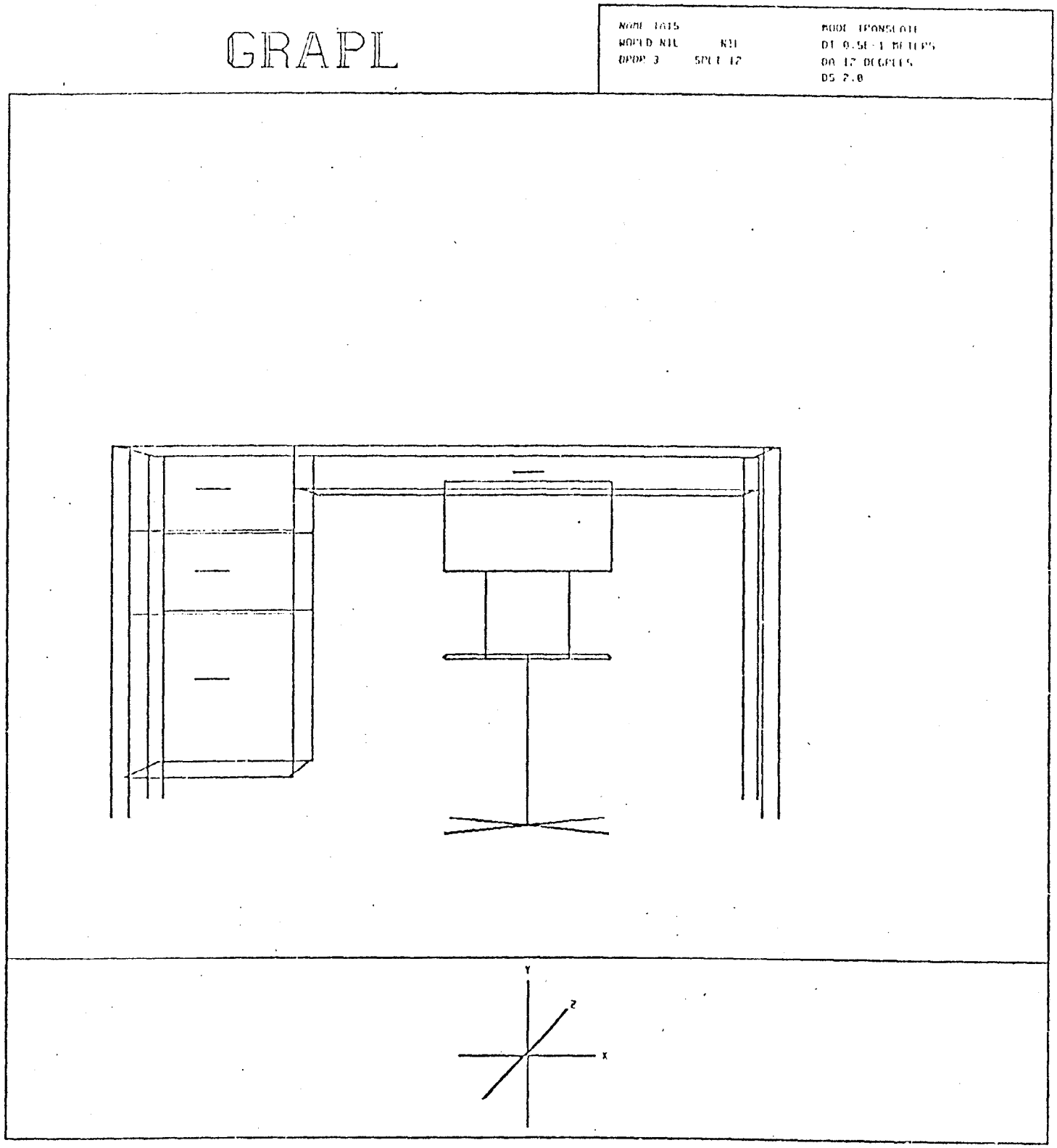


Figure 6.1.11 Zooming in on the left

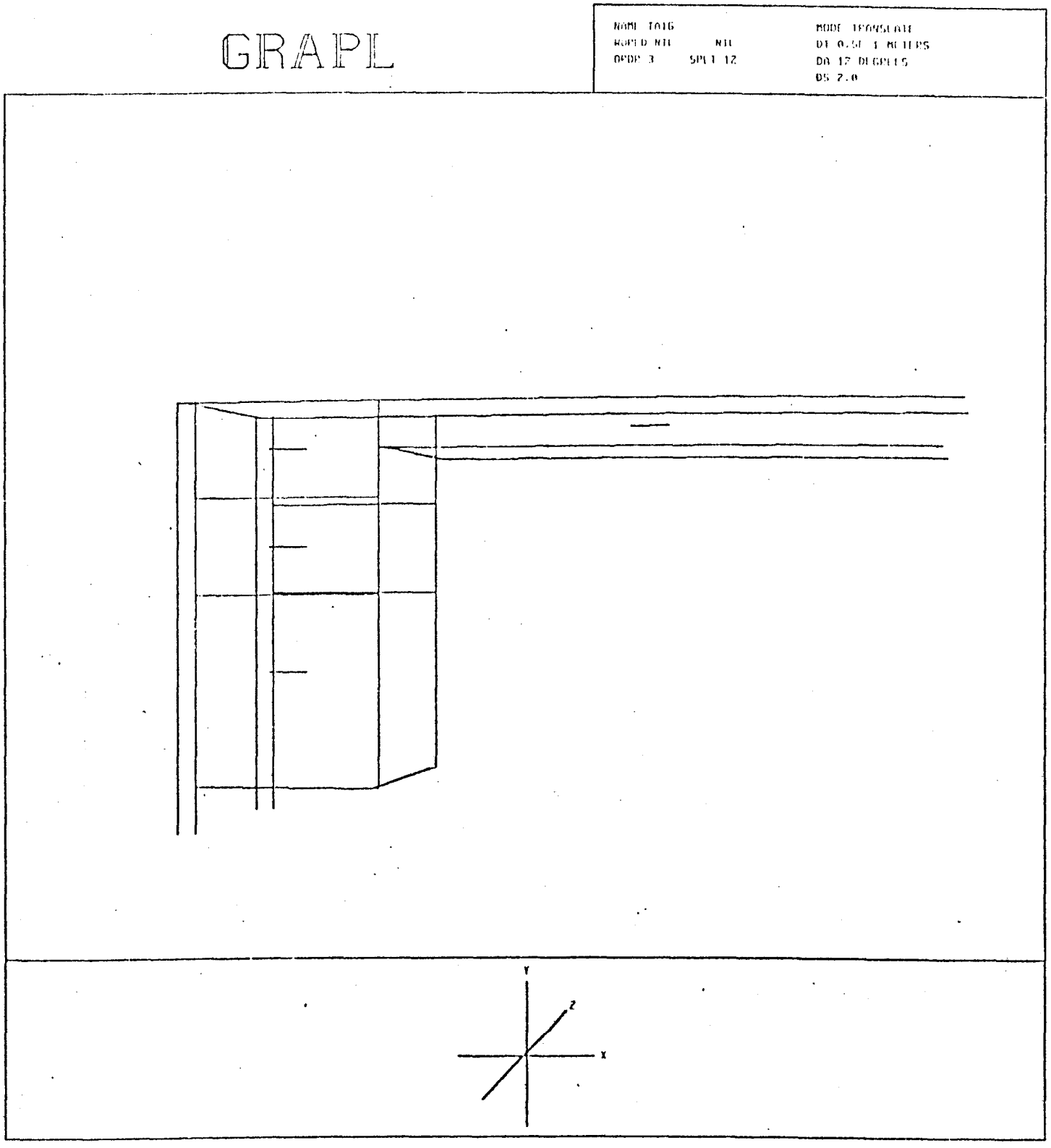
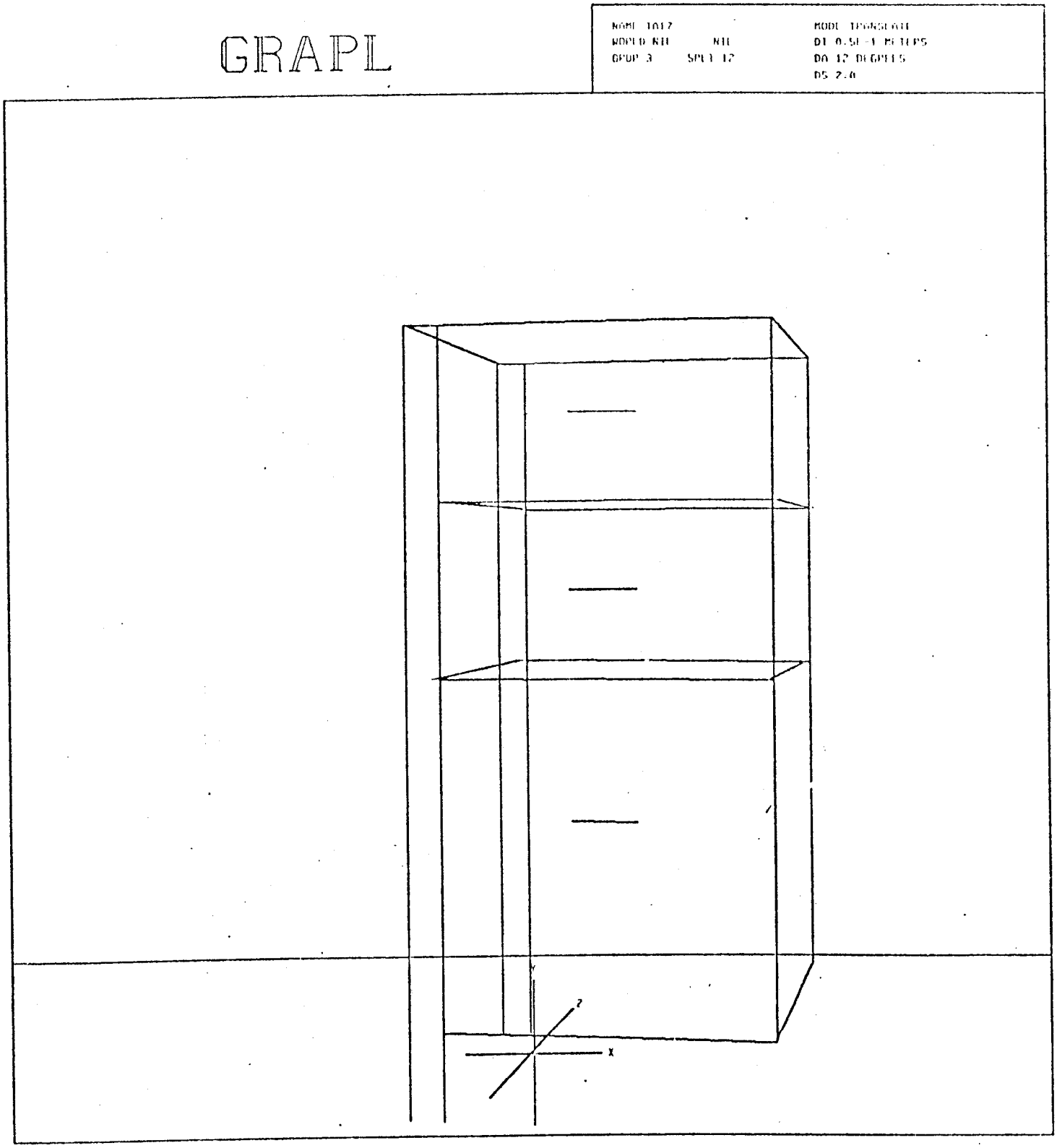


Figure 6.1.12 The three drawers



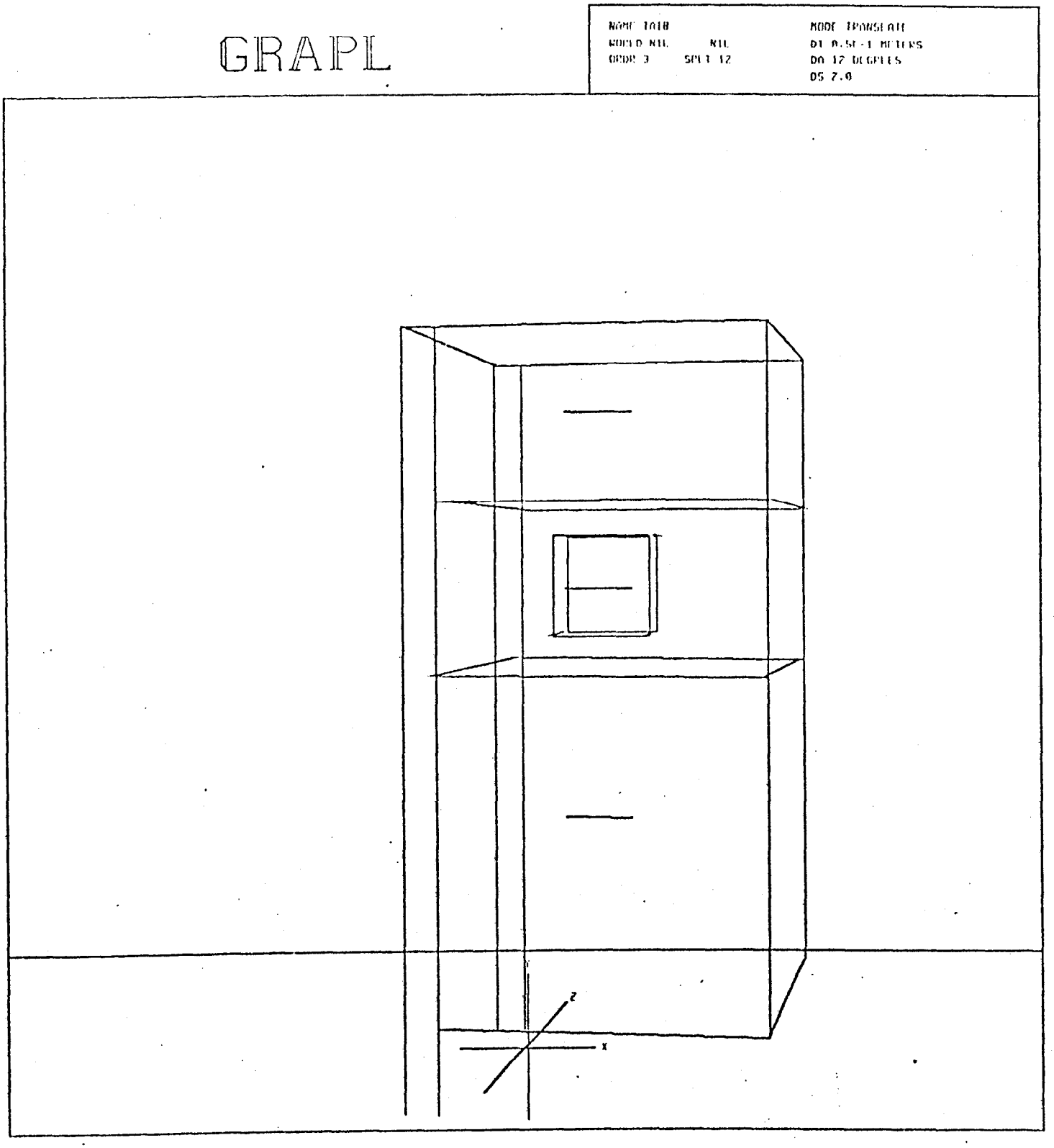

GRAPL
NAME IAIB
WORLD NIL NIL
ORDR 3 SPLT 12
MODE TRANSLATE
DI 0.5E-1 METERS
DA 12 DEGREES
DS 2.0
Z
X

Figure 6.1.14 Closer still

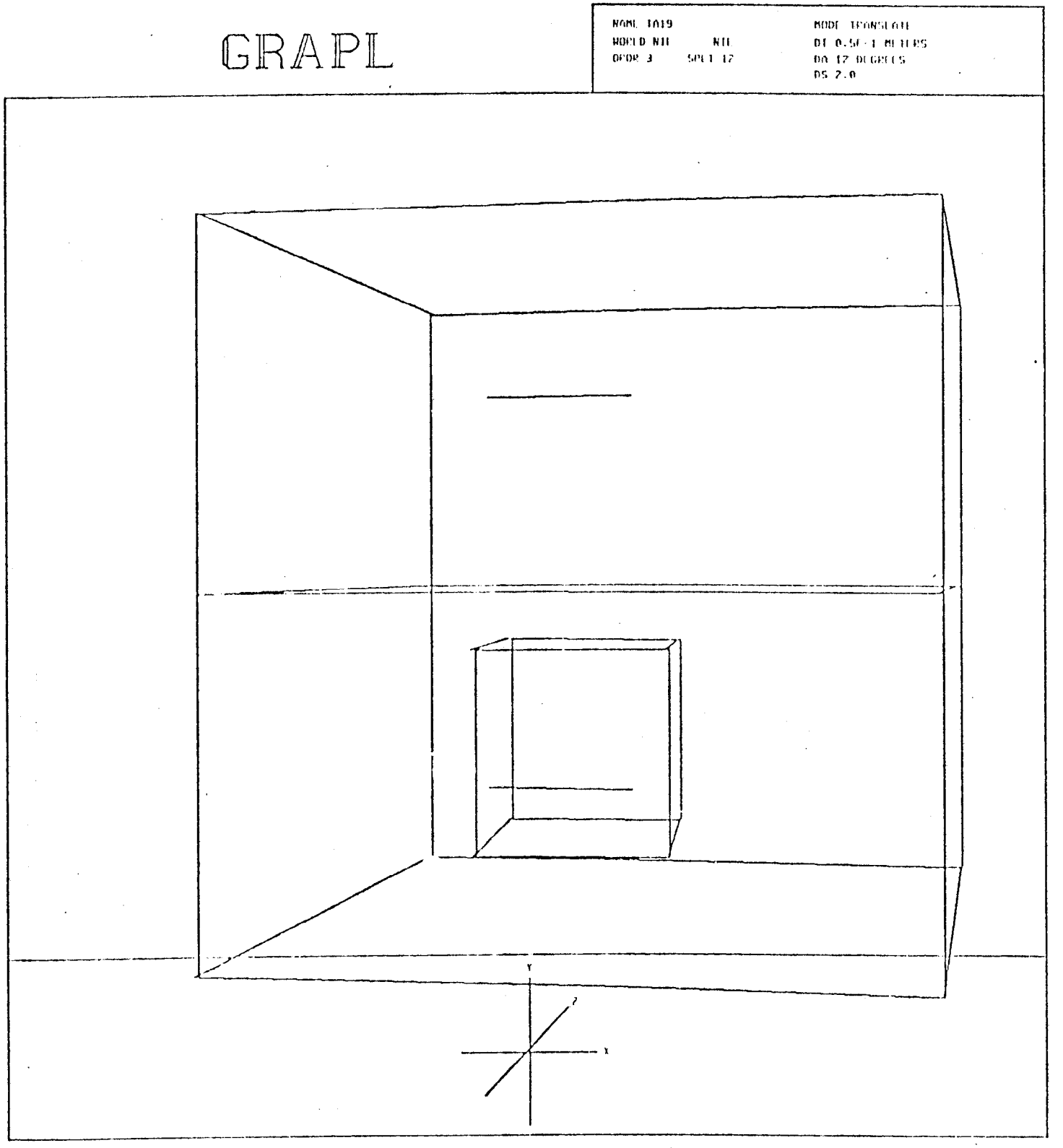


Figure 6.1.15 The pyramid appears

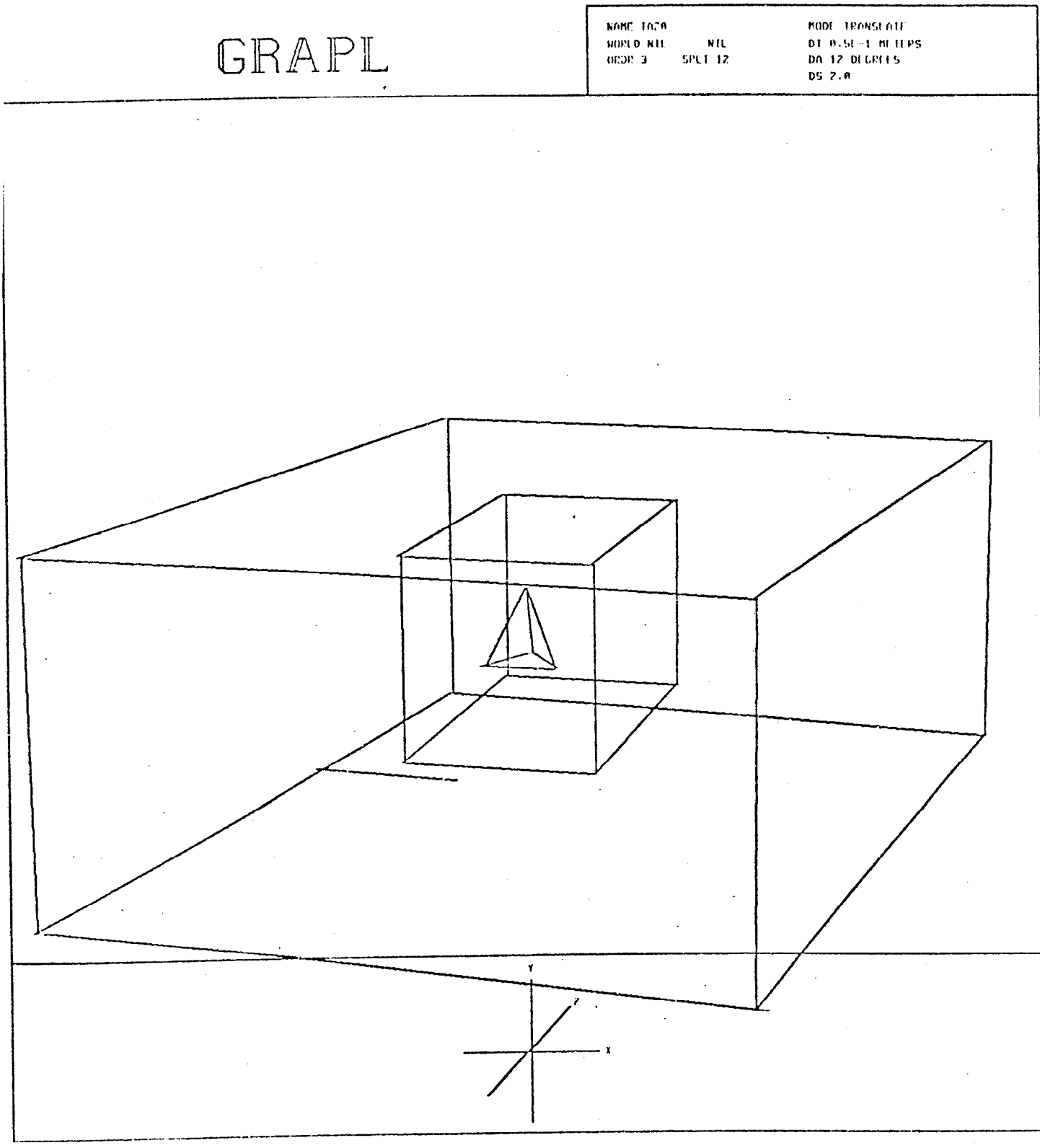


Figure 6.1.16 Within the drawer

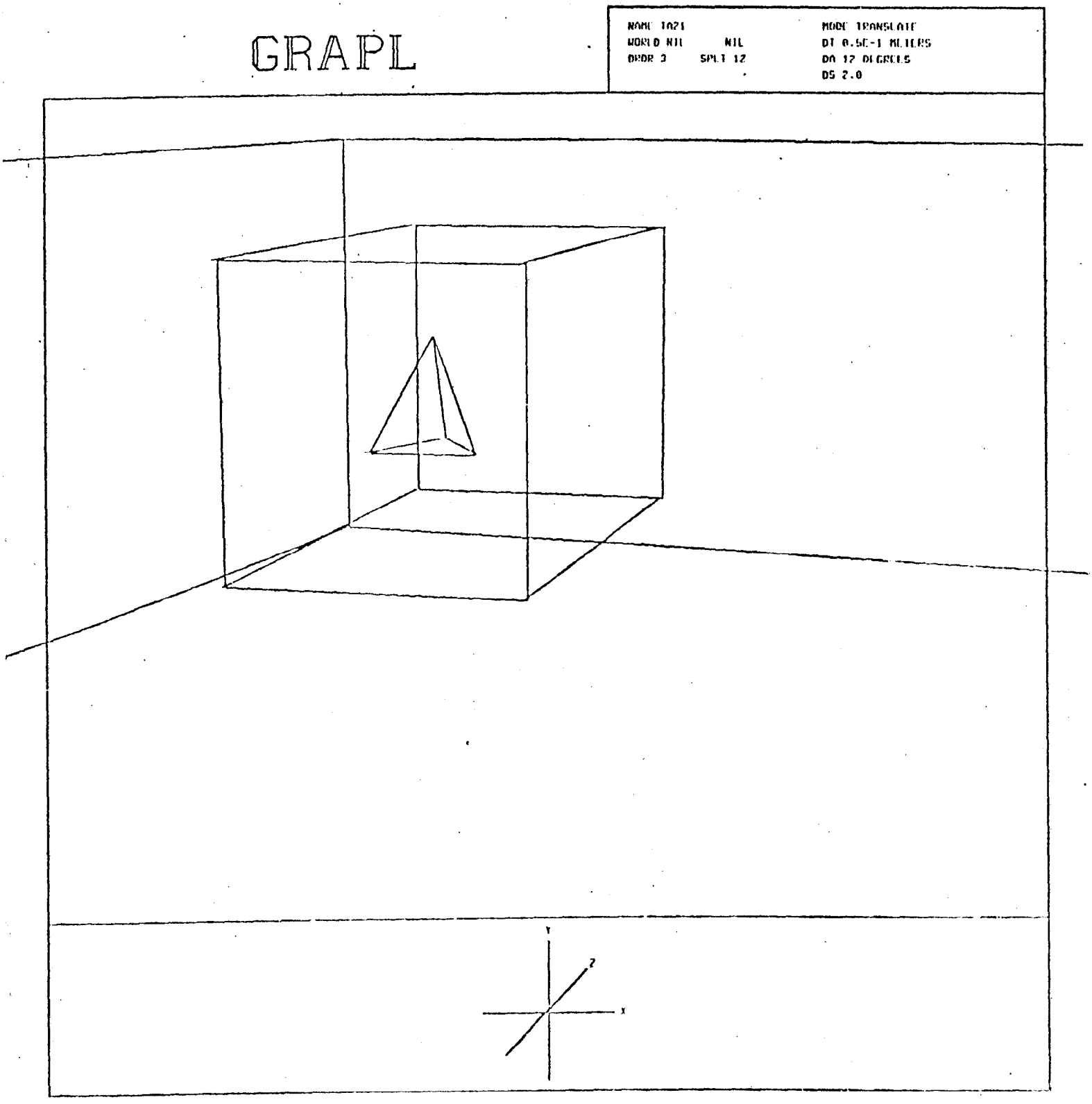


Figure 6.1.17 Closer again

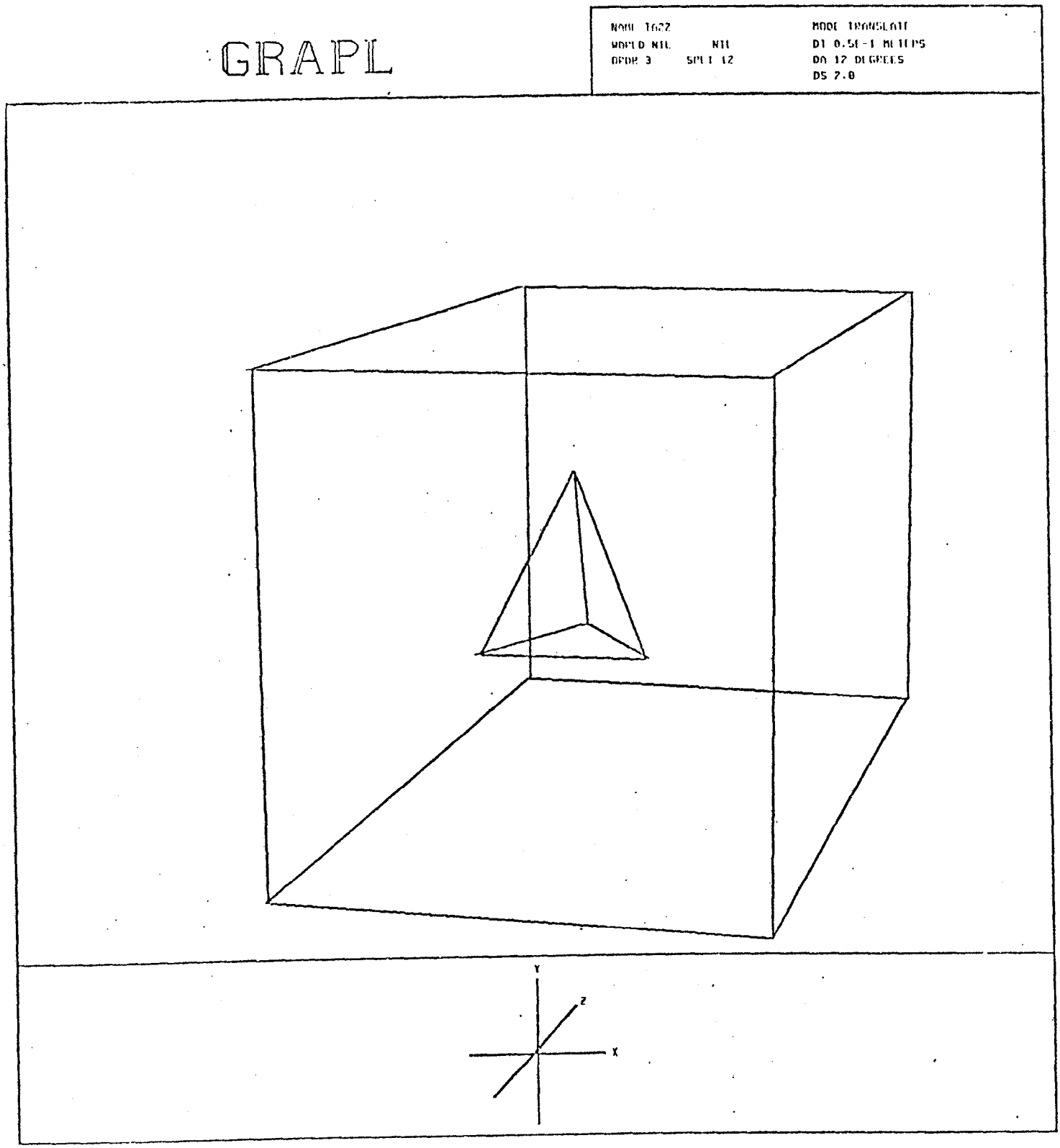
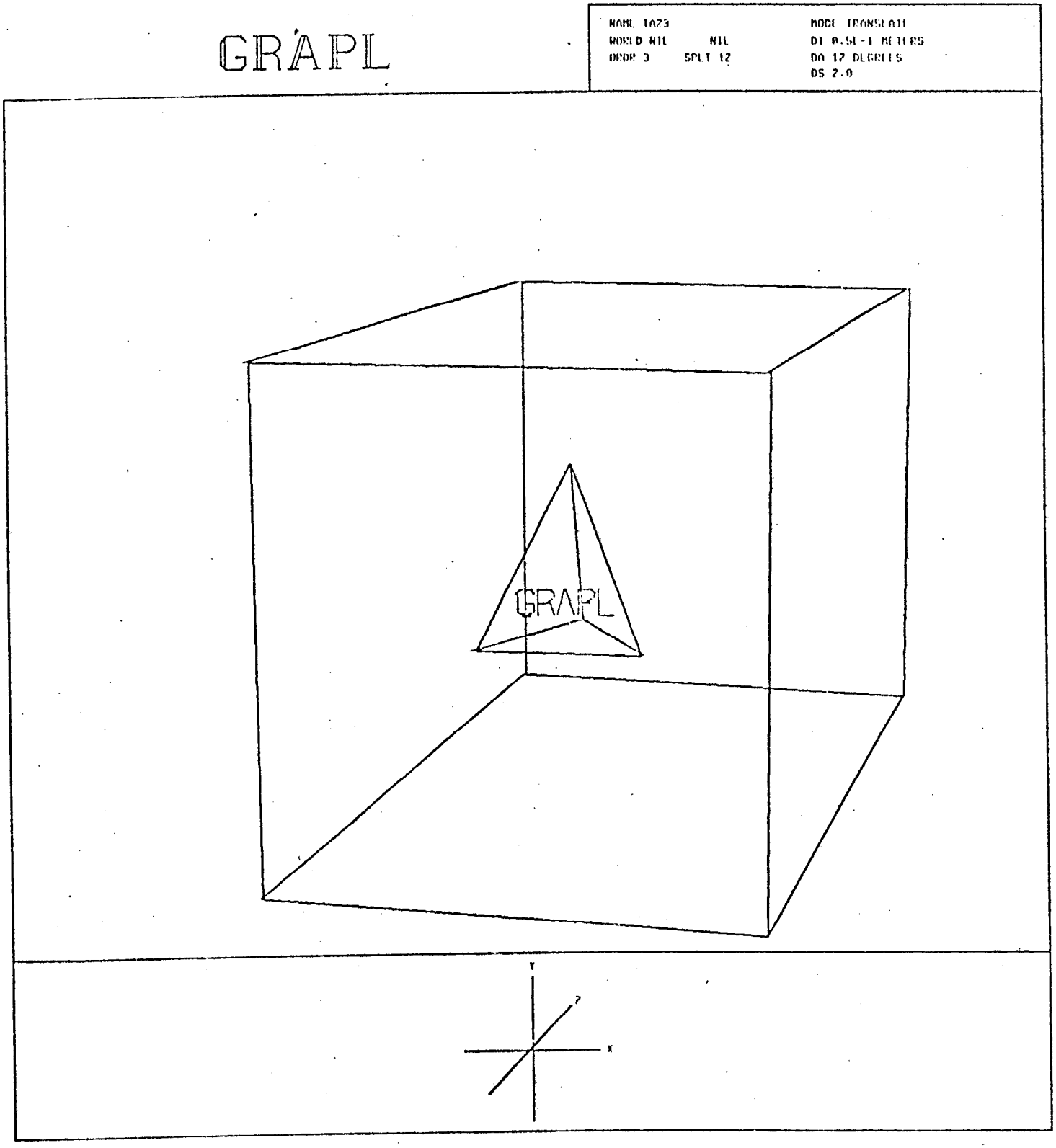


Figure 6.1.18 "GRAPL" appears within the pyramid



6.2 A simple operating system

To demonstrate the use of the GRAPL system in a more applications oriented environment we decided to model a simple operating system. The CALIDOSCOPE Operating System <UC 72> for the CDC 6400 running at the University of California at Berkeley was chosen as a basis for our model both for its simplicity and because we are reasonably familiar with its characteristics. The CALIDOSCOPE system supports a multiprogramming environment consisting of at most five execution tasks plus miscellaneous input/output functions.

6.2.1 Hardware environment

Cal's CDC 6400 includes the standard ten peripheral processors plus a main cpu. The system supports 65K of central memory plus 133K extended core storage. Users have access to at most three tape units, although several additional units usually are on-line. Two IBM 1403 printers and a CDC 501 printer serve as the primary output devices. Additionally, there are a card punch, operator's console, and high-speed card reader.

The 6400 also supports a remote computer system (RCS), primarily designed for the attachment of small

computers driving driving card readers, line printers, and other input/output devices; and a remote terminal system (RTS), used for remote job entry and retrieval. No true time-sharing system is implemented.

6.2.2 Software environment

The CALIDCSCOPE Operating System <UC 72> is a modification of the CDC Scope 3.0 system tailored for Berkeley's particular requirements. The system is modular and consists of several essentially independent parts, the most important of which is HYDRA which handles input/output tasks and spooling.

The scheduler selects tasks for execution using a simple first in first out algorithm subject to the following restrictions. Users may give jobs one of five priorities:

Priority E - Express	-	Highest priority
Priority J - Job	-	Usual priority
Priority S - Short job	-	Similar to J jobs
Priority D - Deferred	-	Run after all J jobs
Priority I - Idle	-	Run only if idle

Users must indicate the maximum allowed running time for their jobs as well as the maximum allowed number of pages of output. The scheduler then subsorts the job queues into the following classes:

Class 0 - 0 - 20 (octal) seconds CPU time

Class 1 - 21 - 100
Class 2 - 101 - 400
Class 3 - 401 - ...

Print jobs fall into two classes, those under 25 pages and those over, with priority generally being given to the shorter jobs.

Additional features of the system handle HYDRA RTS (remote terminal system, priority P) jobs, ensuring that deferred jobs do not persist in the system forever, and so forth.

6.2.3 The GRAPL model

Because of the ready availability of data on the operational characteristics of CALIDOSCOPE through use of the QUEUETEST program, it was decided to model the external performance of CALIDOSCOPE rather than to construct a totally accurate model of its internal behavior. (After all, CALIDOSCOPE models itself perfectly; our goal is not duplication; rather, our goal is to demonstrate GRAPL.)

The model is based upon the following:

The priority queues: E, J, D, I
The scheduling algorithm
Physical considerations such as core size, etc.
Observed input rates as a function of time

Output from the model consists of most of the

essential information reported by QUEUETEST including:

Number of jobs in each queue
Total number of jobs processed so far
Job backlogs in terms of CPU time
Average turnaround

Additional information regarding any portion of the simulation is easily obtainable through use of GRAPL's Monitor commands.

6.2.4 Performance

Performance of the model (called SCOPE within the GRAPL system) has been more than satisfactory. A wide range of system characteristics may be observed including the infinite deferral of Idle jobs when the system is heavily loaded, a midday backlog of jobs in all classes due to the high submission rate, excellent turnaround during the late evening and early morning hours, etc.

6.3 A simple graphing system

We implemented a system for graphing functions of the form $y = f(x)$. As is mentioned in Chapter 7, the code for this system fits easily inside less than a page. To extend the system to two dimensions, add automatic scaling, change the form of plot to bar graphs, and other such extensions and modifications would be the work of an

afternoon.

In each Figure, the x-axis ranges from -40 to +40; the y-axis has the same range, but has been scaled by an arbitrary amount so as to fit the entire graph within the visual frame.

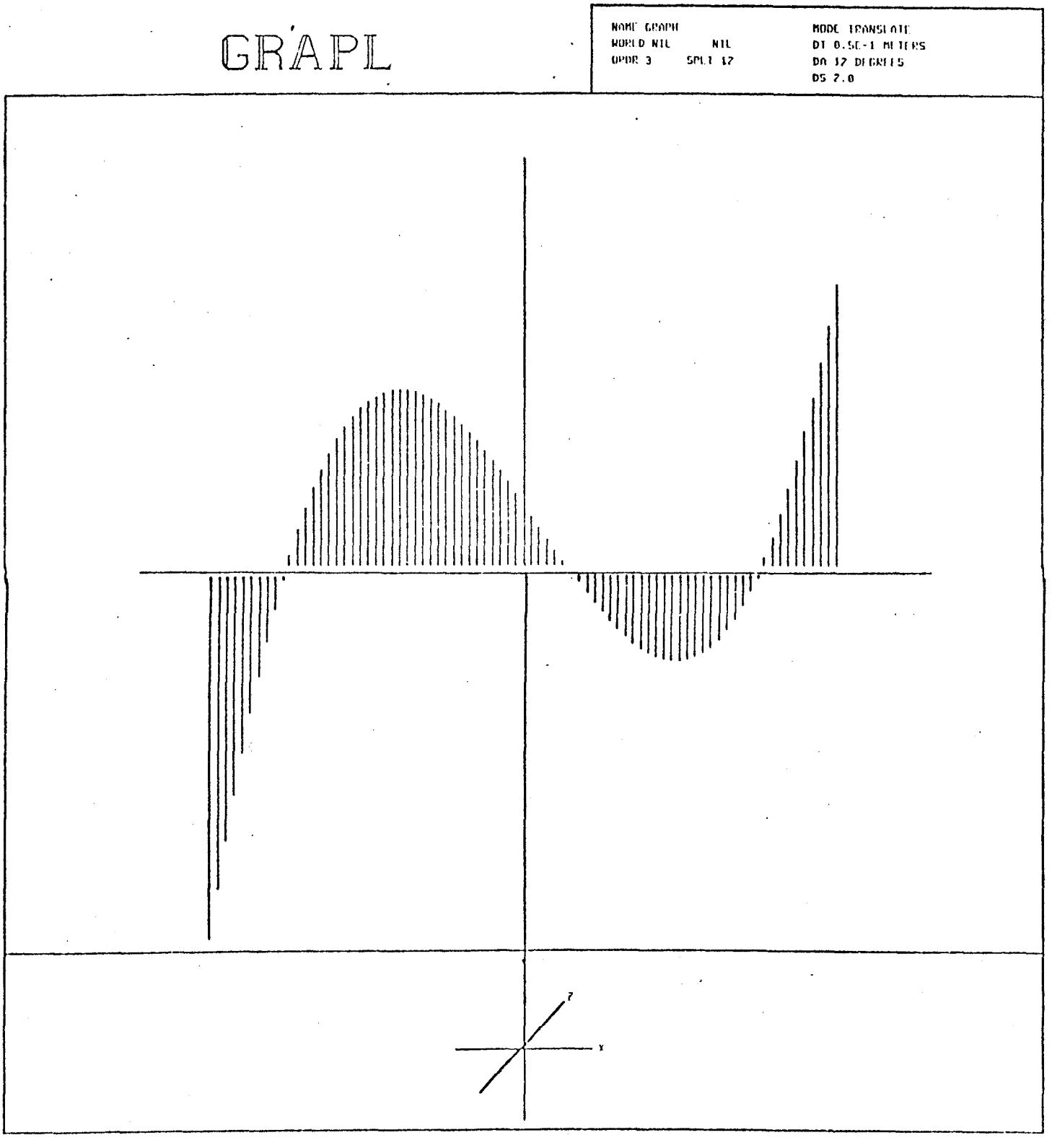
Figure 6.3.1 shows the graph of a typical cubic equation, y = (x+30) (x-6) (x-30).

In Figure 6.3.2, we show the graph of one of the arrival functions used in the operating system simulation of section 6.2: y = (abs(x) + ab)/(x*x + b*b), with a=15, and b=5.

Figure 6.3.3 shows another arrival function, y = (x mod 24)*(x mod 24 + a - 12), with a=24.

In Figure 6.3.4, we show a typical symmetric quartic equation, y = (x+35) (x+5) (x-5) (x-35).

Figure 6.3.1 $y = (x+30)(x-6)(x-30)$



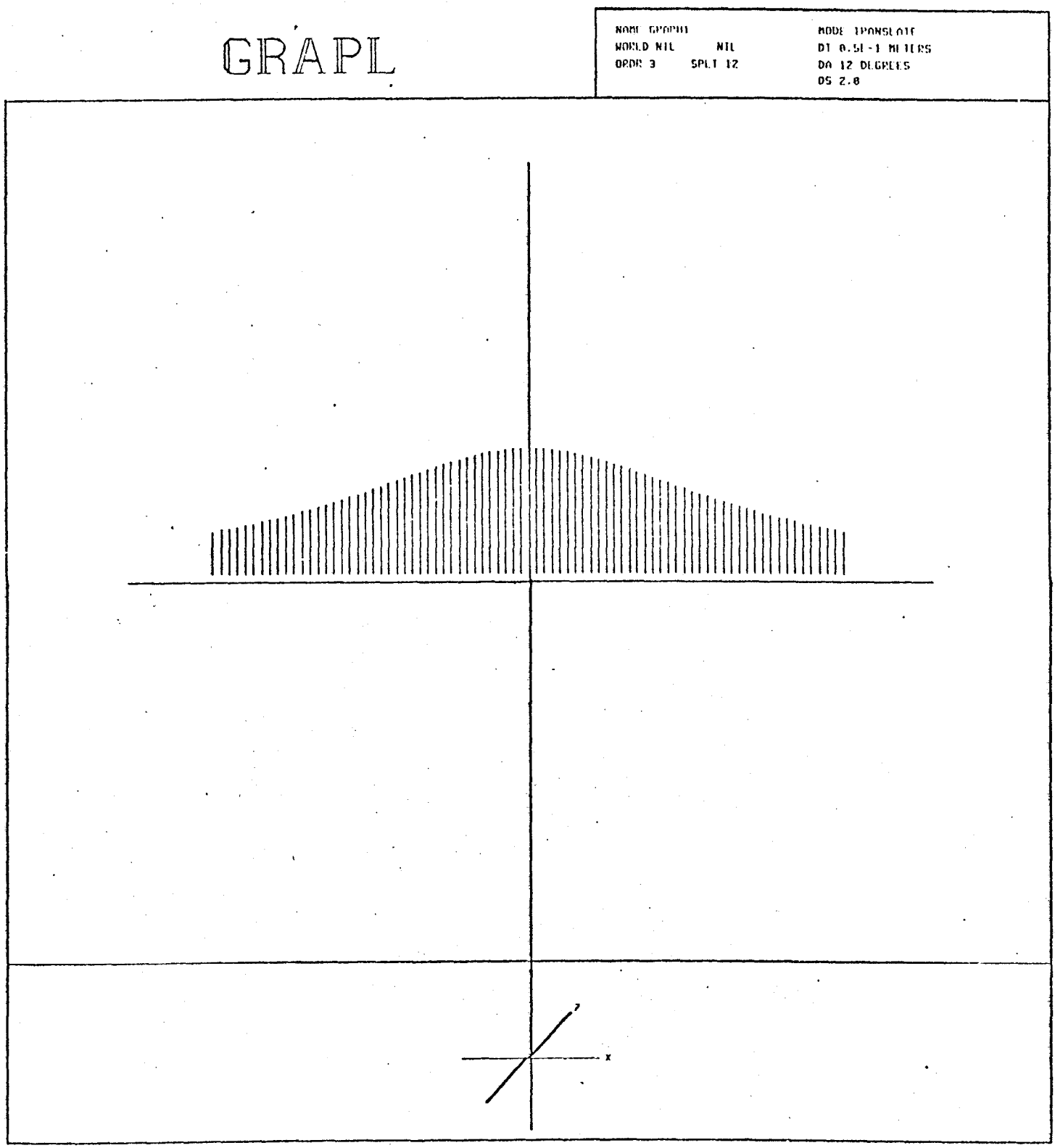
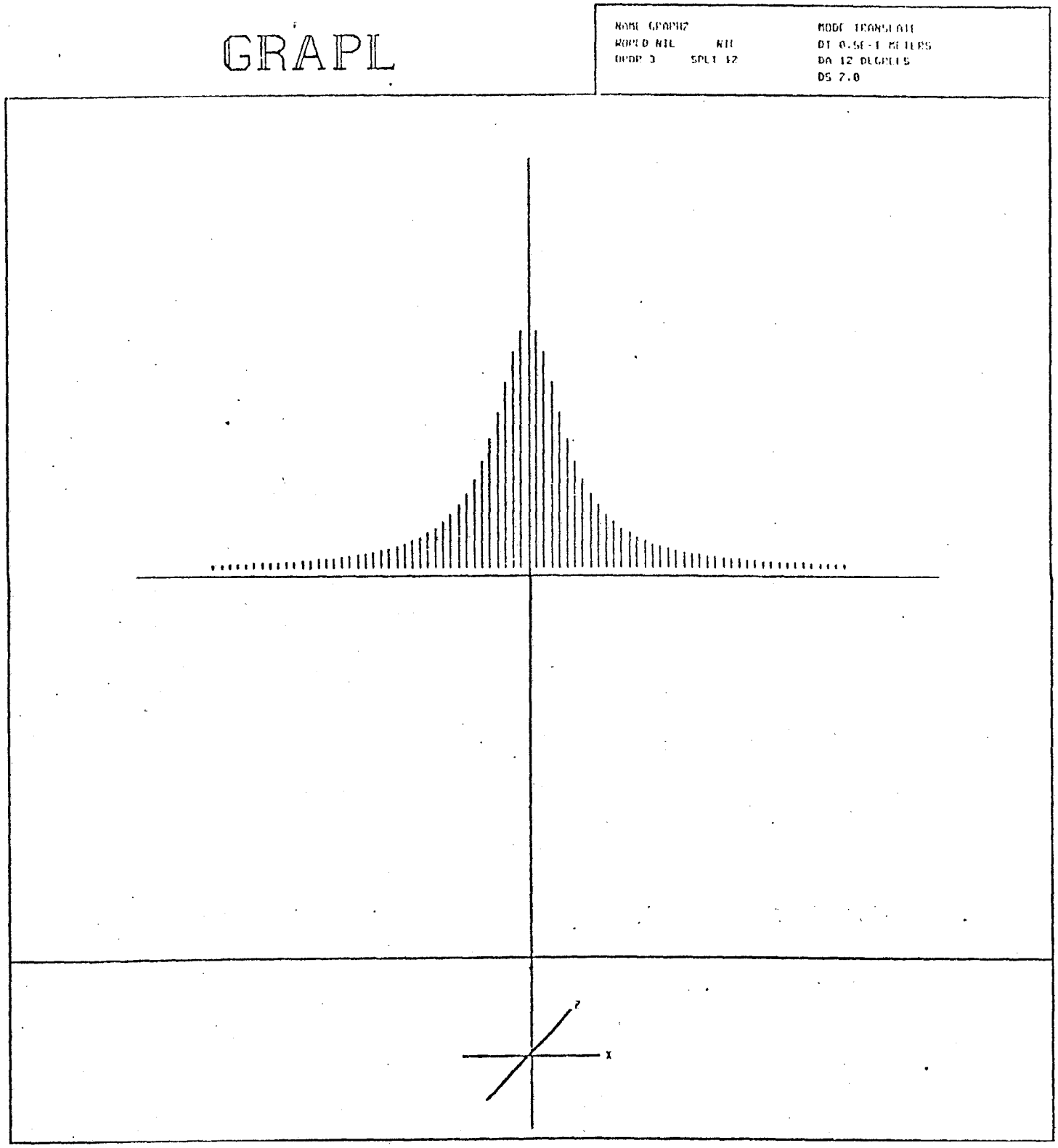
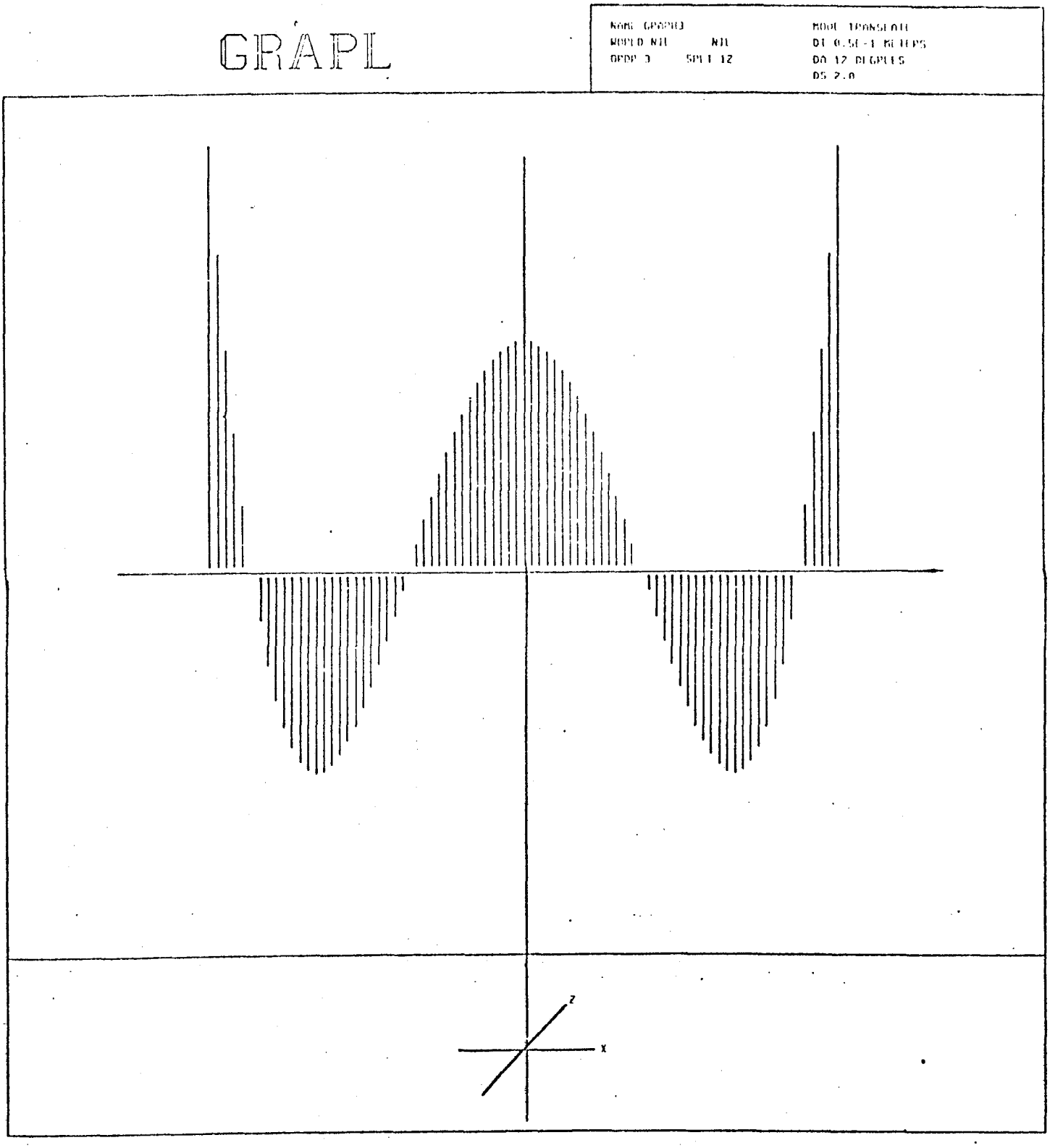

GRAPL
NAME GRAPH1
WORLD NIL NIL
ORDR 3 SPLT 12
MODE TRANSLATE
DT 0.5E-1 METERS
DA 12 DEGREES
DS 2.0
Y
X

Figure 6.3.3 $y = (x \bmod 24)(x \bmod 24 + a - 12)$




GRAPL
NAME GRAPH3
MODE TRANSLATE
MODEL NIL NIL
DI 0.5E-1 METERS
DRAW 3 SPLT 12
DA 12 DEGREES
DS 2.0
Z
X

6.4 Building a house

In this section we present an example of how an architect might use a system such as ours to construct a house. We illustrate both his interactions with the system and the system's responses to these interactions.

The GRAPL system as currently implemented could be used to design a house. Most of the commands described below already exist. However, to be easily usable by the untrained architect a redesign of the command language (such as is mentioned in Chapter 8) should be done.

6.4.1 Overview of the architectural design process

Architectural design usually proceeds in several phases some of which are dependent upon the results of previous phases and some of which are not. In general, we may break down the design process into three major steps: planning, preliminary design, and final design; although the actual distinctions among these are usually rather loose. In some firms this distinction may be made by observing in which department the design currently resides. In others, and especially in the case of designing a house, the distinction will be even more difficult to make.

The planning phase involves the generation of a statement of requirements. This statement should include most of the important requirements the structure must meet. In particular, this statement would include the number of rooms, general partitioning of space, access from one part of the structure to another, access to the street, and so forth. To this statement is added the building code requirements for the particular site and type of structure. A new document is generated, sometimes called the "architectural program," which contains the specifications of all spaces and their corresponding functions, utilities, square footage, mechanical systems, and so forth.

The preliminary design phase begins at this point. Decisions must be made as to the general design philosophy: Should the building be open or closed? How important is height? Is the structure to be essentially a shell housing its occupants, or is it to be a showplace? How important is and what are the limitations of cost? Following this, the general location of spaces and their functions are made. This yields the building shell. A "schematic" is generated which tells how the building works; and the preliminary specification of the structural, mechanical, electrical, utility, and communication systems is made.

At this point final design begins. In this phase

the preliminary design is further resolved until it is realized in its final form as blueprints to be used in construction.

In any architectural design project the architect will be concerned with the so-called "Object Systems of Design." These are:

- Site
  - Earthwork
  - Grading
  - Compacting
  - Paving
  - Landscaping
  - Orientation
- Structures
  - Foundations
  - Footings
  - Superstructure
- Enclosures
  - Floors
  - Walls
  - Ceilings
  - Roofs
- Space-Use
  - Rooms
  - Wings
  - Suites
  - Storys
  - Complexes
  - Porches
  - Balconies
  - Fireplaces
  - Patios
  - Lanais
  - Garages
- Mechanical
  - Heating
  - Ventilation
  - Air Conditioning
  - Plumbing
- Electrical
  - Power
  - Lighting
- Communications

- Telephone, telegraph, etc.
- Intercom
- Computer
- Radio, television

Appurtenances

- Furniture
- Fixtures
- Equipment

In addition, the architect will consider the following "Attribute Systems":

Shape
- Area
- Volume

Weight

Cost
- Initial
- Maintenance
- Building useful life

Materials

Visual
- Color
- Reflectance
- Light (intensity, distribution)
- Texture (visual)

Acoustics
- Sound transmission
- Sound diminution
- Reverberation time
- Reinforcement

Thermal
- Heat transmission
- Expansion-contraction

Safety
- Fire resistance
- Fallout radiation protection

Tactile
- Texture
- Vibration
- Rigidity

Miscellaneous
- Use flexibility
- Aesthetics
- Ecology
- Social aspects

As an example of the types of analyses the

architect may require, we list here some of the considerations he may encounter in determining the structural design of the building. In each case, a corresponding software package could be invoked:

Calculation			
	Input data	General tables	
	Tension	Compression	Torsion
	Shear	Deformation	Displacement
	Bending	Buckling	Stability
	Mobility	Collapse	etc.
Members			
	Beams	Columns	Struts
	Connectors	Walls	Footings
	Complex structures		etc.
Structures			
	Trusses	Frames	Shells
	Tension structures		Composite structures
	etc.		
Materials			
	Concrete	Wood	Plastic
	Steel	Reinforced resins	
	Structural foams		Brick
	Aluminum	Rock	etc.

The architect also will have at his disposal a wide variety of standard forms. These are currently found in various catalogs, such as:

The Architect's and Builder's Handbook
Architectural Specifications
Architectural Standards, etc.

These catalogs include not only the specifications of standard building materials but also various kinds of furnishings, fixtures, office equipment, etc.

6.4.2 Specifics for design of a house

An architect might make himself the following list (program) to guide him in the construction of a house:

- Thumbnail sketches
- Preliminary drawings
- Structure
- Site, landscaping, foliage
- Foundation
- Superstructure
- Outside walls
- Roofs
- Windows, skylights
- Partitions
- Wall sections
- Doors, stairs, stairwells, etc.
- Room plans
  - Attic
  - Bathrooms
  - Halls
  - Bedrooms
  - Work rooms
  - Den
  - Dining room
  - Library
  - Living room
  - Closets
  - Kitchen
  - Garage
  - etc.
- Plumbing
- Electrical
- Gas
- Heating & ventilation
- Communications (intercom)
- Mechanical (dumb waiters, etc.)
- Details
  - Windows, skylights
  - Cornice
  - Stairs
  - Fireplace
  - Wall section
  - Doors
- Interior decorating
- Furnishings

Before any design can begin, the architect should have a list of requirements for the structure. This list might include:

Approximate cost
Style
Number of rooms of specific types
Lighting
Special requirements for adjacency of rooms
Particular requirements of the
prospective tenants
etc.

We will assume that our list of requirements is:

Cost	$ 20,000 - $ 25,000
Style	California modern
Rooms	Living room, dining alcove, bathroom, kitchen, bedroom
Square-footage	1500
Special req's	None

6.4.3 The architect's actions

We now suggest the steps an architect might take in the design of a house to meet the above requirements. It probably will be helpful to refer to the following section (Section 6.4.4) as it explains point-by-point the system's responses to the architect's actions.

(1) Log into the GRAPL system.

(2) We identify ourselves to the system, specify our client's name and the project's name.

(3) We request sketching mode, so we may deal with the CRT as if it were a highly sophisticated type of paper.

(4) We begin to create a thumbnail sketch of the floor plan. We are assuming that this will be a single-floor dwelling.

(5) We sketch in walls, doorways, entrances, and label the various enclosures. (eg. "BR," "LR," "BA," etc.) We insert closets and cabinets, etc.

(6) Not satisfied with this first sketch, we indicate to the system that we wish to move the bedroom and its neighboring bathroom to a different place in the sketch.

(7) This is a more pleasing configuration; so we give it a name: PLAN1, and then erase our drawing.

(8) This sequence is repeated several times; we try several variations sometimes using the current sketch, sometimes creating new ones.

(9) We ask that each of our floor plans be displayed on different parts of the screen so that we may compare them simultaneously. At this point we decide that PLAN3 is inferior to the others, as is PLAN5, so we delete them. We present the remaining floor plans to our client and come to an agreement that a slight modification of PLAN4 will best suit his needs.

(10) We now give the name FLOORPLANSKETCH to PLAN4 so that we can identify it more easily. Incidently, each time we create an object or access one, the system automatically retains the date of creation, the creator, and the last time it was referred to.

(11) We begin to firm up the sketch. We indicate which lines to straighten, possibly also indicating that this one is a standard interior wall of type "SIW34" and that one is an exterior wall which will be finished with redwood siding, etc.

(12) We ask for the square-footage of the bedroom, and the system responds that we have yet to give sufficient specifications as to the lengths of the walls. So rectifying our mistake, we begin to give dimensions where required. As we do so, the picture on the CRT changes to reflect the proper sizes. Now we ask for the area of the bedroom; think better of it, and ask for the square-footage in all the rooms.

(13) At this point we should probably confer with our client to be certain that the current floor plan still reflects his wishes and that the sizes of the various rooms are adequate. He indicates that the living room should be somewhat longer, perhaps by 5 feet, and that the bedroom should also be enlarged slightly. We remind him that this will increase the cost of his house as well as its size.

(14) We therefore lengthen the living room by asking the system to move the wall 5 feet and to move the bedroom walls around slightly as well.

(15) This now is a firm floor plan and for easy reference we give it the name FLOORPLAN.

(16) We now ask the system to rotate the floor plan so that we can see it in perspective.

(17) Now we ask the system to extend all of the walls 9 feet vertically.

(18) Everyplace that we had specified a window on the floor plan is now a pair of "tic marks" in the walls. In each case, we now specify a window opening, perhaps giving additional details such as sliding, louvered, etc.

(19) We go on to specify each of the doors in the house and each of the passageways.

(20) At this point it may be advantageous to examine the house in more detail. For instance, we may rotate it to see what it looks like from each side, obtain perspective views, and "walk" through the house, getting an idea of what each of the rooms will look like.

(21) Now we add a roof; the client has decided upon a crushed rock and tar paper roof, flat, with a moderate amount of overhang on the southern exposure to provide shade in the summer.

(22) We are now ready to begin specifying the structural aspects of the house: the exact type of structure for each wall, the ceiling, the roof, etc. We must occasionally move studs or other supporting members slightly in order to ensure structural integrity. From experience we know that there will be no undue loading on any part of the house, but we ask that the structural analysis routine be run to verify this.

(23) So long as we are running analysis routines, we may wish to compute the cost of the dwelling on a cubic foot basis, the cost of the structural wood used, and so forth. This will help to give us a feeling of how close we will be to the specified dollar limit.

(24) We next specify the foundation and footings, a poured concrete slab; and we may also specify the driveway, sidewalks, curbing, and proximity to the street.

(25) We are now ready to install the major plumbing systems. The water main on the street is at a given location. We indicate a main running to the house, decide where the water heater will be (in a closet in the kitchen), and route the main there. We also give the sewage pipe routing information.

(26) Next we indicate where the plumbing should go to service the kitchen and bathroom. We may also make modifications to meet certain building code requirements.

(27) We must also specify where sinks, shower, bathtub, lavatory, dishwasher, and washing machine facilities are to be placed.

(28) Having completed the plumbing systems, we now begin to specify the electrical system. Power comes from underground at a given location; we specify a conduit near the plumbing lines running to the house.

(29) We next bring the power lines up to a circuit box, then run main lines to each of the rooms.

(30) We insert electrical outlets at convenient places. Now we specify what kind of lighting fixtures will be in each room and run power lines to them.

(31) We indicate where the refrigerator, the electric range, and electric heating unit will be. We give the wiring necessary for them.

(32) At this point we may again wish to examine the house from several points of view, perhaps wandering through the various rooms.

(33) We continue the design, specifying the phone line into the house and the extensions the client wishes in the kitchen and bedroom.

(34) Next we indicate where the heating ducts will be. The house will have a forced-air electric heater, centrally controlled, with outlets in each room excepting the kitchen. The control box will be in the living room. So we must specify some additional wiring for it.

(35) At this point we could ask the system for detailed blueprints which may be given directly to the various contractors and sub-contractors for use in construction.

(36) Although our client will furnish the house himself, it is a small matter to insert beds, dressers,

couches, etc. in crder to show him how his house will appear upon completion. Together, we now examine the finished plans. Obtaining his final approval, construction can begin.

6.4.4 The system's responses

The architect may wish to include his list of requirements in the system so that he may readily refer to them as the design proceeds. He may do this by entering them as a (text) file to which he might give the name REQUIREMENTS.

(1) The lcg-in process will reinitialize the GRAPL system to the version which the architect last used. The architects's system is viewed as a continuing process which is suspended whenever he logs out and is resumed when he logs in.

(2) Identification of the project's and client's names serve to establish a primary context for future actions in the system. The architect may change this context at will. For instance, if he is doing several designs at the same time.

(3) Rather than forcing the architect to learn the system in its entirety, we construct several subsystems with which he may learn to interact. One such subsystem is "sketch mode," in which he may deal with the CRT as if it were essentially a piece of paper. This mode is somewhat similar to the normal mode of interaction of SKETCHPAD, but does not force all the automatic line straightening and line connecting features -- these are options the designer may choose to use or nct as he prefers.

(4) This sketch will look as if it were made on paper with a charcoal pencil.

(5) Each line he sketches is retained as an "analog patch." The architect may refer to each by giving it a name or by pointing at it with the stylus, mouse, etc. Enclosures are likewise given names at the option of the architect for easy reference. Enclosures are represented by trees in the data structure.

(6) Movement of an enclosure is reflected by a change in the tree structure which represents the current state of the house.

(7) Naming the sketch corresponds to giving the tree a name. It may already partially exist in secondary storage, but the naming operation will most probably have as a side effect the outputing of the structure. Erasing the drawing probably will force the outputing of same to secondary storage. If the structure does not already have an external name, the system will ask for one. The name for an unnamed object is CURRENT.

(8) Renaming is obvious. Creating a data structure similar to another is a tree copy, followed by updates on the new copy.

(9) At any time we may specify portions of the screen as windows. In each window we may request objects to be displayed. This display does not affect the data structure in any way. Each window is handled by another instantiation of the "window demon." The contents of each window may also be manipulated. If this manipulation results in a change in the actual structure of the object, then it will be reflected as a change in the object's representation in the data structure.

(10) This multiple naming permits us to access objects (drawings) by name, by date, by time of last use, etc. In addition, we may always look at the current dictionary of objects in the system (selected by client, architect, date, etc.) if we forget what we have already stored away and what its name is.

(11) This straightening process is actually one of redefinition. Each analog patch is replaced by the data structure reflecting its new definition. The additional information as to what kind of wall, cost, structure, and so forth goes in as attributes of the structure.

(12) Asking for the floor area of the bedroom corresponds to running an attribute system routine. These may be run at any time. If the current information in the data structure is insufficient to calculate an exact answer, the system will request that the missing information be supplied.

(13)

(14) This request to the system is effected by changing the length of the walls of the livingroom and

bedroom. The system will automatically adjust the display reflecting the new dimensions. An alternate manner in which this may be done is by selecting the appropriate wall and asking the system to move it following the stylus, mouse, etc. The current dimensions of the wall are displayed.

(15) We rename the data structure.

(16) Rotation as well as all other affine and perspective transformations are handled as attribute system routines which are supervised by the display demons. They do not cause any change in the data structure.

(17) This important action changes our two-dimensional plan into a three-dimensional object. It forces a redefinition of each wall, closet, opening, or enclosure.

(18) Specifying the windows is a matter of replacing the definitions of the appropriate walls by the new data structure which has the selected window inserted in it at the specified position. The type of window, wall, and so forth are simply attributes of each, respectively.

(19) Doors are special in that we may ask the system to show the space swept. (This is not the usual mode of display.) Passageways are reflected by the attribute "access."

(20) Again, these operations are handled by the display demons as attribute system routines. No data structure changes are made.

(21) Addition of the roof inserts the appropriate objects into the data structure, complete with attributes as to type, size, and other pertinent data.

(22) This operation is essentially an elaboration of the attributes already in the data structure. Specific types, number of studs, kind of wood, and so forth are all inserted as attributes. Additionally, we may examine walls now in much greater detail, seeing cross-sections if we wish.

(23) Here we are running attribute system routines. If we request a computation with which the system is unfamiliar, we may specify how to compute it and give it an appropriate name. The new routine will henceforth be available to the architect at all future sessions.

(24) These operations are elaborations of the data structure and the addition of attributes.

(25) The addition of plumbing is similar to the specification of the structural details of the building. Additional data structure is created, attributes added; now the wall cross-sections will have plumbing shown as well as structural members.

(26)

(27)

(28) The electrical system goes in similarly to the plumbing.

(29)

(30) There are additional details in each room.

(31) Additional details within each room are additions to the data structures corresponding to same.

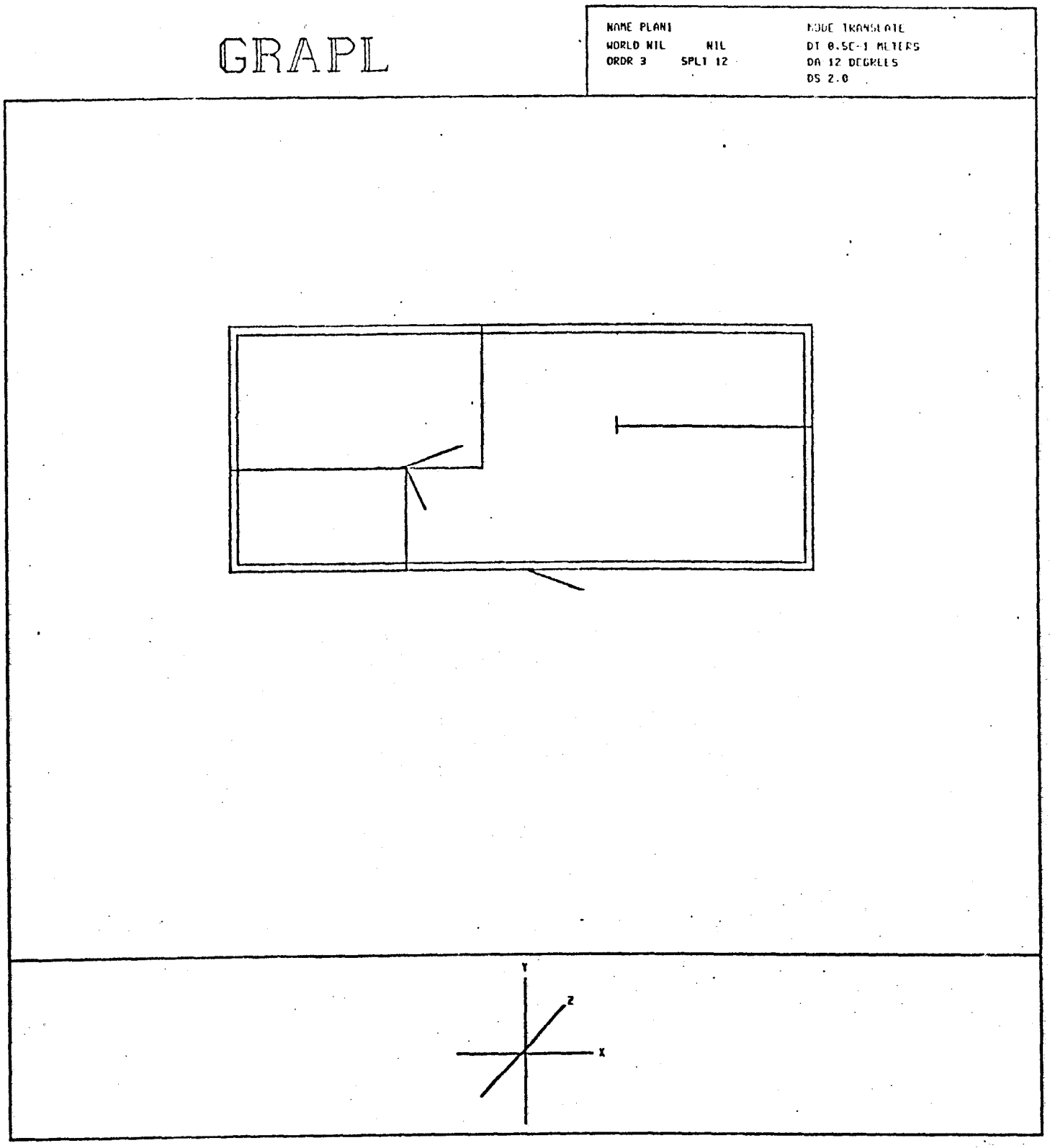
(32)

(33) The communication system goes in similarly.

(34) Heating and ventillation, similarly.

(35) Blueprints go out from the CRT on film or onto a plotter. This is handled by the display demons.

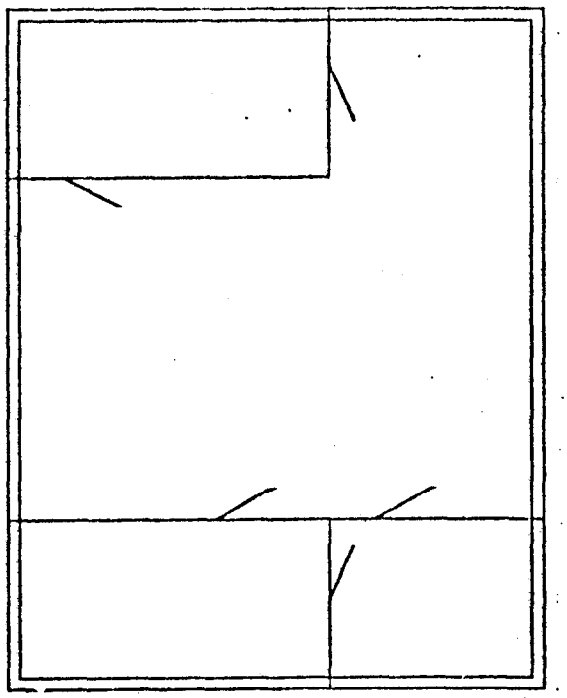
(36) Additional modifications to the house may be made at any time and new blueprints produced.


GRAPL
NAME PLAN1
WORLD NIL NIL
ORDR 3 SPLT 12
MODE TRANSLATE
DT 0.5E-1 METERS
DA 12 DEGREES
DS 2.0
Y
Z
X

GRAPL

NAME PLAN2
WORLD NIL NIL
ORDR 3 SPLT 12

MODE TRANSLATE
DT 0.5E-1 METERS
DA 12 DEGREES
DS 2.0



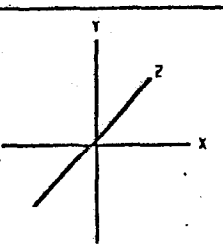
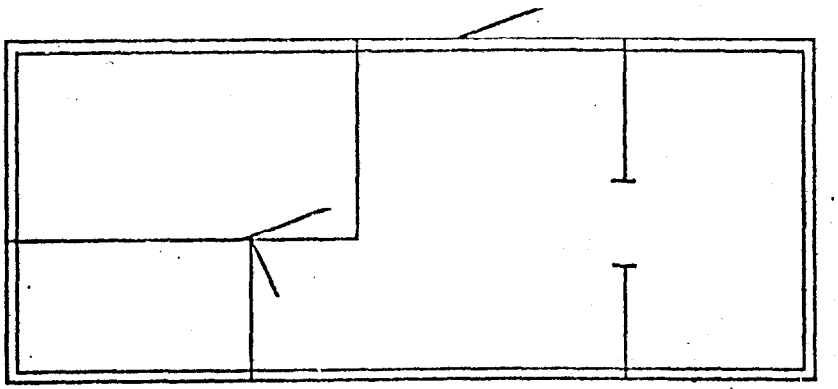


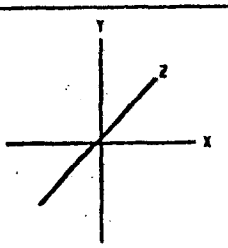
Figure 6.4.3 Floorplan #3

GRAPL

NAME PLAN3
WORLD NIL NIL
ORDR 3 SPLT 12

MODE TRANSLATE
DT 0.5E-1 METERS
DA 12 DEGREES
DS 2.0

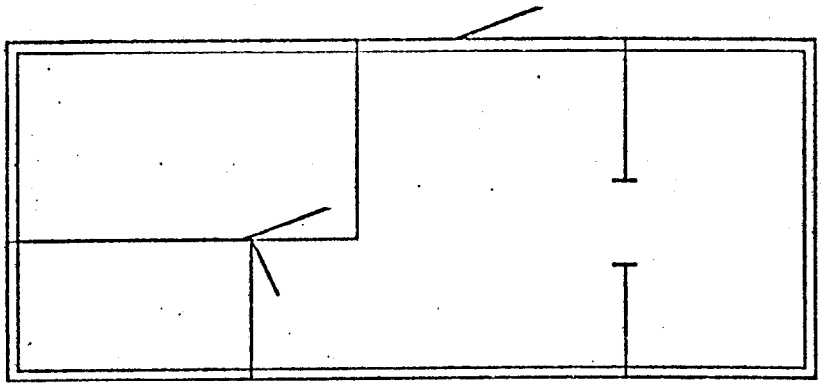




GRAPL

NAME PLAN4
WORLD NIL NIL
ORDR 3 SPLT 12

MODE TRANSLATE
DT 0.5E-1 METERS
DA 12 DEGREES
DS 2.0



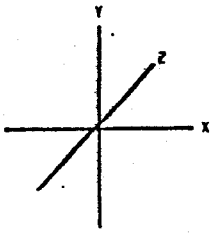
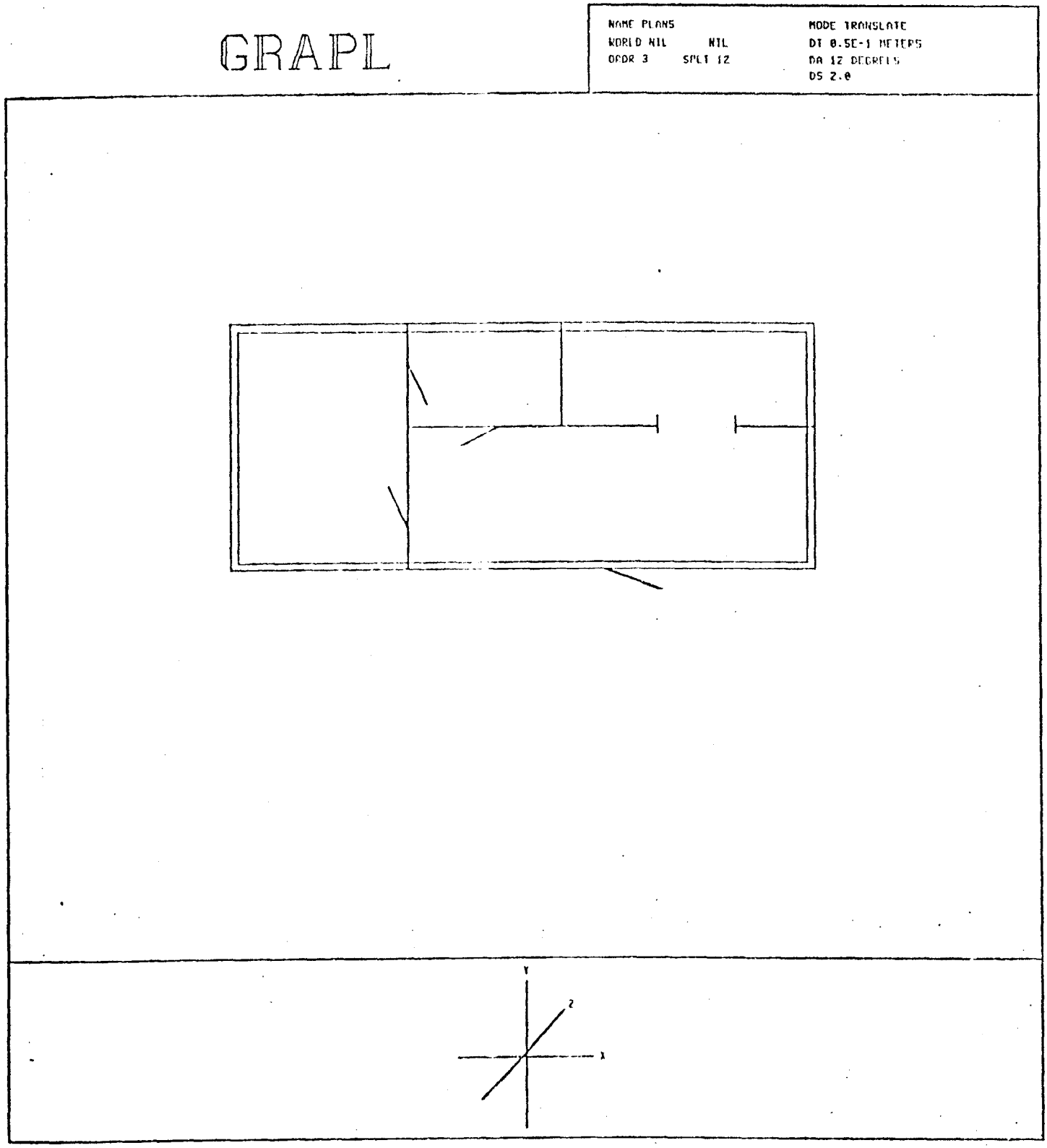


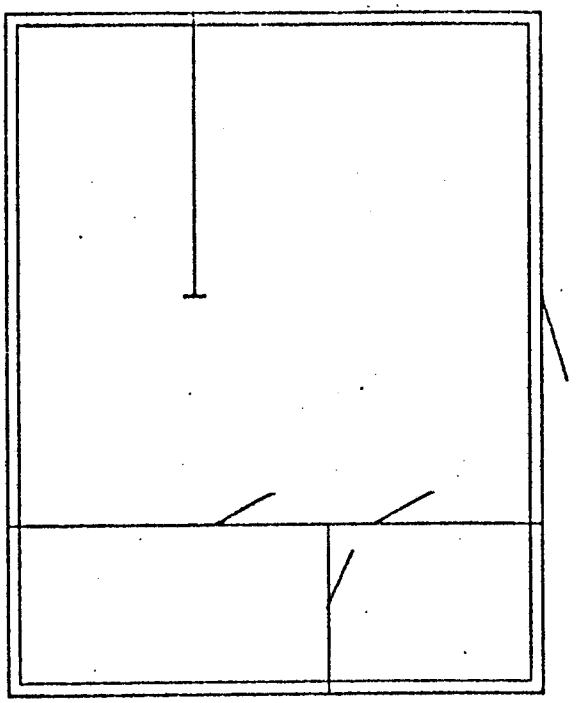
Figure 6.4.5 Floorplan #5



GRAPL

NAME PLANG
WORLD NIL NIL
OPDR 3 SPLT 12

MODE TRANSLATE
DT 0.5E-1 METERS
DA 12 DEGREES
DS 2.0



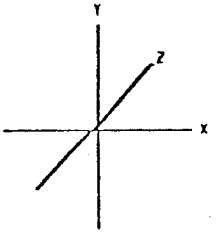


Figure 6.4.7 Floorplan #4 modified

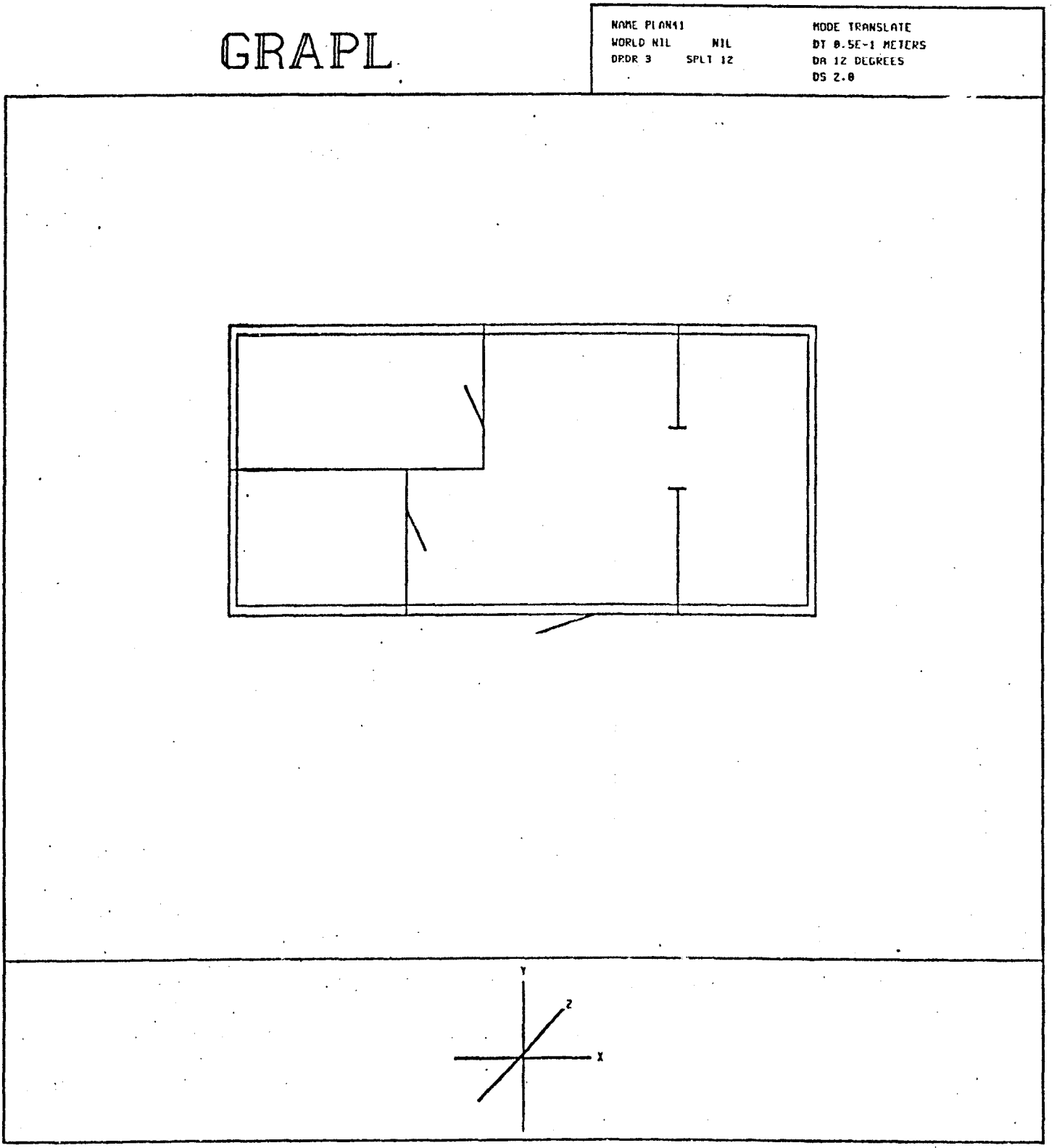


Figure 6.4.8 Floorplan #4 in perspective

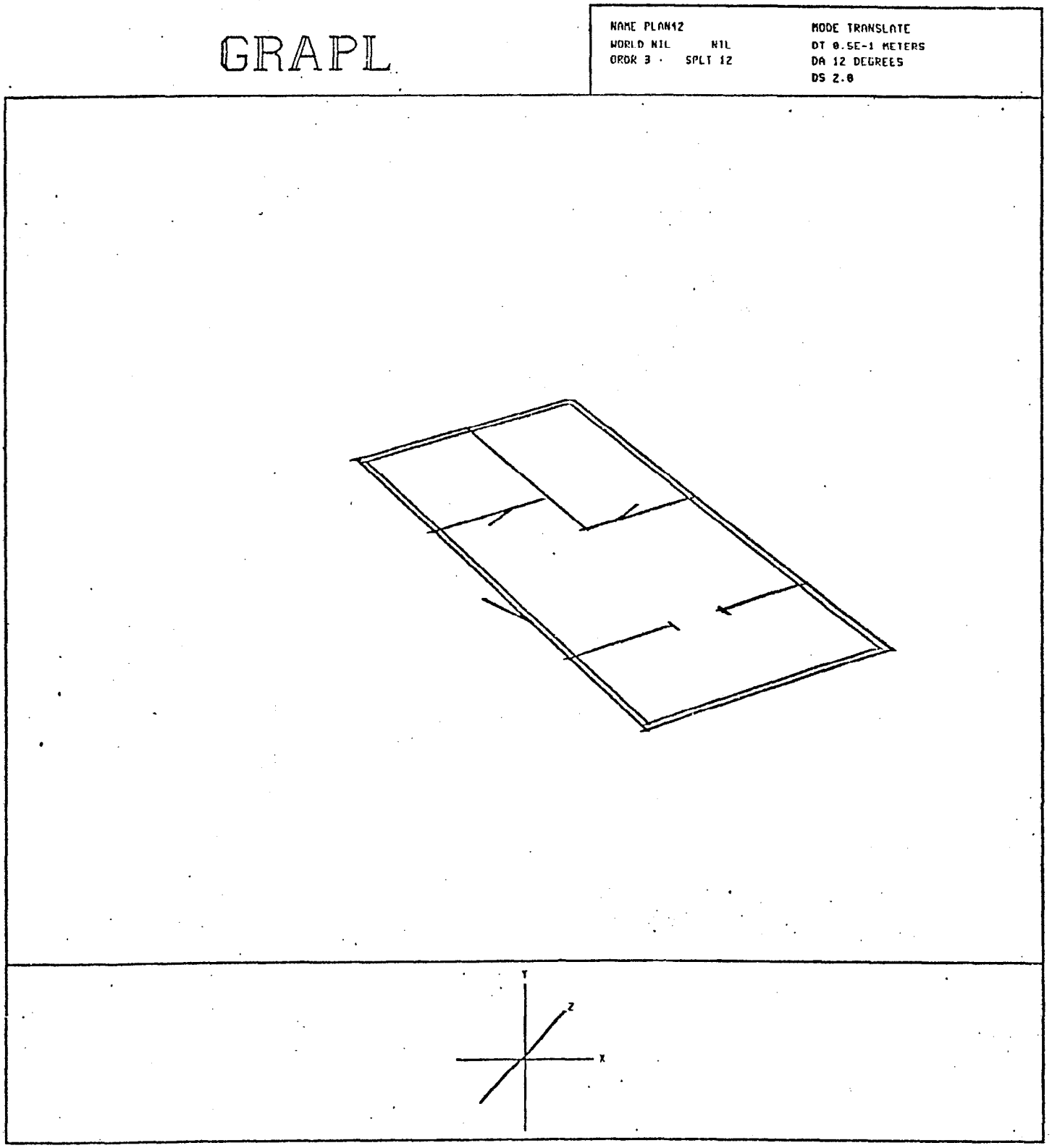


Figure 6.4.9 Floorplan #4 in perspective

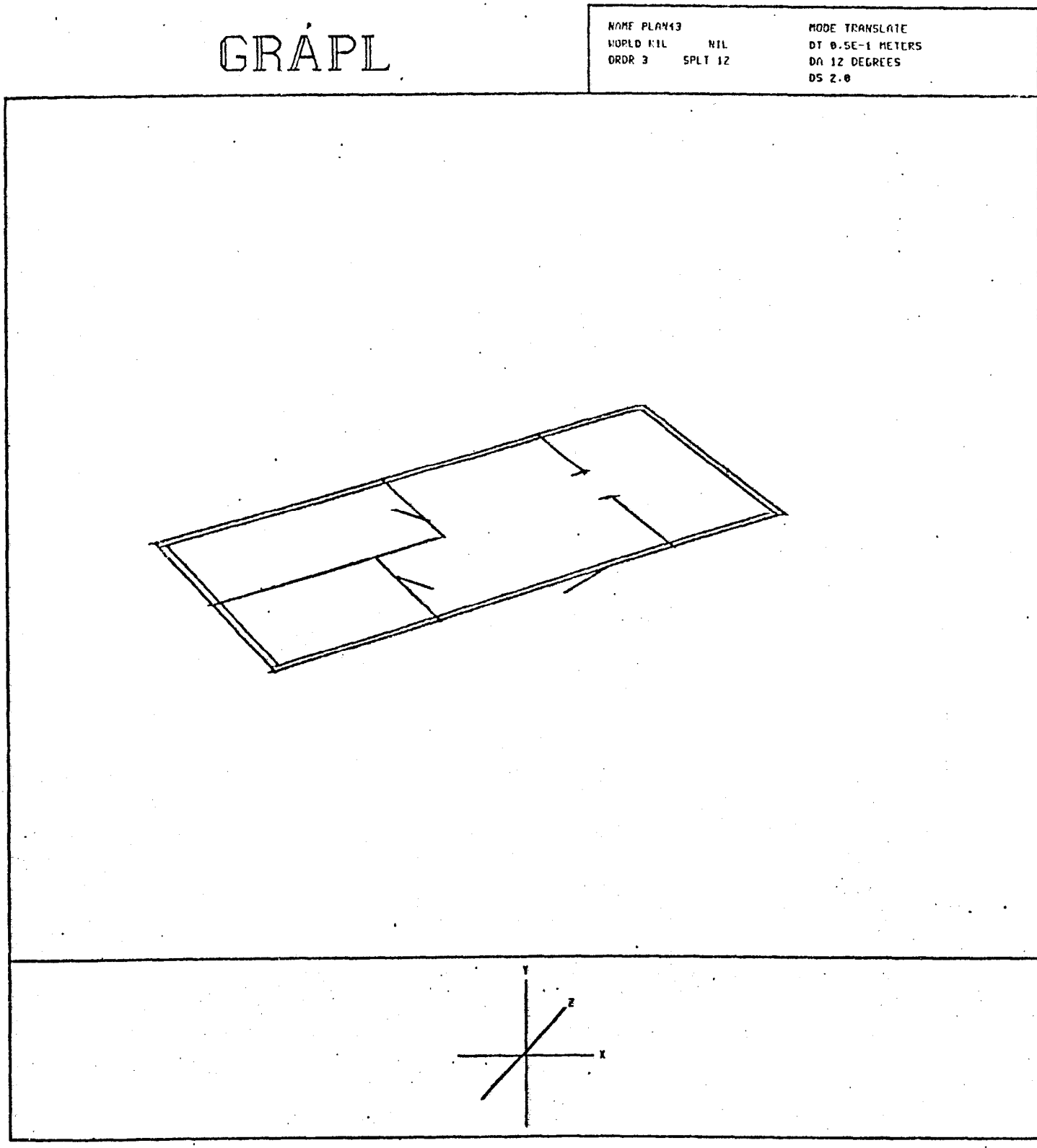


Figure 6.4.10 Floorplan #4 with interior from the top

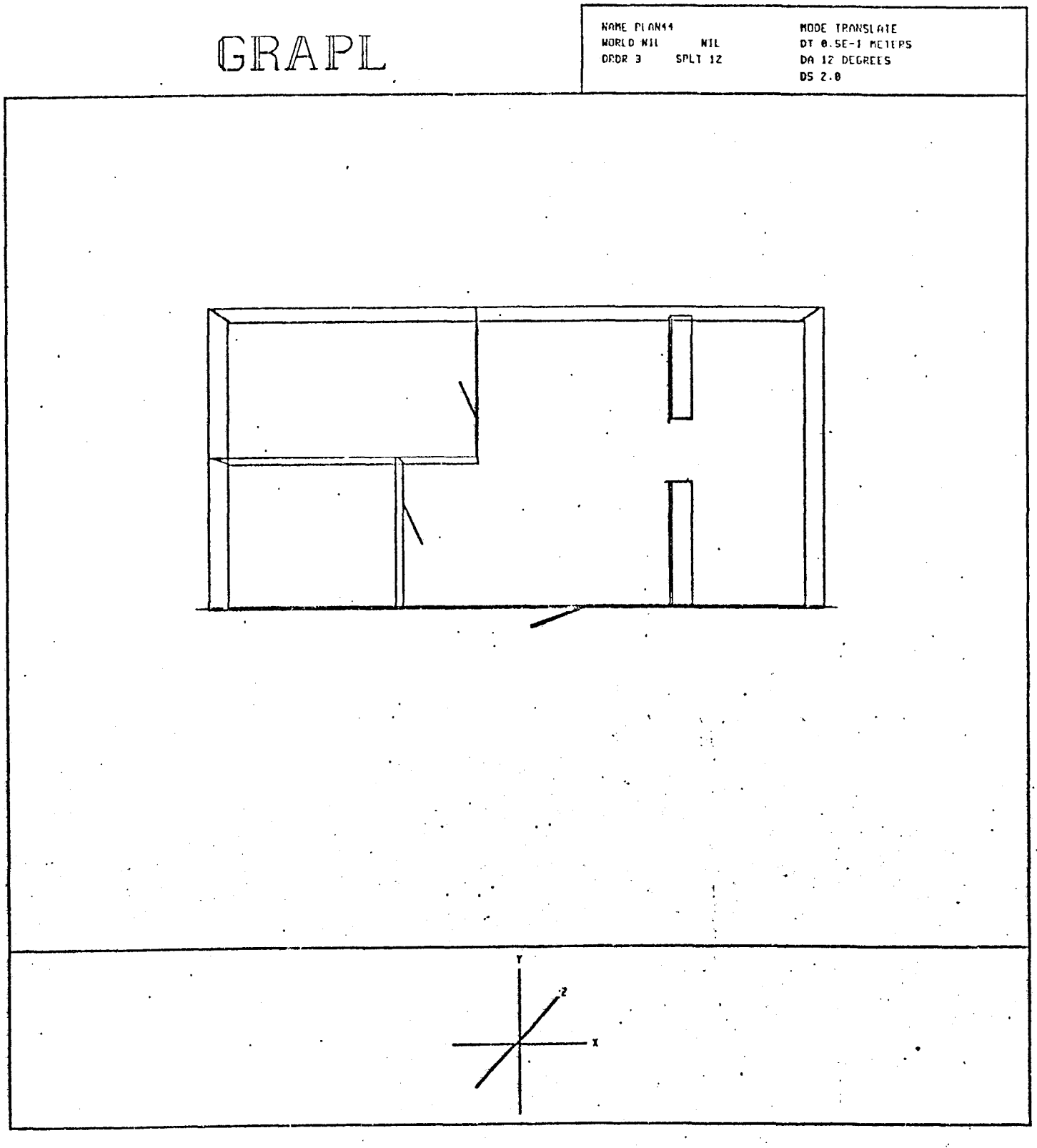
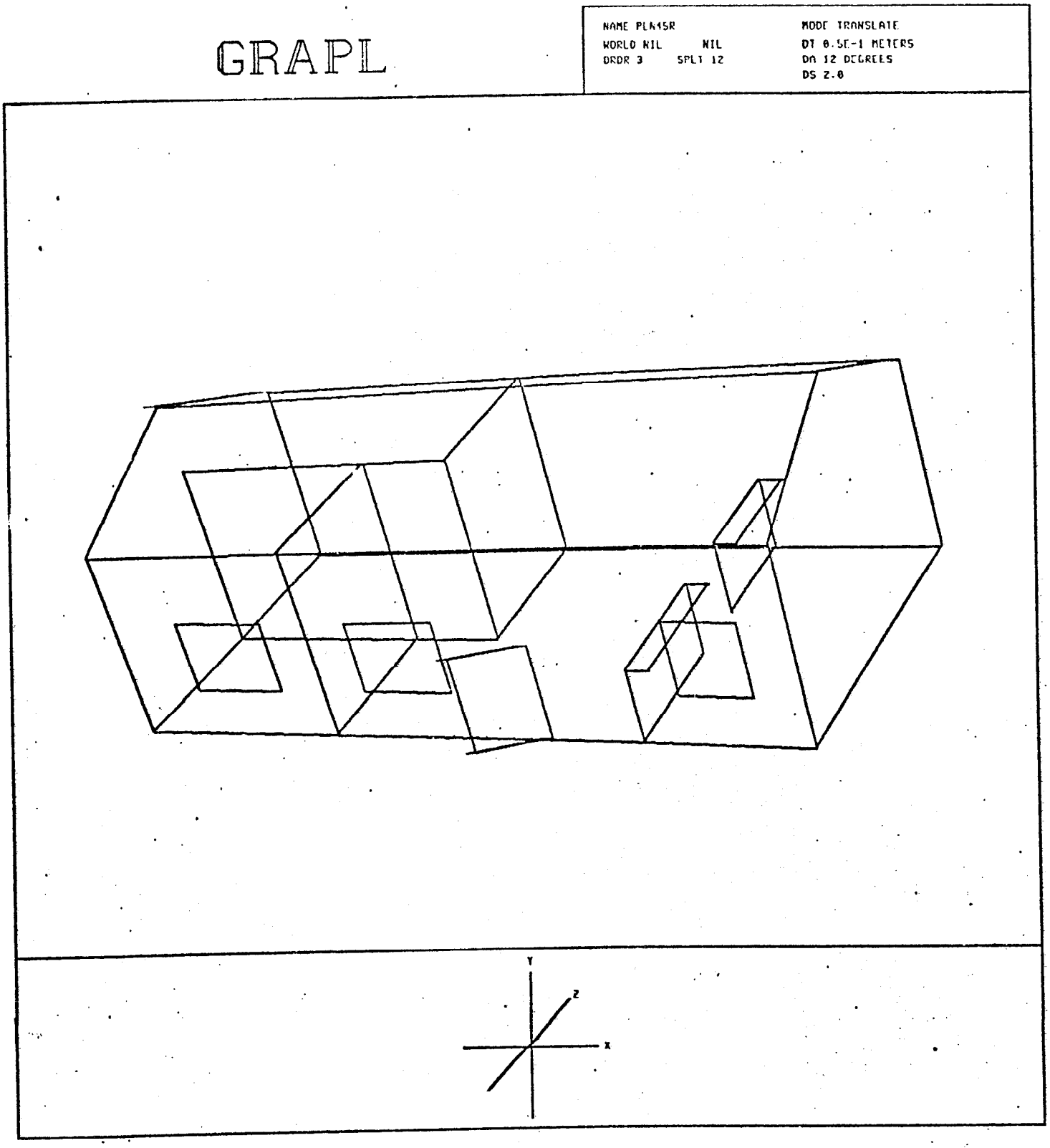


Figure 6.4.11 Floorplan #4 with interior in perspective



6.5 Some additional constructions

This section shows a small selection of illustrations generated while testing the GRAPL system.

In Figure 6.5.1 we see the side view of a compact "GM-type" automobile. Figure 6.5.2 shows the front view. In Figure 6.5.3 we have increased the depth so as to show the entire automobile; and finally, in Figure 6.5.4, we show the automobile in perspective.

We designed a contemporary building using the Stanford University Artificial Intelligence Laboratory as a model, as shown in Figure 6.5.5. In Figure 6.5.6 we brought the building closer, so as to be able to check some details. Then in Figure 6.5.7 we rotated it into a perspective view; and in Figure 6.5.8 we replicated the building six times and created a "tract home" style environment.

Figure 6.5.1 Automobile side view

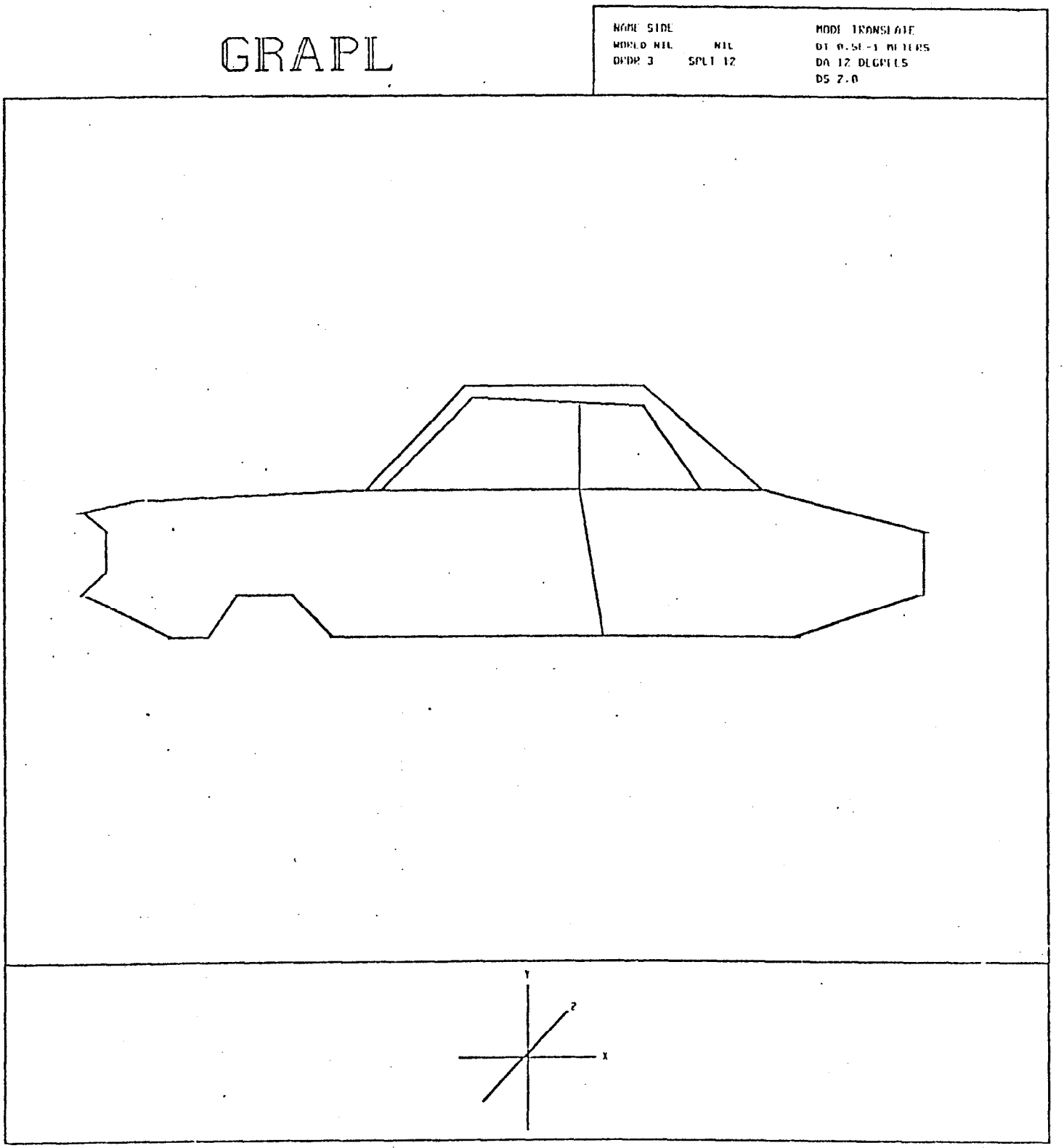


Figure 6.5.2 Automobile front view

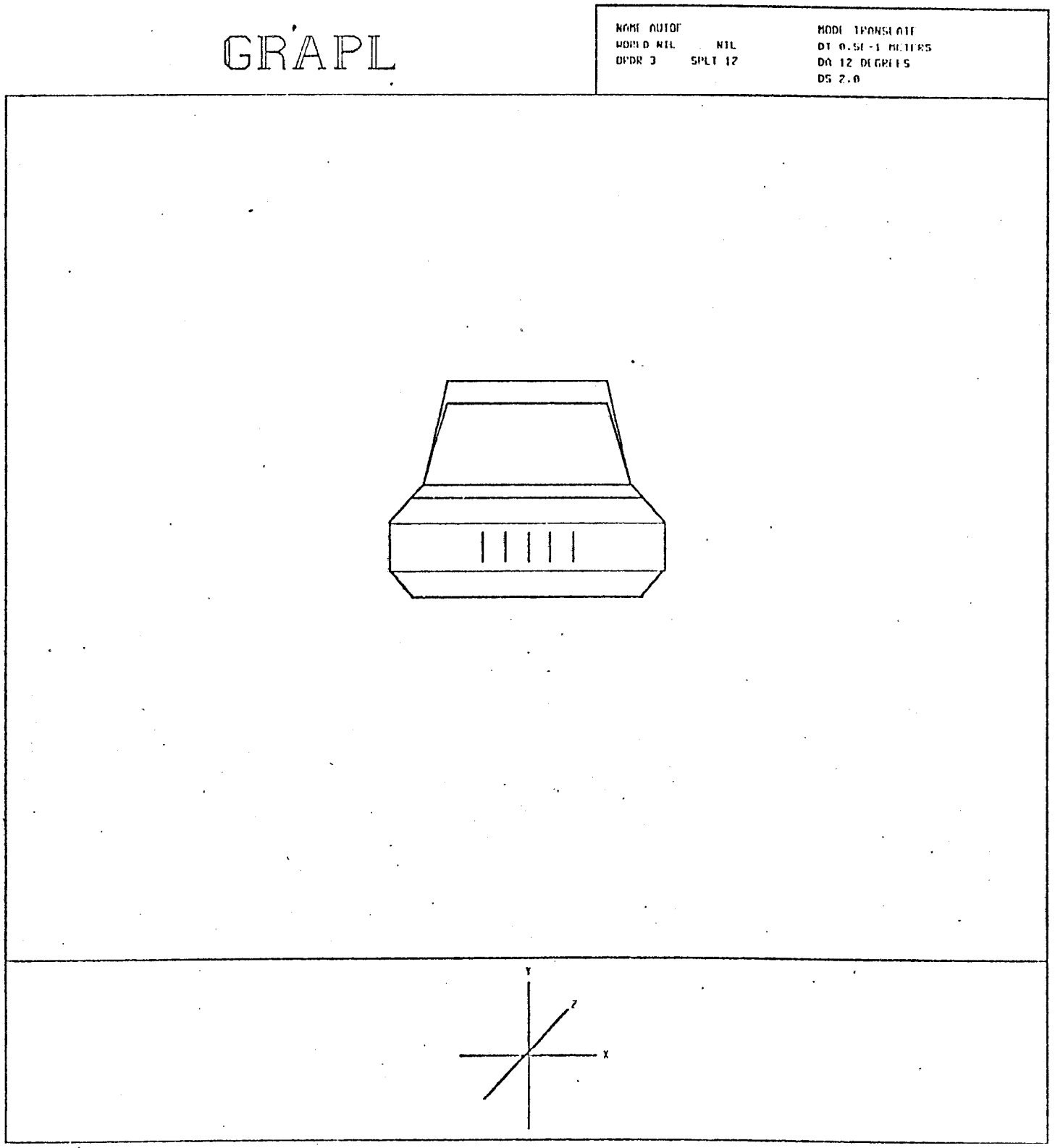


Figure 6.5.3 The entire automobile

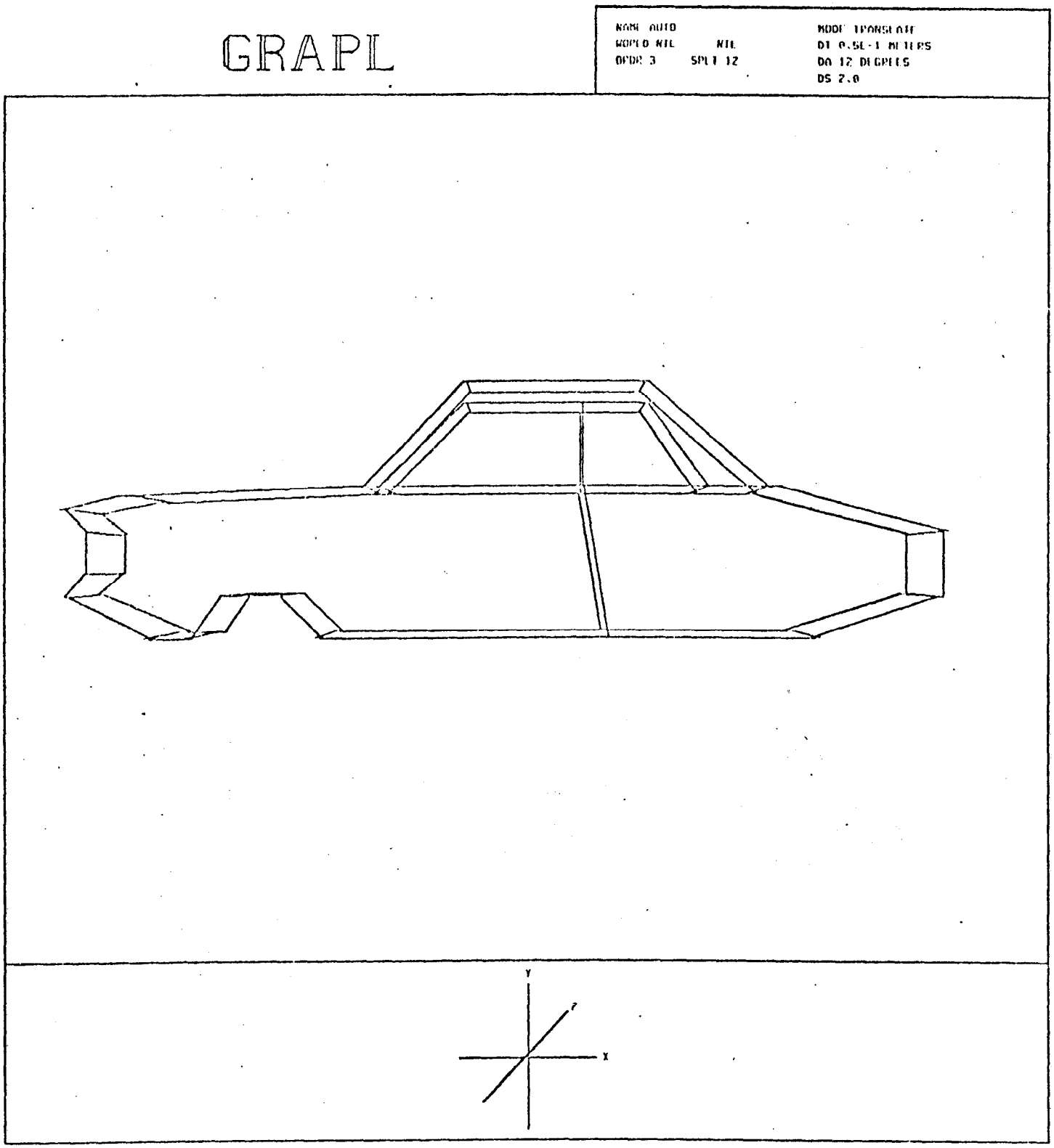


Figure 6.5.4 Automobile in perspective

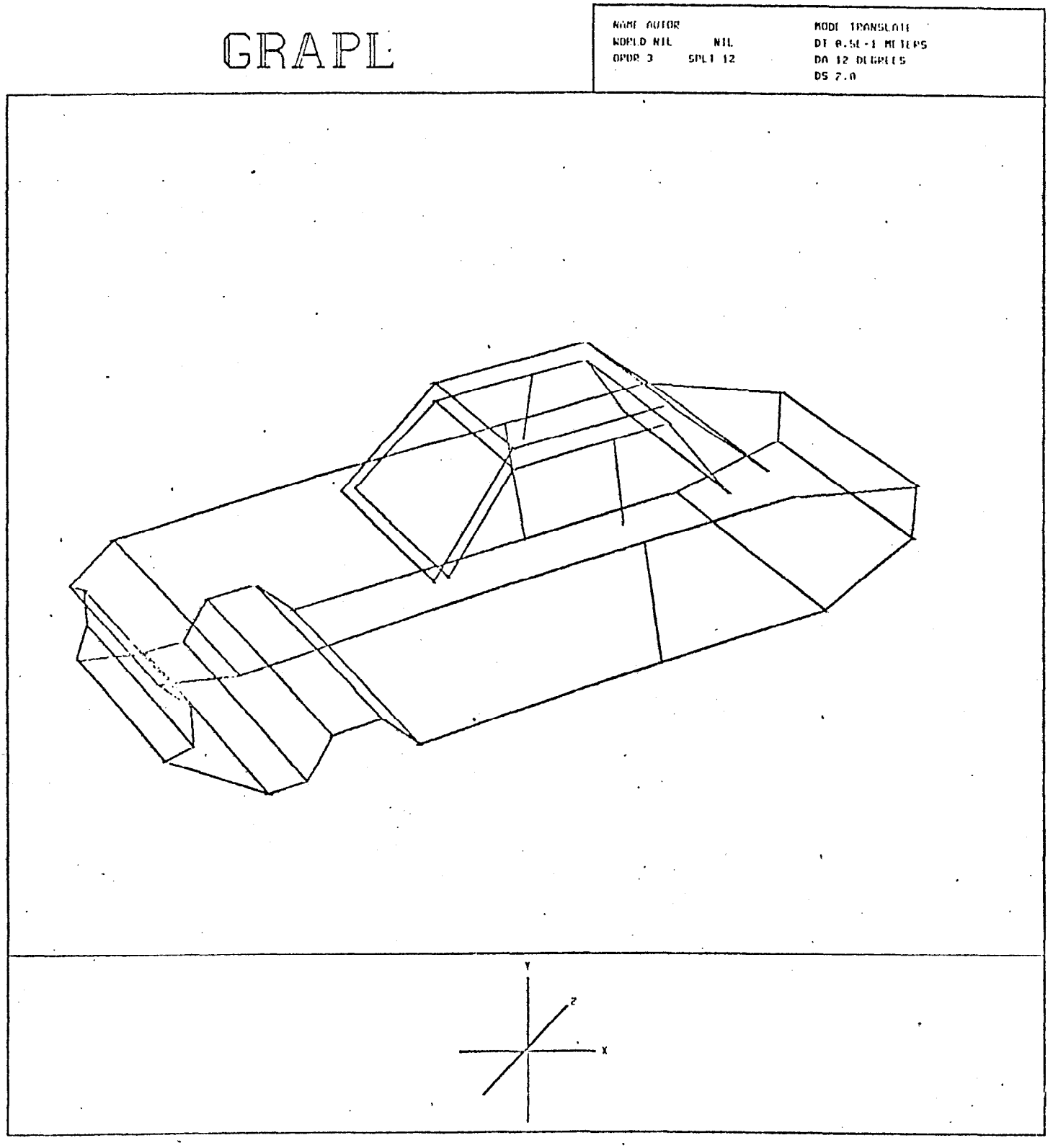
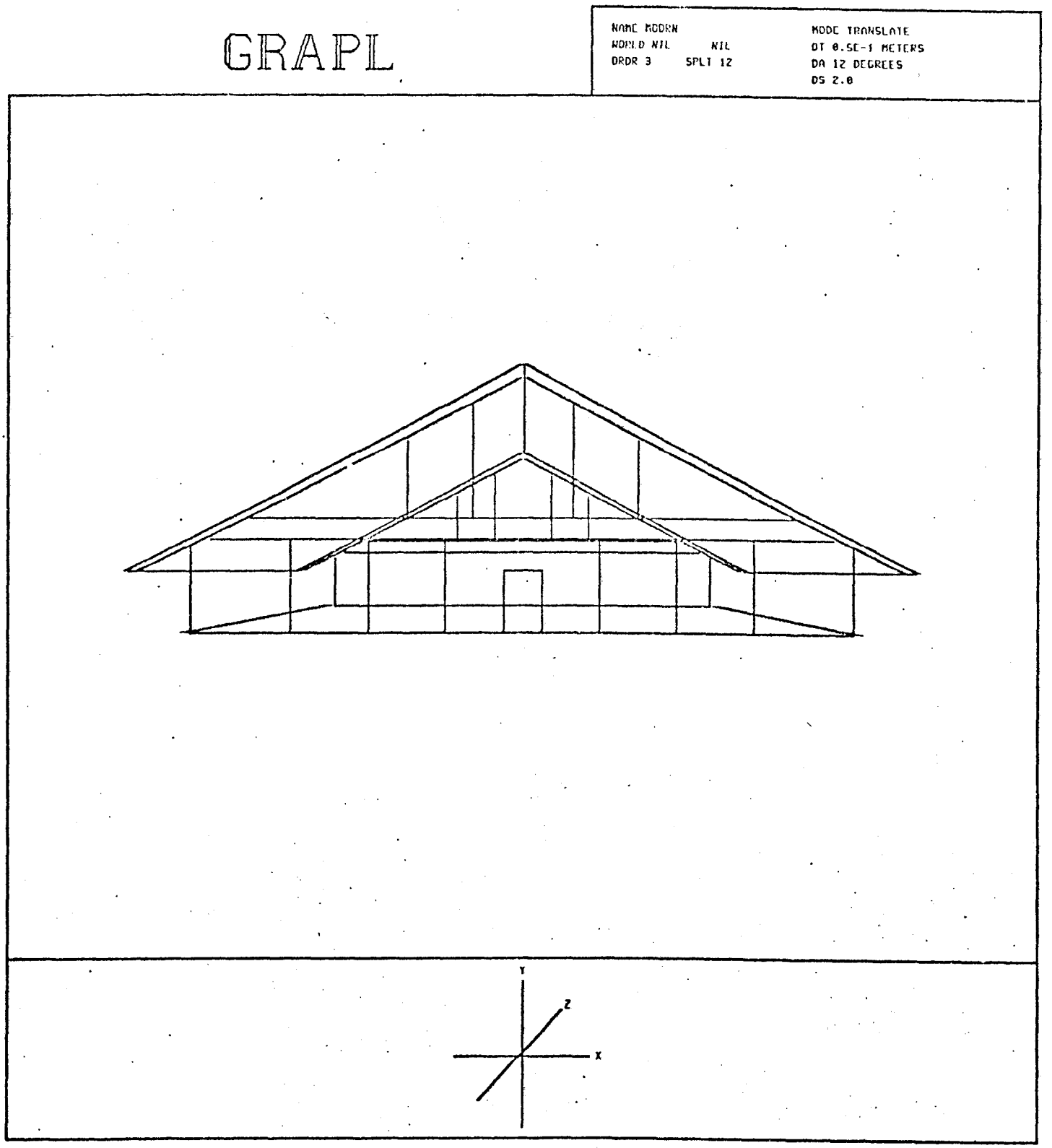


Figure 6.5.5 A contemporary building



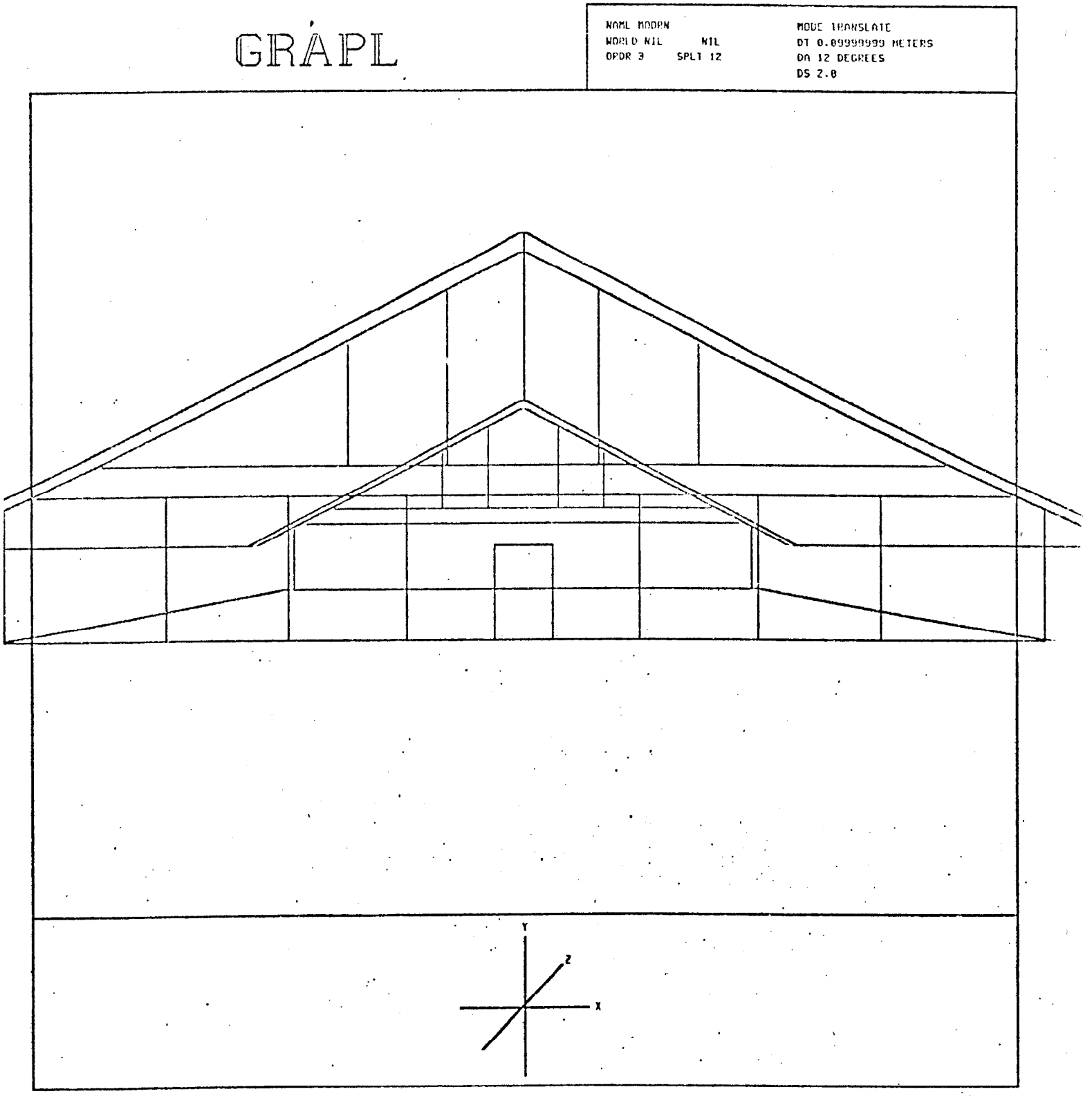

GRAPL
NAME MODEN
WORLD NIL NIL
OPDR 3 SPLT 12
MODE TRANSLATE
DT 0.09999999 METERS
DA 12 DEGREES
DS 2.0
Y
Z
X

Figure 6.5.7 The building in perspective

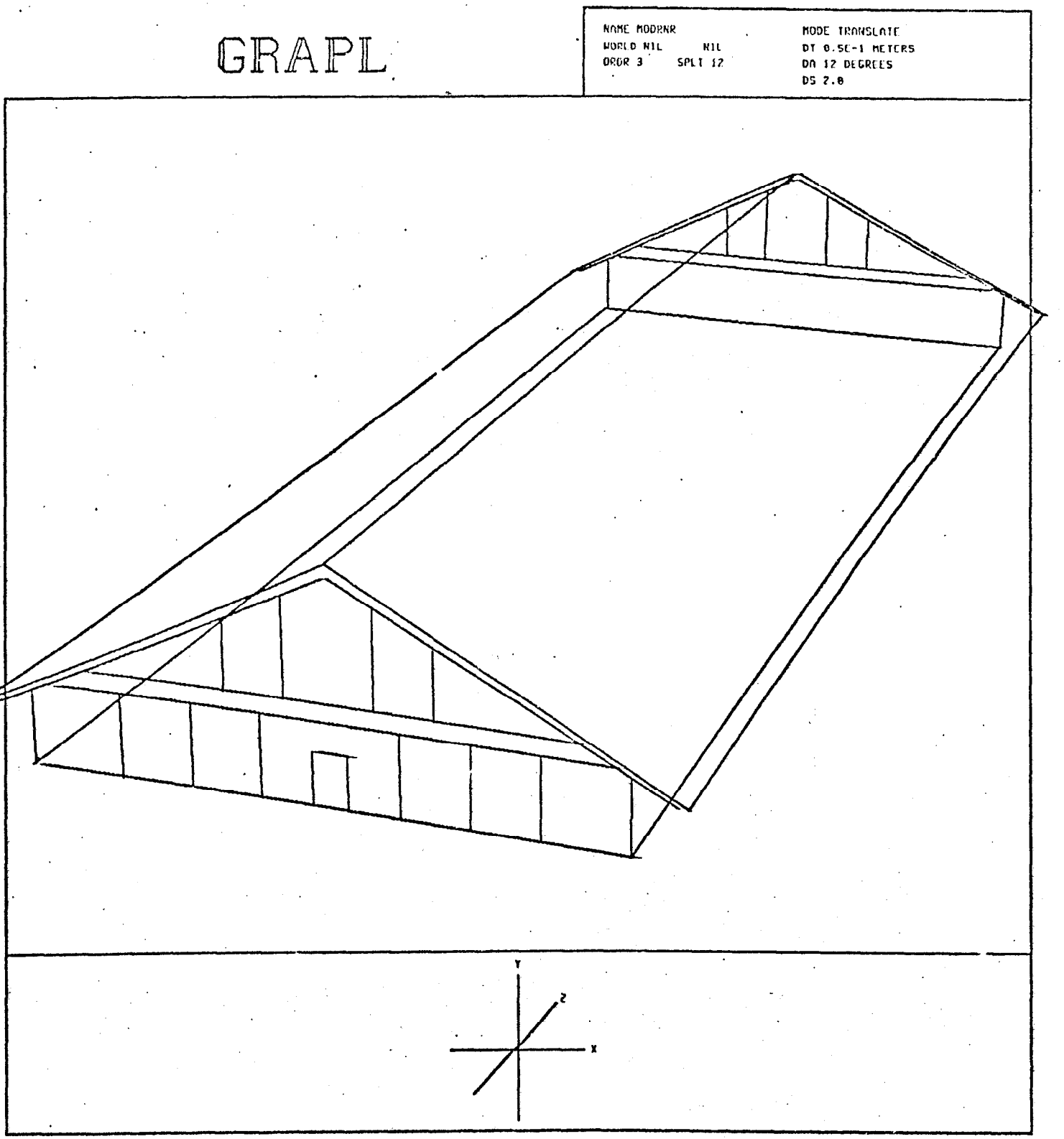
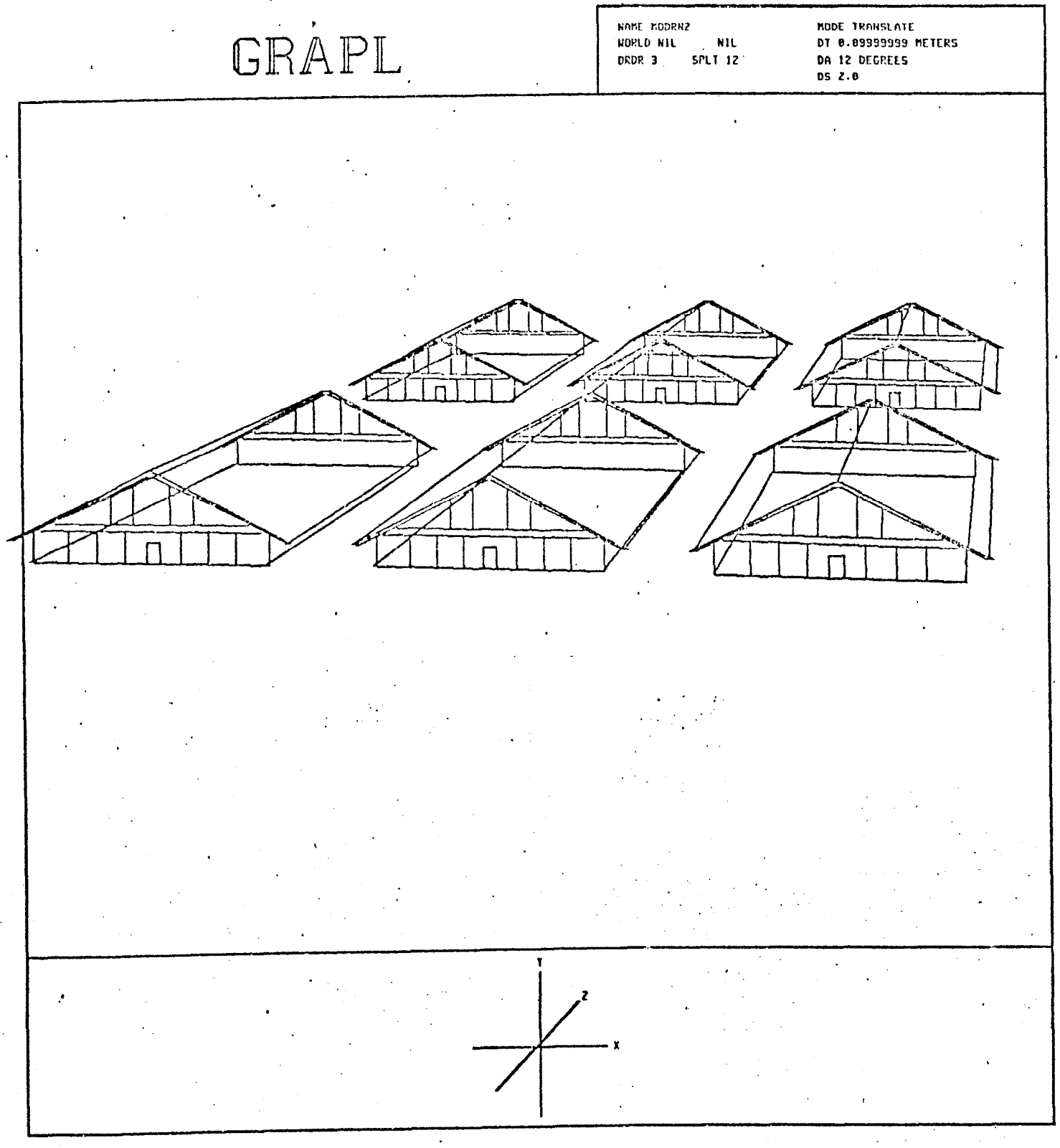


Figure 6.5.8 The building in a "tract home" setting



CHAPTER 7 02:38:08 08/04/75

7 System Performance

We present a summary of GRAPL's observed performance based on the examples of Chapter 6 and various other experiments which we have performed. We begin by discussing those aspects of the GRAPL system which are of especial interest because of their efficiency. Then we briefly summarize other aspects of system performance.

7.1 GRAPL efficiencies

The efficiency of the GRAPL system is noteworthy in three distinct (but related) areas:

1) Space: the system utilizes an efficient representation for its data objects.

2) Time: the system includes algorithms for accessing data efficiently.

3) Human: the system is efficient for the user -- it is easily learned, simply maintained, and quickly modified.

It is the combination of these factors which makes GRAPL into such an extremely powerful and versatile system.

7.1.1 Space efficiency

GRAPL's data structure is particularly efficient for representing objects of the physical world. Remembering from Chapter 4, the dual data structure consists of a heirarchy of objects plus a heirarchy of cubes (consisting of 10 levels, with 64 subcubes per cube).

The use of heirarchy to represent objects -- the use of masters and instances, templates and usages, etc. -- is common to most sophisticated graphics systems, and GRAPL realizes the same savings by its use as do other systems. This saving is expcnential in that one trades an exponential amount of storage for a corresponding amount of processing time as the depth of the heirarchy increases.

In GRAPL, however, rather than fully paying this exponential expense, only those cubes within the visual neighborhood are actually brought intc core, and it is only those visible objects within these cubes which take up memory space.

For example, let us consider the description of an 8-story building of approximately 500 square meters area and 20 meters height. The following cubes will be required in order to describe the contents of the entire structure with the resolution of a single room of about 4 cubic meters:

Cube Order		No. Req'd	
O(10)	-	1	
O(9)	-	1	
O(8)	-	1	(the whole building fits here)
O(7)	-	8	(16 meter cubes)
O(6)	-	175	(4 meter cubes)

* To display 1 room will require 5/186 of the total cube storage, or about 2.69 percent.
* The amount of cube storage is independent of the complexity within the room.
* Only that portion of the cube data structure actually would be brought into core for processing.
* This saving is realized before the heirarchy of objects within the displayed room is investigated.

The number of cubes required to represent any structure is a function of 1), the gross size of the structure; and 2), the resolution desired. The maximum number of primary cubes required to represent a structure which is no more than k meters on a side may be given by:

$$C = 11^{-m} \qquad m = \min_{j} O(j) >= k$$

The O(m) cubes are the smallest ones which may properly contain the given structure.

To gain the resolution of an r-meter cube, the C cubes above are adequate. However, if one wishes to make optimal use of memory, one should partition the O(m) cubes down to the r-meter size. This would require no more than

an additional R cubes, where R is given by:

$$R = \sum_{i=r}^{n-1} \left[k/O(i) \right]^3 \qquad \text{(where } [x] = \text{ceil}(x)\text{)}$$

The total number of cubes in the representation is then no more than C + R, and to examine a single r-meter substructure with full resolution would require only S cube accesses, where S = 11-r. The fraction of the total cube structure actually brought into core would be at most S/(C+R).

Each cube represents four words of memory: three for the base position vector, and one telling the order of the cube.

Bodies and objects are represented in more or less traditional ways, as described in Chapter 4. The number of words required to describe a structure is given by:

BODY		OBJECT	
name	1	name	1
enclosing box	6	enclosing box	6
position vector	16	position vector	16
V vertices	3*V	detail flag	1
E edges	2*E	list of L objects	L
F faces of D edges	D*F		

Bodies grown in size linearly in vertices and edges, and quadraticly in faces. Objects are linear in their complexity -- the number of subobjects.

7.1.2 Time efficiency

GRAPL is especially efficient in processing its data structure for a variety of reasons.

Access to objects within the visual neighborhood is pruned exponentially by the cube data structure, independent of the complexity of the structures within the cubes. That is, one realizes the same exponential saving in access time as one does in memory space.

GRAPL has the ability to compile both objects and cubes, thus trading increased memory for decreased processing time. The amount saved is proportional to the complexity of the object being compiled. This saving also is exponential, growing with the depth of the heirarchy. Assume that we wish to display every room of the 8-story building simultaneously (impossible due to hardware restrictions). Then the system must access all 186 cubes. However,

if we compile to O(7) - only 11 cubes accessed
if we compile to O(8) - only 3 cubes accessed

if we compile to O(9) - only 2 cubes accessed
if we compile to O(10) - only 1 cube accessed

Additionally, the system permits the user to control the level of detail displayed. This is reflected in the amount of data structure -- both cube and heirarchical -- which actually is accessed. This means that independent of structural complexity, if the visual neighborhood is of size N, access cutoff will occur at N/(10**C), where C is the cutoff factor. Thus, if we were displaying the entire 8-story building from the outside and N was slightly larger than the O(10) cube, with C=3, only 3 cubes would be accessed. In general, as C approaches 1, savings increase by 64**C cubes.

7.1.3 Human efficiency

Articulating the human efficiency is extremely difficult in the absence of an actual demonstration. Several factcrs should be considered: How easy is it to learn to use the system? How easily may be the system be changed -- the command language, the command semantics? How much effort must be expended to implement a graphically oriented project? How much for a non-graphically oriented project? How general is the system? How flexible? How "natural"? It should be clear that the answers to most if not all of these questions really are subjective; they

cannot be measured objectively.

Any modification or extension to the GRAPL system requires some understanding of the system's internal organization. We will attempt to give an impression of the magnitude of effort necessary to accomplish any significant modification. Assuming that the functional characteristics of the modification have been determined, the following must be done:

> Decide upon the syntax to be used by the user to specify the necessary commands. Implementation consists of modifying only three lines of code.
>
> Implement the semantics for the extension. This will be the single major coding effort. Using the GRAPL uniform naming conventions, demand loading of all appropriate routines will occur automatically, as will garbage collection.

For example, let us consider the implementation of a package for graphing rational functions of a single variable. We must make decisions regarding placement of axes, scaling, labeling, how to specify the function and its domain, what to do about undefined values, and so forth. Next we must decide how to invoke the package, and what commands we will use to specify each of the above items. Several alternatives present themselves: the package may be completely self-contained, using the GRAPL implementation language support alone; the package may be partially self-contained, using some of the higher-level GRAPL

functions; or the package might be implemented almost totally using GRAPL high-level routines.

As an exercise, we implemented such a graphing package using each of the three approaches (See Chapter 6). In each case, the display package consisted of less than a page of code, plus about half a page of initialization routines. Moreover, arbitrary functions of a single variable would be accepted, with no restrictions upon the function's actual form: an arbitrary GRAPL routine was acceptible; it just had to return numbers.

If one wished to implement an animation package, the procedure would be similar. Most probably the best approach would be to implement it using as many of the predefined high-level GRAPL support routines as possible, plus the operating system simulation routines. The most strenuous coding effort would be in specifying how one wished the displayed objects to change over time.

A still more ambitious effort would be the implementation of a circuit analysis package. Here most of the coding would go into deciding what kinds of circuit elements to include and how they are to be simulated.

7.2 Kernel system

In this and the following section we present a description of various aspects of the system.

The kernel system requires slightly over 55K words of memory plus 50 disk blocks. Most commonly, the system is used in under 75K of core, depending upon the complexity of the structures being described. True processing time is extremely difficult to specify due to the nature of the timesharing environment in which GRAPL is implemented. Processing time for most commands is on the order of 0.1 second. The display or compilation of large structures may take times on the order of 5-10 seconds. Effective processing time consists of these minimums plus the timesharing system load.

7.3 Operating system simulation

The operating system simulation requires approximately 1K of memory. When running with no competing timesharing users a full day's simulation requires approximately 40 seconds elapsed time. This varies, of course, depending upon the simulation time interval between events. The code occupies about three-fourth's the space of a comparable simulation in a language such as SIMULA.

CHAPTER 8 02:38:08 08/04/75

8 Conclusions and Suggestions for Further Study

In this chapter we present our conclusions based on over two years' experience with the GRAPL system and suggest areas for further study.

8.1 GRAPL's successes

The most notable GRAPL successes include the ease with which the system may be modified, the dual data structure representation, the parallel system design, and the uniformity of implementation. GRAPL demonstrates a uniform solution of a set of problems in system design and implementation.

8.1.1 Simultaneous computations

The design process more and more requires the calculation of various quantities and the deduction of performance of a wide variety of interrelated objects (through simulation, if no other means is available). GRAPL provides an environment where these calculations, simulations, deductions, and models all may be carried out

simultaneously with the design process. Moreover, in GRAPL it is precisely this capability that yields such a responsive, attractive system.

8.1.2 Notification and "posted" actions

Additionally, not only is it possible to calculate and compute in parallel with design, but it is possible to request GRAPL to notify the user when an arbitrary condition in the design process becomes true, and automatically to take some specified action. For example, cost overruns become obvious immediately, therefore; and corrective actions may be made at the time the error is made.

8.1.3 Variations

Because it is possible to freeze GRAPL's state at any time it is possible to attempt variations of design approaches, do calculations on feasibility and the like, yet return to the original plan if it is desired.

8.1.4 Easily modified command language

The command language is quickly and easily modified. For most changes only a minimal acquaintance with the GRAPL system is required. Extensions may be of any

nature whatsoever; and one has the advantage of a general purpose algorithmic language at one's disposal with which to implement them.

8.1.5 Efficient access into data structures

Although the hidden line problem has been solved for various classes of structures, it is not necessary to apply any restrictions to the types of objects displayed in GRAPL, nor is it necessary to apply hidden line/surface algorithms to the entire data structure in order to create a more pleasing view.

8.1.6 Portability

The GRAPL system is completely portable. The basic requirements for the implementation on any computer system are the existence of a LISP interpreter or compiler and a direct view storage tube type display. If one were interested in utilizing different display devices, plasma panels for example, the modification of GRAPL to produce true display files is minor.

8.2 GRAPL's shortcomings

GRAPL has several shortcomings, as do most large

systems. Since hindsight usually is so much better than forsight,these deficiencies have become more and more apparent with time, and now "haunt" us.

8.2.1 Response time

The major shortcoming of the system is in its extremely slow response time. This primarily is due to two factors: One is the fact that the system is interpretive, and the particular time-sharing environment in which it was implemented. The other shortcoming is the file environment of the time-sharing system.

The response problem might be partially solved in one of two ways. Compilation of the major GRAPL routines would yield a factor of 5 to 10 in speed. Dedication of a portion of the system to GRAPL, including locking the GRAPL system into core to eliminate the necessity for swapping could yield an additional factor of 5 to 10. Were one able to implement the system with a dedicated machine with no time-sharing overhead, the performance of GRAPL would be comparable to that of most current interactive systems.

8.2.2 System command language

The command language was developed as the GRAPL

system took shape, and thus shows the influence of "growth." For the most part, it is uniform, deals with constructs in a consistent manner, and is relatively natural. However, the addition of a light-pen capability would greatly enhance the ability to identify parts of objects and parts of structures. The motion commands are adequate, but should be implemented with some form of joy stick (or graphical equivalent).

The command language is relatively "unforgiving" in that if a command is ill-formed in any way, the system will comment to that effect and abort processing. What would be more desirable is a more sympathetic and helpful facility -- one which attempted to aid the user in the correct formulation of what he is trying to say.

8.3 Suggestions for further study

The process of research never quite ends; there always is another approach to be investigated, an alternative not taken earlier (or not possible earlier), or new ideas to incorporate. In this section we present some possible avenues for further research which we believe it would be fruitful to pursue.

8.3.1 A proposed sketching command language

We wish to accomodate the designer with full sketching capabilities. Essentially what we propose is to simulate "ideal paper," but retain the information in a very different form. The sketching operations are of course device independent, but there is an explicit assumption that one would use some kind of "artistic" input hardware such as a stylus, light pen, mouse, or joy stick. Furthermore, corresponding to the operations of erasing, redrawing at a different scale or point of view, and so forth, are Sketch Mode commands achieving the same results.

The Sketch Mode commands we propose initially are:

SKETCH - Track the input device and trail a line

ERASE - Erase along the track with given width

SURROUND - Enclose the designated lines and treat them as a unit

COPY - Copy the designated object to a new position, orientation, and scale

MOVE - Move the designated object to a new position, orientation, and scale

SCALE - Lengthen or contract lines, scale up or scale down objects

MOVE DESTRUCTIVE - Move the designated object to a new position, orientation, scale replacing what was previously there

EXPAND - Create a hole into which new objects may be placed

SMOOTH IT - Smooth lines, make them straight, arcs, or elipses, etc., connect near vertices, force parallelism, perpendicularity, angularity, and so forth. Essentially this is a map into the SKETCHPAD domain

RUBBER BAND - Rubber band line drawing

It should be noted that with these few commands we already have a system which is reasonably sophisticated in comparison to ordinary paper. Yet we have retained the freeness of expression and the lack of constraint of usual drafting systems.

The manner in which we propose to store sketched information is of some interest. The system uses objects which we call "analog patches." These are essentially small matrices of grey-levels which represent the lines and surfaces described. Obviously, the store would overflow quickly if one attempted to create too large or complex a structure. But this is precisely the tradeoff between generality ard flexibility versus program and data size. One does not ccmmonly sketch the most detailed objects at the same time as one blocks in the overall size and shape of a structure; so, we feel relatively secure in providing this capability. Whenever the system finds that the display time or, in general, the processing cost for much sketched information becomes too high, it should begin to smooth things by itself and might ask the user to do so as well. And of course, the user would always have the option of converting his sketch into a more final form, greatly reducing the processing cost.

8.3.2 Shadows, grey-scale, color, texture

The addition of shadows using one or more simulated light sources might be very valuable for structural designs where the effects of the sun play a major role in the heating and cooling requirements of a building. Research is currently being pursued in this direction both by the University of Utah and independent architectural firms such as Skidmore, Owens, and Merrill.

The uses of grey-scale and color have been investigated in some detail, especially at the University of Utah. Perhaps the most advanced digital color pictures have been developed there. Incorporating the results of that research might easily be done.

The display of texture is a current area of research in graphics. It is a rather difficult problem and not too much success has been obtained to date.

8.3.3 Stereo, exploded views, curves, and surfaces

The generation of stereo views for display on some stereoscopic device is currently available in GRAPL. One might wish to add the user commands to facilitate this display.

Exploded views are especially useful in the construction of aggregates of complex parts. The addition of this capability would necessitate the modification of the display and drawing routines. The theory behind exploded view generation is well-known. This addition would present little difficulty.

Addition of arbitrary curves and surfaces would involve the creation of some additional data structures. A careful series of extensions to GRAPL in this direction, incorporating the most recent works of Coons, Forrest, and Bezier, might be done.

8.3.4 Solution by analogy

The solution of problems by analogy to already known solutions (or problems) is one of the areas currently being investigated by workers in Artificial Intelligence. This certainly is a capability one would wish to have in an interactive environment. The user then could specify the computation of his various requirements and constraints by either giving their explicit formulas (or programs) or by referring to already known formulas (or programs) and specifying how the new computations are similar or different from the old.

8.3.5 Various other partitioning algorithms

The current cube partitioning algorithm certainly is not optimal. It was chosen primarily on the basis of ease of implementation and the fact that it seemed to meet our requirements at the time.

We would like to do a series of experiments implementing partitioning algorithms based on:

1) A better measure of the complexity of the structure of the cube

2) A dynamic complexity measure, rather than a static measure

3) An algorithm which was related to the cube's usage, rather than to its structure, and a combination of both usage and structure

4) An algorithm based on the size of the display neighborhood

8.3.6 Other measures of an object's complexity

The complexity measure currently used in GRAPL is merely a count of the number of subobjects in the enclosing cube at the first description level. Complexity in the real world certainly is proportional to the structural complexity of objects, but it would be interesting to investigate measures based on cost, size, volume, ease of construction,

time for construction, and other objective and subjective values.

8.3.7 Clipping and hidden line/surface removal

The addition of interactive clipping and hidden line/surface removal would be of special interest in producing more pleasing displays, rather than including it in the post-processor as is currently done.

Of course, the most desirable approach is to purchace or build special purpose hardware for these tasks. However, if one wished to implement these functions in the display software, the most rewarding route probably would be to implement special commands for display in clipping, hidden line/surface mode. Alternatively, it would be extremely easy to describe the new display algorithms, and then load them into GRAPL at will. One can conceive of a collection of display algorithms; GRAPL using the one specified by the user. An excellent summary of current software techniques for hidden surface removal may be found in a recent Computer Surveys article by Sutherland, Sproull, and Schumacker <SS 74>.

CHAPTER 9 02:38:08 08/04/75

9 Bibliography

This bibliography primarily contains those works actually cited in the text and those which bear most directly on our research. A very comprehensive bibliography on computer graphics (the fruit of a survey of the literature made prior to and during this research) was published in 1972 <Po 72a>.

An excellent text on interactive computer graphics including a good bibliography has been published by Newman and Sproull <NS 73>.

<Ab 71> Abrams, M.D.
Data structures for computer graphics.
In <TW 71>, 268-286.

<AC 68> Ahuja, D.V., and Coons, S.A.
Geometry for construction and display.
In <IB 68>, 188-205.

<Al 64> Alexander, C.
Notes on the synthesis of form.
Harvard University Press, Cambridge, Mass., (1964).

<Ap 66> Appel, A.
The visibility problem and machine rendering of solids.
IBM T. J. Watson Research Center, New York, Rept. No. RC 1618, (May 1966).

<Ap 67> Appel, A.
The notion of quantitative invisibility and the

machine rendering of solids.
Proc ACM 22nd National Conf. 1967, Thompson Book Co., Washington, D. C., 387-393.

<Ap 68a> Appel, A.
Some techniques for shading machine renderings of solids.
Proc AFIPS 1968 SJCC, vol 32, 37-49.

<Ap 68b> Appel, A.
Modeling in three dimensions.
In <IB 68>, 310-321.

<AS 72> Appel, A., and Stein, A.
A system for the interactive design of polyhedra.
IBM T. J. Watson Research Center, New York, Rept. No. RC 3804, (Apr 1972).

<Ba 67> Balzer, R.
Dataless programming.
The RAND Corporation, Santa Monica, Calif., Memo RM-5290-ARPA, condensed version in Proc AFIPS 1967 FJCC, vol 31, 535-544.

<BM 68> Baskin, H.B., and Morse, S.P.
A multilevel modeling structure for interactive graphics design.
In <IB 68>, 218-228.

<BB 64> Berkeley, E.C., and Bobrow, D.G., (eds.)
The programming language LISP: its operation and applications.
Information International, Inc., Cambridge, Mass., (1964).

<Be 71> Berry, D.M.
Introduction to Cregeno.
In <TW 71>, 171-190.

<Bo 69> Bouknight, W.J.
An improved procedure for generation of half-tone computer graphics presentations.
Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, Tech. Rept. No. R-432, (Sep 1969).

<Bo 70> Bouknight, W.J.
A procedure for generation of three-dimensional half-toned computer graphics presentations.
Comm ACM 13, 9 (Sep 1970), 527-536.

<BK 70> Bouknight, W.J., and Kelley, K.

An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources.
Proc AFIPS 1970 SJCC, vol 36, 1-10.

<Ca 69> Carr, C.S.
Geometric modeling.
Computer Science Department, University of Utah, Salt Lake City, Utah,
Tech. Rept. No. TR 4-13, (Jun 1969).

<Ch 69> Cheatham, T.E., Jr.
Motivation for extensible languages.
In <CS 69>, 45-49.

<CS 69> Christensen, C., and Shaw, C.J.
Proceedings of the extensible languages symposium.
SIGPLAN Notices 4, 8 (Aug 1969).

<CL 69> Cohen, D.J., and Lee, T.M.P.
Fast drawing of curves for computer display.
Proc AFIPS 1969 SJCC, vol 34, (1969).

<Co 68a> Comba, P.G.
A procedure for detecting intersections of three dimensional objects.
J ACM 15, 3 (Jul 1968), 354-366, earlier version: IBM New York Scientific Center, New York, New York, Rept. No. 39.020, (Jan 1967).

<Co 68b> Comba, P.G.
A language for three-dimensional geometry.
In <IB 68>, 292-308.

<Co 63> Coons, S.A.
An outline of the requirements for a computer-aided design system.
Proc AFIPS 1963 SJCC, vol 22, (1963), 299-304.

<Co 67> Coons, S.A.
Surfaces for computer aided design of space forms.
Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., Rept. No. MAC-TR-41, (Jun 1967).

<DN 66> Dahl, O.J., and Nygaard, K.
SIMULA -- an ALGOL-based simulation language.
Comm ACM 9, 9 (Sep 1969), 671-678.

<DM 70> Dahl, O.J., Myhrhaug, B., and Nygaard, K.
Common base language. (SIMULA 67)
Norwegian Computing Center,
Publication No. S-22, (May 1970).

<De 68> Dennis, J.B.
Programming generality, parallelism, and computer architecture.
Proc IFIP 1968 Congress, (1968), c1-c7.

<DV 66> Dennis, J.B., and van Horn, E.C.
Programming semantics for multiprogrammed computations.
Comm ACM 9, 3 (Mar 1966), 143-155.

<Ea 69> Earley, J.
VERS -- an extensible language with an implementation facility.
Computer Science Department, University of California, Berkeley, Calif., (1969).

<Ea 71> Earley, J.
Toward an understanding of data structures.
Comm ACM 14, 10 (Oct 1971), 617-627.

<FN 69> Faiman, E.M., and Nievergelt, J., (eds.)
Pertinent concepts in computer graphics.
University of Illinois Press, (1969).

<Fi 70> Fisher, D.A.
Control structures for programming languages.
Doctoral Dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania, (May 1970).

<Fo 68> Forrest, A.R.
Curves and surfaces for computer aided design.
Cambridge University, England,
CAD Group Ph.D. Thesis, (Jul 1968).

<Fo 70> Forrest, A.R.
Interpolation and approximation by Bezier polynomials.
Computer Aided Design Group, Cambridge University, England, CAD Group Document 45, (Oct 1970).

<Fr 70> Frankel, A.
What is the design process?
In The Use of Computers in Engineering Design, Furman, (Ed.), English Universities Press, (1970).

<Fr 71> Fraser, A.G.
On the meaning of names in programming systems.
Comm ACM 14, 6 (Jun 1971), 409-416.

<GM 69> Galimberty, R. and Montanari, U.
An algorithm for hidden line elimination.

Comm ACM 12, 4 (Apr 1969), 206-211.

<Ga 69> Garwick, J.V.
GPL, a general purpose language.
In <CS 69>, 6-8, an earlier version in
Comm ACM 11, 9 (Sep 1968), 634-638.

<Ge 71> George, J.E.
GEMS - A graphical experimental meta system.
Computer Science Department, Stanford University,
Rept. No. STAN-CS-71-227, (Aug 1971).

<Go 71> Gouraud, H.
Computer display of curved surfaces.
Computer Science Department, University of Utah,
Salt Lake City, Utah,
Rept. No. UTEC-CSc-71-113, (Jun 1971).

<He 71> Hewitt, C.
Description and theoretical analysis (using
schemata) of PLANNER: a language for proving
theorems and manipulating models in a robot.
Doctoral Dissertation, Massachusetts Institute of
Technology, Cambridge, Mass., (Jan 1971).

<Hu 71> Huffman, D.A.
Impossible objects as nonsense sentences.
Machine Intelligence 6, (1971), 295-323.

<IM 69> Ichbiah, J.D., and Morse, S.P.
General concepts of the SIMULA 67 programming
language.
Companie Internationale pour l'Informatique,
(Dec 1969).

<IB 68> International Business Machines Corporation.
Interactive graphics in data processing.
IBM Systems Journal 7, 3 & 4 (1968).

<Ir 69> Irons, E.T.
The extension facilities of IMP.
In <CS 69>, 18-19.

<Jo 63> Johnson, T.E.
SKETCHPAD III: a computer program for drawing in
three dimensions.
Proc AFIPS 1963 SJCC, vol 22, 347-353.

<Jo 69> Jorrand, P.
Some aspects of BASEL, the base language for an
extensible language facility.
In <CS 69>, 14-17.

<Ka 70> Kaneff, S., (ed.)
Picture language machines.
Proceedings of a conference held at the Australian National University, Canberra, on 24-28 February, 1969,
Academic Press, New York, New York, (1970).

<Ka 69> Kay, A.C.
The reactive engine.
Doctoral Dissertation, Computer Science Department, University of Utah, Salt Lake City, Utah, (1969).

<Ke 69> Kelley, K.
A computer graphics program for the generation of half-tone images with shadows.
Coordinated Science Laboratory, University of Illinois, Urbana, Illinois,
Tech. Rept. No. 444, (Nov 1969).

<Ko 67> Koestler, A.
The act of creation.
Dell Publishing Co., New York, New York, (1967).

<Ku 68a> Kubert, B.B.
A computer method for perspective representation of curves and surfaces.
Aerospace Corp., San Bernadino, Calif., (Dec 1968).

<Ku 68b> Kulsrud, H.E.
A general-purpose graphic language.
Comm ACM 11, 4 (Apr 1968), 247-254.

<Le 69a> Lee, T.M.P.
Three-dimensional curves and surfaces for rapid computer display.
Harvard University, Cambridge, Mass.,
Tech. Rept. No. ESD-TR-69-189, (Apr 1969).

<Le 69b> Lee, T.M.P.
A class of surfaces for computer display.
Proc AFIPS 1969 SJCC, vol 34, (1969).

<Lo 67a> Lombardi, L.A.
Incremental computation.
In Advances in Computers, vol 8, (1967), 247-333.

<LR 64> Lombardi, L.A., and Raphael, B.
LISP as the language for an incremental computer.
MIT Project MAC, Memo MAC-M-142, (Mar 1964).

<Lo 67b> Loutrel, P.

A solution to the hidden-line problem for computer-drawn polyhedra.
Electrical Engineering Department, New York University, New York, New York,
Tech. Rept. No. 39.020, (Jan 1967), and
Tech. Rept. No. 400-167, (Sep 1967).

<Lo 67c> Loutrel, P.
Determination of hidden edges in polyhedral figures: convex case.
Laboratory for Electroscience Research, New York University, New York, New York,
Tech. Rept. No. 400-145, (Sep 1966).

<Lo 70> Loutrel, P.
A solution to the hidden-line problem for computer-drawn polyhedra.
IEEE Transactions on Computers, C-19,
(Mar 1970), 205.

<Ma 72> Mahl, R.
Visible surface algorithms for quadric patches.
IEEE Transactions on Computers, C-21, 1 (Jan 1972),
Earlier version: Computer Science Department, University of Utah, Salt Lake City, Utah,
Rept. No. UTEC-CSc-70-111, (Dec 1970).

<Ma 69> Matsushita, Y.A.
A solution to the hidden line problem.
Computer Science Department, University of Illinois, Urbana, Illinois, Tech. Rept. No. 335, (Jun 1969).

<MS 70> McCallister, S., and Sutherland, I.E.
Final report on the area Warnock hidden line algorithm.
Evans and Suterland Computer Corp., Salt Lake City, Utah, Internal Document, (Feb 12 1970).

<MW 71> McGowan, C., and Wegner, P.
The equivalence of sequential and associative structure models.
In <TW 71>, 191-216.

<Mi 70> Mitchell, J.G.
The design and construction of flexible and efficient interactive programming systems.
Doctoral Dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania, (Jun 1970).

<MA 68> MAGI, Mathematical Applications Group, Inc.
3-D simulated graphics.
Damation, 14, 2 (Feb 1968), 69.

<Ne 70> Negroponte, N.
The architecture machine.
The MIT Press, Cambridge, Mass. (1970).

<Ne 68> Newman, W.M.
A system for interactive graphical programming.
Proc AFIPS 1968 SJCC, vol 32, 47-54.

<Ne 71> Newman, W.M.
Display procedures.
Comm ACM 10, 14 (Oct 1971), 651-660.

<NS 73> Newman, W.M., and Sproull, R.F.
Principles of Interactive Computer Graphics.
McGraw-Hill, (1973).

<Pa 68> Pankhurst, R.J.
GULP -- a compiler-compiler for verbal and graphic languages.
Joint Computer Aided Design Project, University of Cambridge, England, Rept. No. 68-274, (1968), also Proc 23rd ACM National Conference, (1968), 405-421.

<Pe 69> Perlis, A.J.
Introduction to extensible languages.
In <CS 69>, 3-5.

<Po 71> Pollack, B.W.
An annotated bibliography on the construction of compilers.
Computer Science Department, Stanford University, Rept. No. STAN-CS-71-249, (Dec 1971).

<Po 72a> Pollack, B.W.
A bibliography on computer graphics.
Computer Science Department, Stanford University, Rept. No. STAN-CS-72-306, (Aug 1972).

<Po 72b> Pollack, B.W.
Compiler Techniques.
Auerbach Publishers, Inc., (1972).

<Po 73> Pollack, B.W.
Using the GRAPL system.
Computer Science Department, Stanford University, (forthcoming).

<Pr 71> Prince, M.D.
Interactive graphics for computer aided design.
Addison-Wesley Publishing Co., (1971).

<QD 72> Quam, L.H., and Diffie, W.
Stanford LISP 1.6 manual.
Computer Science Department, Stanford University,
Rept. No. SAILON 28.6, (1972).

<Re 70> Reynolds, J.C.
GEDANKEN -- a simple typeless language which permits functional data structure and coroutines.
Comm ACM 13, 5 (May 1970), 308-319, and as
Argonne National Laboratory,
Tech. Rept. No. ANL-7621, (Sep 1969).

<Ro 63> Roberts, L.G.
Machine perception of three-dimensional solids.
Lincoln Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.,
Tech. Rept. No. 315, (May 1963), and as
Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Mass., (Feb 1963).

<Ro 70> Romney, G.W.
Computer assisted assembly and rendering of solids.
Computer Science Department, University of Utah, Salt Lake City, Utah, Tech. Rept.
No. TR 4-20, (1970).

<RF 67> Rovner, P.D., and Feldman, J.A.
The LEAP language and data structure.
Lincoln Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., (Oct 1967).

<Ru 70> Rulifson, J.F.
Preliminary specification of the QA4 language.
Artificial Intelligence Center, Stanford Research Institute, Tech. Note 50, (Apr 1970).

<Ru 71> Rulifson, J.F.
QA4 programming concepts.
Artificial Intelligence Center, Stanford Research Institute, Tech. Note 60, (Aug 1971).

<RW 70> Rulifson, J.F., Waldinger, R.J., and Derksen, J.A.
QA4 working paper.
Artificial Intelligence Center, Stanford Research Institute, Tech. Note 42, (Oct 1970).

<RD 72> Rulifson, J.F., Derksen, J.A., and Waldinger, R.J.
QA4: a procedural calculus for intuitive reasoning.
Artificial Intelligence Center, Stanford Research Institute, Tech. Note 73, (Nov 1972).

<Sa 68> Sandewall, E.J.

LISP A: A LISP-like system for incremental computing.
Proc AFIPS 1968 SJCC, vol 32, 375-384.

<Sm 70> Smith, D.C.
MLISP.
Computer Science Department, Stanford University, Rept. No. SAILON 135, CS 179, (Oct 1970).

<SE 73> Smith, D.C., and Enea, H.J.
MLISP2.
Computer Science Department, Stanford University, Rept. No. STAN-73-CS-356, (May 1973), and as Stanford Artificial Intelligence Laboratory Memo. No. AIM 195.

<St 67> Standish, T.A.
A data definition facility for programming languages.
Doctoral Dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania, (May 1967).

<St 69> Standish, T.A.
Some features of PPL, a polymorphic programming language.
In <CS 69>, 20-26.

<St 66> Strachey, C.
Towards a formal semantics.
In Formal Language Description Languages, North-Holland Publishing Co., Amsterdam, (1966).

<Su 72> Sussman, G.J.
Why conniving is better than planning.
Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Memo No. 255, (Feb 1972).

<Su 63> Sutherland, I.E.
SKETCHPAD: a man-machine graphical communication system.
Lincoln Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.,
Tech. Rept. No. 296, (Jan 1963), condensed version in Proc AFIPS 1963 SJCC, vol 22, 329-346.

<SS 74> Sutherland, I.E., Sproull, R.F., and Schumacker, R.A.
A Characterization of 10 Hidden-Surface Algorithms.
Comp Surveys 6, 1 (Mar 74).

<Te 66> Teitelman, W.
PILOT: a step toward man-computer symbiosis.

Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., Tech. Rept. No. MAC-TR-32, (Sep 1966).

<TW 71> Tou, J.T., and Wegner, P., (eds.) Proceedings of a symposium on data structures in programming languages. SIGPLAN Notices 6, 2 (Feb 1971).

<UC 72> University of California at Berkeley CALIDOSCOPE Control Statements University of California at Berkeley, (Nov 1972).

<VD 71> van Dam, A. Data and storage structures for interactive graphics. In <TW 71>, 237-267.

<Wa 68> Warnock, J.E. A hidden line algorithm for half-tone picture representation. Computer Science Department, University of Utah, Salt Lake City, Utah, Tech. Rept. No. 4-5, (May 1968).

<Wa 69> Warnock, J.E. A hidden surface algorithm for computer generated halftone pictures. Computer Science Department, University of Utah, Salt Lake City, Utah, Tech. Rept. No. 4-15, (Jun 1969).

<Wa 70> Watkins, G.S. A real-time visible surface algorithm. Computer Science Department, University of Utah, Salt Lake City, Utah, Rept. No. UTEC-CSc-70-101, (Jun 1970).

<We 71> Wegner, P. Data structure models for programming languages. In <TW 71>, 1-54.

<WS 70> Wehrli, R., Smith, M.J., and Smith, E.F. ARCAID: The ARChitect's computer graphics AID. Computer Science Department, University of Utah, Salt Lake City, Utah, Rept. No. UTEC-CSc-70-102, (Jun 1970).

<We 66> Weiss, R.A. BE VISION, a package of IBM 7090 Fortran programs to draw orthographic views of combinations of plane and quadric surfaces.

Journal ACM 13, 2 (Apr 1966), 194-204.

<WT 71> Weston, P.E., and Taylor, S.M.
CYLINDERS: a relational data structure.
In <TW 71>, 398-416.

<Wi 71> Winograd, T.
Procedures as a representation for data in a computer program for understanding natural language.
Project MAC, Massachusetts Institute of Technology, Tech. Rept. No. MAC-TR-84, (Feb 1971).

<Wi 70> Winston, P.H.
Learning structural descriptions from examples.
Project MAC, Massachusetts Institute of Technology, Tech. Rept. No. MAC-TR-76, (Sep 1970).

<WR 71> Wulf, W.A., Russell, D.B., and Habermann, A.N.
BLISS: a language for systems programming.
Comm ACM 14, 12 (Dec 1971), 780-790.

<WR 67> Wylie, C., Romney, G., Evans, D.C., and Erdahl, A.
Half-tone perspective drawings by computer.
Proc AFIPS 1967 FJCC, vol 31, 49-58.